進化計算

はじめに

科学や工学、そしてビジネスの分野においても最適化は重要な位置を占めている。最適化のためのアルゴリズム、つまり計算機上で実行可能な手法は数多く提案されており、その歴史も長い。興味深いことに、様々な問題に対して"最適"な結果を示す最適化手法の存在は、"No free lunch"という言葉で否定されている。例えば Deep learning の重みパラメータの最適化には SGD や Adam のような勾配法が用いられており、多目的最適化ではヒューリスティックな手法がよく使われる。CAE でよく見る連立一次方程式の求解には Krylov 部分空間法による最適化が主流であるし、非線型連続体力学 (流体力学は除く) の CAE では Newton-Raphson 法による求解がよく用いられている。このような事情を踏まえると、エンジニアとして様々な最適化手法に触れておくことは重要だとつくづく思い知らされる。

数ある最適化アルゴリズムの中で、本資料では進化計算について議論する [1]。No free lunch を紹介して直ぐに一つの最適化アルゴリズムを褒めすぎるのは憚られるが、ブラックボックス関数など様々な問題に適用可能なことや、並列計算機との親和性などゆえに、進化計算は非常に人気な手法だと言える。

もちろん素晴らしい書籍 [1, 2] も存在するので、今更自作資料を作成する意味はないかもしれない。しかしながら、エンジニアリングでよく出会う実数値数ベクトルの最適化に限定し、かつ単目的最適化のみ議論することでコンパクト化された資料が欲しいと思い、この度執筆した次第である。また、資料作成と平行して Python モジュールを自作した。Github で公開しているので (https://github.com/numerical-engine/mimic)、ご興味があれば参照頂きたい。

本資料で取り上げない多目的最適化のための進化計算については、例えば [2, 3] を参考にされたい。マルチモーダルな目的関数における進化計算は [4] が良くまとめてくれている。また、生物の進化を模倣した手法には進化計算の他にも遺伝的アルゴリズムや進化戦略、遺伝的プログラミングなどがあり、初学者に対する混乱の原因となっている。それぞれの区別については [1] の説明が詳しい。

本資料の構成は以下の通りである。1 章は単目的最適化の定義をしたのちに、進化計算の基本的な流れを紹介する。2 章では進化計算において必要な処理、つまり初期候補解のサンプリングや選択、交叉、突然変異、並びに生存選択の具体的手法について紹介する。3 章では制約付きの最適化問題に対するアプローチを議論する。

1 事前準備

1.1 単目的最適化に関する諸定義

前述の通り、最適化アルゴリズムとは最適化問題を計算機上で取り扱える形にしたものである。そのため、最適化問題自体も数式で定義される。進化計算の話題に入る前に、この定義を確認することから始めたい。

まず、最適化したいパラメータ (例えば航空機の翼の長さや厚みなど) を数ベクトル $\boldsymbol{x} = (x_1, ..., x_n)^{\mathrm{T}} \in \mathbb{R}^n$ で纏める。ここで $n \in \mathbb{N}$ はパラメータの個数である。ちなみに $\boldsymbol{x}$ の取り得る値が $\mathbb{R}^n$ 中の任意の元であることは稀である。例えば航空機の翼の長さがゼロ以下に成り得ることは無いし、1000 m になることもきっと無いであろう。実際に取り得る $\boldsymbol{x}$ の全体集合のことを**実行可能領域**と言い、$\Omega \subset \mathbb{R}^n$ と書き表すことにする。また $\boldsymbol{x} \in \Omega$ を満たす $\boldsymbol{x}$ のことを**実行可能解**と言う。

$\boldsymbol{x}$ に対する目的関数値を $y \in \mathbb{R}$ と書くことにする。また $f(\boldsymbol{x}) = y$ に関する関数 $f : \mathbb{R}^n \to \mathbb{R}$ を考えたとき、最適化は

$$\text{minimize} \quad f(\boldsymbol{x}) \quad s.t. \quad \boldsymbol{x} \in \Omega \tag{1}$$

のように書くことができる。ここで、最適化=最小化と仮定した ($f(\boldsymbol{x})$ の最大化問題を考えたい場合は、代わりに $g(\boldsymbol{x}) = -f(\boldsymbol{x})$ の最小化問題を考えればよい)。

1.2 進化計算の諸定義と基本的な流れ

進化計算とはその名の通り、生物の進化を模倣した手法である。具体的には、自然界で見られる自然淘汰と多様性の維持機能を参考にし、最適解の収束とその道中にある局所最適解からの脱出を試みる。自然淘汰は優れた個体のみを残すように働くため、最適化アルゴリズムでも望ましくない解から離れるように働いてくれる。一方の多様性維持は、目的関数値が悪化する方向への解の更新も許容する仕組みと言い換えることができ、それゆえ局所最適解からの脱出に貢献する。このような相反する 2 つの圧力をアルゴリズムに落とし込んだ手法の一つこそが進化計算である。

地球という一つの環境に同種の生物が多数存在するように、進化計算も一つの最適化問題の下に多数の候補解を用意して最適化に挑む。進化計算では候補解の集合のことを特別に**集団 (population)** と言い、

$$P = \{\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_N\} \tag{2}$$

のように書き表す。また、集団の各候補解のことを**個体 (individual)** と呼ぶことが多い。

進化計算はこの集団に対し、各個体の目的関数値が改善されていく方向に更新 (つまり世代交代) させていく。多くの進化計算アルゴリズムでは、現世代の集団 P から目的関数値等に従い適当な個体を**選択 (selection)** し、新しい集団 P_p を生成する (選択された個体が重複していても構わない)。次に選択された個体同士で**交叉 (recombination もしくは crossover)** をさせ、子孫に相当する個体、並びに子孫で構成された集団 P_o を生成する。同じ親から生まれても性格が大きく異なる兄弟がいるように、進化計算における子孫生成も確率的であってよい。この不確かさは集団の多様性維持に貢献する。次に、自然界で見られる**突然変異 (mutation)** を模倣して、P_o に含まれる各個体に適当な値 $\boldsymbol{\delta m}$ を付与する。(本当なのかは知らないが) 突然変異が決定論的に説明できないように、付加する $\boldsymbol{\delta m}$ も確率変数とする。最後に 2 つの集団 P_p と P_o(もしくは P_o のみ) から次の世代に進む固体を選択する。この選択を特別に**生存選択 (survivor selection)** と言う。

Algorithm 1 進化計算

1: ランダムサンプリングにより初期候補解 $P = \{\boldsymbol{x}_i | i = 1, ..., N\}$ を生成。
2: $Y = \{f(\boldsymbol{x}_i) | i = 1, ..., N\}$。
3: Y に従い P から親となる個体を選択し、集団 $P_p = \{\boldsymbol{x}'_i | i = 1, ..., \mu \ \ \boldsymbol{x}'_i \in P\}$ を得る。
4: P_p から複数個体をサンプリング及び交叉を行い、子孫 $\boldsymbol{x}''$ を生成する。必要な数の子孫が得られるまでこれを繰り返し、$P_o = \{\boldsymbol{x}''_i | i = 1, ..., \lambda\}$ を得る。
5: P_o に対し突然変異を施す。
6: $P_p \cup P_o$ に対し生存選択を施す。P を選択された集合に更新する。
7: 収束判定基準を満たしている場合は計算を終了する。それ以外の場合は (2) に戻る。

以上の選択から生存選択までの流れが、基本的な進化計算の世代交代である。これを反復的に繰り返すことで、最終的に集団内に最適解が含まれていることを目指す。Algorithm1 に進化計算の基本的なアルゴリズムを示す。なお、全ての進化計算アルゴリズムが上記に従う訳ではない。手法によっては交叉や突然変異がなかったりするし、交叉と突然変異処理が明確に分けられていない場合もある。また、多くの生物がオスとメスの間で交叉する一方で、進化計算アルゴリズムでは 3 体以上の個体による交叉も提案されている。あくまで計算機上での処理なので、生物の進化より発想がいくらか自由である。

2 進化計算の構成要素

2.1 初期候補解の生成

前章の通り進化計算の計算には集団が必要なため、初期候補解をランダムにサンプリングする方法も考えなければならない。月並みだが一般的な方法を以下に纏めた。

- $\boldsymbol{x} \in \Omega$ を満たすように一様分布でサンプリングする。
- ガウス分布やスチューデントの t 分布など、何か適当な確率分布に従いサンプリングする。確率分布の期待値が最適解 $\boldsymbol{x}^*$ に近い場合、少ない探索回数で最適解に収束することが期待できる。逆に期待値と最適解が大きく離れている場合、アルゴリズムには探索的機能が求められる。
- 実行可能領域 Ω の部分集合 Ω' に対して一様分布でサンプリングする。これは、$\Omega = \mathbb{R}^n$ などのようにいわゆる閉じていない実行可能領域の場合や、もしくは最適解 $\boldsymbol{x}^*$ についてある程度目星がついているときに探索回数を減らすための場合によく用いられる。

2.2 選択

本節では子孫生成を担う個体、つまり親を選択するための手法をいくつか紹介する。生物の進化における競争原理に従えば、優れた個体ほど遺伝子継承の機会を得られる。従って進化計算における選択でも、優れた個体を優先的に選択するようなものでなければならない。一方で極端な選択圧力は集団の多様性を損なわせる。例えば、集団 P のうち最も優れた 1 個体 $\boldsymbol{x}_i$ のみを重複を許して選択する場合、つまり選択により得た集団 P_p が $P_p = \{\boldsymbol{x}_i, \boldsymbol{x}_i, \boldsymbol{x}_i, \boldsymbol{x}_i, ...\}$ みたいになってしまう場合、その後の交叉や突然変異では挽回できないくらい集団の多様性がなくなってしまう。

確かにこれは極端な例だが、劣る個体も子孫生成の機会が与えられるような選択が必要だと言うことは一概に言えるだろう。過度な選択により集団の多様性が損なわれることを**退化**と言う。選択は競争原理に基づく解の改善を担う反面、退化に対しても配慮しなければならない。

2.2.1 FPS(Fitness Proportional Selection)

最も単純な手法として FPS(Fitness Proportional Selection) がある。これは名前が示す通り、目的関数値に比例した確率値に従い、集団から個体を選択する。集団 $P = \{\boldsymbol{x}_i | i = 1, ..., N\}$ の各個体 $\boldsymbol{x}_i$ の目的関数値を y_i とする。このとき、例えば最適化=最大化である場合、各個体を確率 $p_i = y_i / \sum_i y_i$ に従って i.i.d サンプリングし、親として選択された集合 P_p を収集すればよい。こうすることで、y_i が大きい個体ほど優先的に選択しつつも、y_i が低い個体も選択する可能性を残している。なお、最適化=最小化である場合、目的関数値 y_i を

$$y'_i = y_{\max} - y_i \tag{3}$$

のように変換してから同様の処理を行うことが多い。Algorithm2 に FPS のアルゴリズムを示す。

FPS の原理は単純であるものの、残念ながら実際はあまり使われていない。その主な理由として、FPS が目的関数値の差分ではなく比率に着目している点が挙げられる。

例えば目的関数 $f(\boldsymbol{x})$ の代わりに $f(\boldsymbol{x}) + L$ を考えたとする。ここで $L \in \mathbb{R}$ は任意の定数である。このとき、個体 $\boldsymbol{x}_i$ が選択される確率 p_i は

$$p_i = \frac{y_i + L}{\sum_i (y_i + L)} = \frac{y_i/L + 1}{\sum_i (y_i/L + 1)}$$

となる。もし $y_i << L$ であるぐらい L が大きい場合、上式より、$p_i \sim 1/N$ となる。これは如何なる個体も平等に選択されること、つまり選択圧力が全くないことを意味する。

目的関数値の勾配に着目している勾配法では、目的関数値のグローバルな変化から影響を受けなかった。そのため L がどのような値であっても問題なかった訳だが、FPS の場合は注意しなければならない。問題によっては選択圧力を高めるために、素の目的関数から適当な定数を引かなければならない。逆に選択圧力が高すぎる場合は、定数を加えることで選択圧力

Algorithm 2 FPS

Require: 集団 $P = \{\boldsymbol{x}_i | i = 1, ..., N\}$, 目的関数値 $Y = \{y_i | i = 1, ..., \mu\}$, サンプリング数 μ
Ensure: 選択された個体の集団 P_p
1: 各個体の確率値を計算。$prob = \{y_i / \sum_i y_i | i = 1, ..., N\}$。
2: $P_p = \{\}$
3: **for** $n = 1$ to μ **do**
4: $prob$ に従い個体 $\boldsymbol{x}$ をサンプリング。
5: $append(P_p, [\boldsymbol{x}])$
6: **end for**
7: **return** P_p

の緩和を狙うことができる。それでも、生の目的関数を使うのではなく上式の L を上手く調整する作業が必要であるゆえに、FPS は敬遠されている。

2.2.2 ランキング選択

目的関数値のグローバルな変化に影響を受けないような手法としてランキング選択が提案された。これは、目的関数値を直接用いた FPS とは違い、目的関数値の順位を基にサンプリングされる確率を計算する手法である。

まず、集団 $P = \{\boldsymbol{x}_i | i = 1, ..., N\}$ の各個体の目的関数値を $Y = \{y_i | i = 1, ..., N\}$ とする。集団 P の中で個体 $\boldsymbol{x}_i$ の目的関数値が、j 番目に優れている場合、$\boldsymbol{x}_i$ のランク r_i を $N - j$ と定義する。そして、$\boldsymbol{x}_i$ が選択される確率 p_i を、

$$p_i = \frac{2 - s}{N} + \frac{2r_i(s - 1)}{N(N - 1)} \tag{4}$$

に従い設定する。ここで s は $1 \leq s \leq 2$ を満たすハイパーパラメータである。$s = 1$ のとき $p_i = 1/N$、つまり全く選択圧力がない状態になる。一方で s が大きくなるにつれランクの高い個体への選択圧力が高まる。

式 (3) の場合ランクと確率に線形関係が成り立つが、その代わりに指数関数的な

$$p_i = \frac{1 - e^{-r_i}}{c} \tag{5}$$

も提案されている。ここで c は規格化定数である。式 (4)(5) のいずれにしても、選択圧力は目的関数値のグローバルな変化に影響を受けない。Algorithm3 にランキング選択のアルゴリズムを示す。

2.2.3 トーナメント選択

FPS やランキング選択では、一度すべての個体の目的関数値を評価し、各個体が選択される確率を計算していた。確率というバラつきのある選択手法を取ることで、集団の退化を防いでいた訳である。しかしながら、すべての個体の目的関数値を評価することが難しい問題もあったりする。

まず、すべての個体の目的関数値を計算する計算コストは個体数に比例する。そのため、非常に多くの個体数を要する問題では、計算コスト的に求解が不可能になってしまうことがある。

また、そもそも目的関数値自体の計算が難しい問題もある。例えば将棋の方策を最適化する進化計算を考えてみる。目的関数は恐らく”将棋の強さ”になると思うが、これを定量的に評価することは難しい。異なる 2 つの方策の優劣を決めるならば、目的関数値というよく分からないもので評価するよりも、互いに将棋を打たせてその勝敗で決めた方が早い。このように、目的関数の定義は難しいが目的に対する優劣の判断が楽な最適化問題は現実にも数多く存在する。そういった問題では FPS は扱いにくい (ランキング選択は原理的に可能だが長い計算時間を要する)。

本項で紹介するトーナメント選択は、正に上記の欠点を解決する手法であり、多くの問題で採用されている (実際、pymoo[5] のデフォルト設定はトーナメント選択になっている)。大雑把にアルゴリズムを説明すると、N 個の個体がいる集団の中から $k(> 1)$ 個の個体を選択し、選択された個体の目的関数値を相対評価する (つまりランキングを作成する)。そして確率 p で最も優れた個体を選択し、確率 $(1 - p)$ で k 個の中からランダムに個体を一つ選択する。ここで選択された個体は親として取り扱われる。この処理を必要な親の個体数 μ だけ繰り返す。

このように、トーナメント選択では、すべての個体の目的関数値を評価する必要がなく、かつランキング選択のように目的関数値の相対評価を基に親の選択をする。加えて、トーナメント選択のハイパーパラメータの調整が簡単だと言われているため、人気な手法だと思われる。Algorithm4 は厳密なトーナメント選択のアルゴリズムである。

Algorithm 3 ランキング選択

Require: 集団 $P = \{\boldsymbol{x}_i | i = 1, ..., N\}$, 目的関数値 $Y = \{y_i | i = 1, ..., N\}$, サンプリング数 μ
Ensure: 選択された個体の集団 P_p
1: 各個体の確率値 $prob$ を式 (4) もしくは (5) に従い計算。
2: $P_p = \{\}$
3: **for** $n = 1$ to μ **do**
4: $prob$ に従い個体 $\boldsymbol{x}$ をサンプリング。
5: $append(P_p, [\boldsymbol{x}])$
6: **end for**
7: **return** P_p

Algorithm 4 トーナメント選択

Require: 集団 $P = \{\boldsymbol{x}_i | i = 1, ..., N\}$, 目的関数値 $Y = \{y_i | i = 1, ..., N\}$, サンプリング数 μ, トーナメント数 k, 確率値 p
Ensure: 選択された個体の集団 P_p
1: $P_p = \{\}$
2: **for** $n = 1$ to μ **do**
3: P からランダムに k 個の個体 $K = x'_1, ..., x'_k$ を選択する。
4: 確率 p で K から最も優れた個体 $\boldsymbol{x}$ をサンプリング。確率 $(1-p)$ で K からランダムに一つ個体 $\boldsymbol{x}$ をサンプリング。
5: $append(P_p, [\boldsymbol{x}])$
6: $P = P \setminus K$(optional)
7: **end for**
8: **return** P_p

6 行目に書かれているように、一度トーナメントに参加した個体を以後のトーナメントに対して参加可能にすることも不可能にすることもできる。

2.3 交叉

本節で紹介する交叉は集団の多様性維持を担う。自然界の生物も交叉によって子孫を生み出すように、進化計算の交叉も親の個体から新しい個体、つまり子孫を生み出す。前節の手法で選択された親は概ね優れた解であった。交叉は、そういった親の特徴を残しつつ、かといって単なる親の複製とならないように子孫を生成する。親の複製でないゆえに交叉は多様性の維持に貢献するが、親とどの程度異なる子孫を生成し得るかによって、貢献度合いは変わっていく。例えば"選択"の選択圧が強い場合、交叉もそれ相応の探索圧力を持たなければならない。

なお、多様性の維持のために、進化計算は交叉以外にも突然変異も用意している。突然変異は一つの個体に作用して新しい個体を生み出すが (後述)、一方の交叉は複数の親に作用して複数の子孫を生み出す。交叉と突然変異の役割分担や対比に関して、私は一般的な見解を知らない。ただし、進化計算によっては突然変異のみ、もしくは交叉のみを採用していることもある。

2.3.1 一様交叉

一様交叉は個体の表現がビット列である遺伝的アルゴリズムでよく使われている。原理的には実数値数ベクトルに対しても適用可能なので、本資料でも紹介することにした。一般的に、一様交叉は 2 つの親から 2 つの子孫を生み出す。2 つの親の解を $\boldsymbol{x}_i = (x_{i1}, ..., x_{in})^{\mathrm{T}}$ および $\boldsymbol{x}_i = (x_{j1}, ..., x_{jn})^{\mathrm{T}}$ としたとき、ベクトルの各要素を確率 p(一般的に 0.5) で交換する。アルゴリズムを Algorithm5 に示す。

定義より、2 つの親を対角の頂点とした超立方体を考えたとき、一様交叉によって得られる子孫は必ず超立方体の頂点となる (図 1(a))。

2.3.2 WAC(Whole Arithmetic Crossover)

実数値数ベクトルのための交叉として WAC がある (WAC という略記は一般的ではないが、日本語訳を知らないゆえ文字数削減のために本資料では WAC と呼ぶことにする)。WAC では 2 つの親 $\boldsymbol{x}_i$ と $\boldsymbol{x}_j$ から

$$\begin{aligned} \boldsymbol{u} &= \alpha \boldsymbol{x}_i + (1-\alpha)\boldsymbol{x}_j \\ \boldsymbol{v} &= \alpha \boldsymbol{x}_j + (1-\alpha)\boldsymbol{x}_i \end{aligned} \tag{6}$$

Algorithm 5 一様交叉

Require: 親の集団 $P_p = \{\boldsymbol{x}_i | i = 1, ..., \mu\}$, サンプリング数 λ, 確率値 p
Ensure: 子孫の集団 P_o
1: $P_o = \{\}$
2: **while** $|P_o| < \lambda$ **do**
3: P_p からランダムに 2 個体 $\boldsymbol{x}_i$, $\boldsymbol{x}_j$ をサンプリング
4: n 次元ベクトル $\boldsymbol{u}$ と $\boldsymbol{v}$ を定義
5: **for** $k = 1$ to n **do**
6: [0, 1] の一様分布から実数 r をサンプリング
7: **if** $r < p$ **then**
8: $u_k = x_{ik}$, $v_k = x_{jk}$
9: **else**
10: $u_k = x_{jk}$, $v_k = x_{ik}$
11: **end if**
12: **end for**
13: $append(P_o, [\boldsymbol{u}, \boldsymbol{v}])$
14: $P_p = P_p \setminus [\boldsymbol{x}_i, \boldsymbol{x}_j]$(optional)
15: **end while**
16: **return** P_o

Algorithm 6 Whole Arithmetic Crossover

Require: 親の集団 $P_p = \{\boldsymbol{x}_i | i = 1, ..., \mu\}$, サンプリング数 λ, パラメータ範囲 $[\alpha_{min}, \alpha_{max}]$
Ensure: 子孫の集団 P_o
1: $P_o = \{\}$
2: **while** $|P_o| < \lambda$ **do**
3: P_p からランダムに 2 個体 $\boldsymbol{x}_i$, $\boldsymbol{x}_j$ をサンプリング
4: α を $[\alpha_{min}, \alpha_{max}]$ よりサンプリング
5: 式 (6) より $\boldsymbol{u}$ と $\boldsymbol{v}$ を計算
6: $append(P_o, [\boldsymbol{u}, \boldsymbol{v}])$
7: $P_p = P_p \setminus [\boldsymbol{x}_i, \boldsymbol{x}_j]$(optional)
8: **end while**
9: **return** P_o

の 2 つの子孫を生成する。ここで α は実数であり、固定値でも乱数でもよい。上式より、子孫は $\boldsymbol{x}_i$ と $\boldsymbol{x}_j$ の線状に位置する。例えば α を [0, 1] の一様乱数としたとき、子孫は $\boldsymbol{x}_i$ と $\boldsymbol{x}_j$ の線分上の点となる。一般的には α を [-0.25, 1.25] の一様乱数とし、多少の外挿を許すことが多い。アルゴリズムを Algorithm6 に、サンプリング結果を図 1(b) に示す。

2.3.3 BLX-α(Blend Crossover)

一様交叉のように実数値数ベクトルの要素毎に独立して作用し、WAC のように線形和を施す手法として、BLX-α が提案されている [6]。BLX-α は 2 つの親 $\boldsymbol{x}_i$ と $\boldsymbol{x}_j$ から一つの子孫 $\boldsymbol{u}$ を生成する。まず、$\boldsymbol{x}_i$ と $\boldsymbol{x}_j$ を対角の頂点とした超立方体を考える。BLX-α はこの超立方体の重心を変えないように各辺の長さを $(1 + 2\alpha)$ 倍する。ここで α はパラメータ定数である ([6] によると、$\alpha = 0.5$ のとき最も良い性能を示したらしい)。そして拡大された超立方体内からランダムに解をサンプリングし、それを子孫として出力する。具体的なアルゴリズムは Algorithm7 の通りである。

BLX-α は 2 つの親から多様な子孫を生成できるため、突然変異が無くても良い性能を示す手法として知られている (図 1(c))。

2.3.4 SBX(Simulated Binary Crossover)

遺伝的アルゴリズムで得られた交叉に関する数多くの知見を実数値数ベクトルの最適化問題に転用させた手法として、SBX というものがある [7]。Deb らの数値実験によると、BLX-0.5 よりも高い性能を示すことがあるらしい。また、NSGA-II などでも採用されている。

簡単のために、初めに個体表現がスカラーの場合のアルゴリズムを紹介する。SBX は 2 つの親 x_1 と x_1 から 2 つの子孫を生成する。$x_{max} = \max(x_1, x_2)$ 及び $x_{min} = \min(x_1, x_2)$ と定め、さらに $d = x_{max} - x_{min}$ とする。次に

$$\begin{aligned} \beta_1 &= 1 + \frac{2(x_1 - x_l)}{d} \\ \beta_2 &= 1 + \frac{2(x_u - x_2)}{d} \end{aligned} \tag{7}$$

を計算する。ここで (x_l, x_u) は変数の上下限値である。更に、

$$\begin{aligned} \alpha_1 &= 2 - \beta_1^{-(\eta+1)} \\ \alpha_2 &= 2 - \beta_2^{-(\eta+1)} \end{aligned} \tag{8}$$

を計算する。ここで η はパラメータ定数であり、pymoo[5] では 15 をデフォルト値としている。次に [0,1] から乱数 u を生成し、下記の式に従い $\beta_{qi} (i = 1, 2)$ を計算する。

$$\beta_{qi} = \begin{cases} (u\alpha_i)^{1/(\eta+1)} & \text{if} \quad (u \leq \alpha_i^{-1}) \\ \left(\frac{1}{2 - u\alpha_i}\right)^{1/(\eta+1)} & \text{otherwise} \end{cases} . \tag{9}$$

Algorithm 7 BLX-α

Require: 親の集団 $P_p = \{\boldsymbol{x}_i | i = 1, ..., \mu\}$, サンプリング数 λ, パラメータ α
Ensure: 子孫の集団 P_o
1: $P_o = \{\}$
2: **while** $|P_o| < \lambda$ **do**
3: P_p からランダムに 2 個体 $\boldsymbol{x}_i$, $\boldsymbol{x}_j$ をサンプリング
4: 任意の n 次元数ベクトル $\boldsymbol{u}$ を定義
5: **for** $k = 1$ to n **do**
6: $x_{max} = \max(x_{ik}, x_{jk})$, $x_{min} = \min(x_{ik}, x_{jk})$, $d = x_{max} - x_{min}$
7: $[x_{min} - \alpha d, x_{max} + \alpha d]$ からランダムに実数 v をサンプリング。$u_k = v$
8: **end for**
9: $append(P_o, [\boldsymbol{u}])$
10: $P_p = P_p \setminus [\boldsymbol{x}_i, \boldsymbol{x}_j]$(optional)
11: **end while**
12: **return** P_o

Algorithm 8 SBX

Require: 親の集団 $P_p = \{\boldsymbol{x}_i | i = 1, ..., \mu\}$, サンプリング数 λ, パラメータ η、確率値 p、上下限値 $(\boldsymbol{x}_l, \boldsymbol{x}_u)$
Ensure: 子孫の集団 P_o

$P_o = \{\}$
while $|P_o| < \lambda$ **do**
  P_p からランダムに 2 個体 $\boldsymbol{x}_i$, $\boldsymbol{x}_j$ をサンプリング
  任意の n 次元数ベクトル $\boldsymbol{u}$ と $\boldsymbol{v}$ を定義
  for $k = 1$ to n **do**
    [0,1] より乱数 r を生成
    if $r < p$ **then**
      [0,1] より乱数 u を生成
      式 (9) 及び x_{ik}、x_{jk} より、β_{q1} と β_{q2} を計算する
      u_k 及び v_k を式 (10) より求める
    end if
  end for
  $append(P_o, [\boldsymbol{u}, \boldsymbol{v}])$
  $P_p = P_p \setminus [\boldsymbol{x}_i, \boldsymbol{x}_j]$(optional)
end while
return P_o

最後に子孫 u_1 及び u_2 を以下の式より求める。

$$\begin{aligned} u_1 &= 0.5\{(x_1 + x_2) - \beta_{q1}(x_2 - x_1)\} \\ u_2 &= 0.5\{(x_1 + x_2) - \beta_{q2}(x_2 + x_1)\} \end{aligned} \tag{10}$$

以上が個体表現がスカラー値の場合の SBX である。個体表現がベクトルの場合は、各要素に対し確率 p で独立に上記処理を行う。なお、pymoo[5] の p のデフォルト値は 0.9 となっている。

個体表現がベクトル場合の SBX のアルゴリズムを Algorithm8 に、サンプリング結果を図 1(d) に示す。

2.4 突然変異

本節では突然変異について議論していく。前述の通り、突然変異は交叉により生成された子孫に施すことが多い。突然変異により個体表現の変化は完全にランダムだと言ってよい。したがって、突然変異は集団の多様性維持のためにある。

2.4.1 自己適応的にガウス分布の分散共分散行列を調整する手法

突然変異でも様々な手法が提案されているが、よく目にするものは個体表現 $\boldsymbol{x}$ にガウス分布に従う乱数 $\boldsymbol{\epsilon}$ $\mathcal{N}(\boldsymbol{0}, \Sigma)$ を足す手法であろう。つまり、突然変異後の個体表現 $\boldsymbol{x}'$ は

$$\boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{\epsilon} \tag{11}$$

となる。突然変異がどれ程多様な個体を生成するかは、分散共分散行列 Σ に依存する (もちろん分散値が高い方が多様な解を得やすい)。それゆえ分散共分散行列の調整は重要だと言え、その値を自己適応的に調整する手法も数多く提案されている。最も有名なものに CMA-ES[8] があるが、本資料ではより単純かつ程々の計算コストで実行できる手法を紹介する。

さて、最適化問題毎に適した分散共分散行列があるとするならば、分散および共分散の値も最適化対象として考えると面白い。つまり、分散及び共分散の値をベクトルとして $\boldsymbol{\sigma} = (\sigma_{11}, \sigma_{12}, ..., \sigma_{nn})^{\boldsymbol{T}}$ のようにまとめたとき、個体表現を本来 $\boldsymbol{x}$ と $\boldsymbol{\sigma}$ を合わせた $(\boldsymbol{x} : \boldsymbol{\sigma}) \in \mathbb{R}^{n+n^2}$ を考える。$\boldsymbol{\sigma}$ の値は目的関数値に影響を与えないが、個体の生存確率に $\boldsymbol{\sigma}$ は影響するだろう。例えば、十分に探索が済んだ終盤の世代では下手に突然変異を施されない方が良いため、$\boldsymbol{\sigma}$ が小さい個体が生き残りやすくなる。逆に序盤の世代では $\boldsymbol{\sigma}$ が大きい個体ほど多くのチャンスを掴むかもしれない。

なお、分散共分散行列の全ての値を加味した場合、個体表現の次元は $n + n^2$ となる。n が大きいと最適化的に非現実的な次元数となるため、実際は共分散をゼロとすることが多い。本資料でも分散値のみを考慮し、$\boldsymbol{\sigma} = (\sigma_1, \sigma_2, ..., \sigma_n)$ とする。ここで σ_i は Σ の第 (i, i) 成分である。

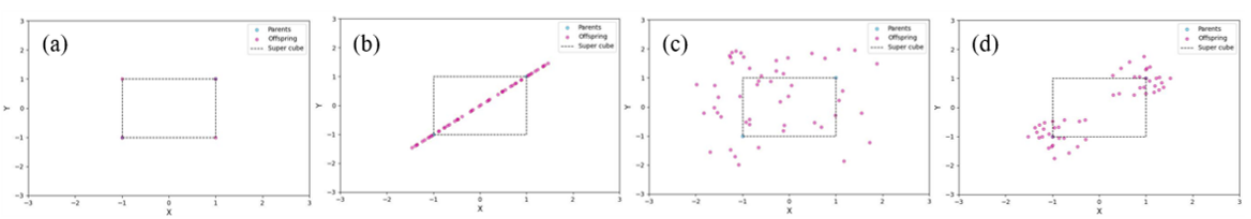

図 1　親の個体表現を (-1, -1) 及び (1, 1) としたときの、交叉により得られる子孫の個体表現分布。(a) 一様交叉 (b)WAC(c)BLX_α(d)SBX。

Algorithm 9 Gaussian mutation

Require: 個体 $(\boldsymbol{x}:\boldsymbol{\sigma})$, τ, τ', 分散下限値 $\boldsymbol{\nu}=(\nu_1,...,\nu_n)^{rmT}$
Ensure: 突然変異後の個体 $(\boldsymbol{x}', \boldsymbol{sigma}')$
1: 任意の n 次元ベクトル $\boldsymbol{x}'$ と $\boldsymbol{\sigma}'$ を定義
2: $\mathcal{N}(0,1)$ に従い ϵ をサンプリング
3: **for** $i=1$ to n **do**
4: $\quad\mathcal{N}(0,1)$ に従い ϵ_i をサンプリング
5: $\quad\sigma_i'$ を式 (12) に従い更新
6: $\quad\mathcal{N}(0,1)$ に従い ξ_i をサンプリング
7: $\quad x_i'$ を式 (13) に従い更新
8: **end for**
9: **return** $(\boldsymbol{x}':\boldsymbol{\sigma}')$

$\boldsymbol{\sigma}$ の更新について、交叉は $\boldsymbol{\sigma}$ を $\boldsymbol{x}$ と同様に扱う。つまり、前節の $\boldsymbol{x}$ に対するアルゴリズムがそのまま $\boldsymbol{\sigma}$ にも適用される。一方で、突然変異の際は各個体の σ_i を

$$\sigma_i' = \max\left\{\sigma_i e^{\tau'\epsilon+\tau\epsilon_i}, \nu_i\right\} \tag{12}$$

に従い更新する。ここで ϵ_i は $\mathcal{N}(0,1)$ に従ってサンプリングされた数値であり、成分 i 毎に用意されたものである。一方の ϵ も $\mathcal{N}(0,1)$ に従ってサンプリングされた数値だが、成分に依らず同じ値を使う。また、τ および τ' はハイパーパラメータであり、$\tau \propto 1/\sqrt{2\sqrt{n}}$、$\tau' \propto 1/\sqrt{2n}$ とすることが推奨されている (n は世代数)[1]。ν_i は各 i で設けられた分散値の下限値である。

突然変異の際、まずは各 σ_i を式 (12) に従い σ_i' に更新し、その後各 x_i を

$$x_i' = x_i + \sigma_i'\xi_i \tag{13}$$

で更新する。ここで ξ_i は $\mathcal{N}(0,1)$ に従ってサンプリングされた数値であり、成分 i 毎に用意されたものである (ϵ_i とは異なることに注意)。上記アルゴリズムを Algorithm9 に纏めた。

2.4.2 PBM(Parameter Based Mutation)

本項では多項式確率分布に従った突然変異の PBM を紹介する。PBM は NSGA-II でも採用されている手法で、個人的に SBX との相性が良い所感がある。個体表現 $\boldsymbol{x}$ の個々の成分 x_i に対する処理は以下の通りである。

まず、区間 [0, 1] 上の一様分布から乱数 u をサンプリングする。そして、δ_q を以下の式より求める。

$$\delta_q = \begin{cases} \left[2u+(1-2u)(1-\delta_l)^{\eta+1}\right]^{\frac{1}{\eta+1}} - 1 & \text{if} \quad u < 0.5 \\ 1-\left[2(1-u)+(2u-1)(1-\delta_u)^{\eta+1}\right]^{\frac{1}{\eta+1}} & \text{otherwise} \end{cases} \tag{14}$$

ここで、η はハイパーパラメータであり、値が小さいほど探索的圧力が大きくなる (pymoo[5] ではデフォルトで 20 に設定されている)。また、

$$\delta_l = \frac{x-x_{il}}{x_{iu}-x_{il}}, \quad \delta_l = \frac{x_{iu}-x}{x_{iu}-x_{il}}$$

であり、x_{il} と x_{iu} はそれぞれ x_i に関する上下限値である。最後に x_i を以下の式に従い更新する。

$$x_i' = x_i + \delta_q(x_{iu}-x_{il}) \tag{15}$$

に従い更新する。以上の処理を各成分に対してそれぞれ p の確率 (ハイパーパラメータであり、pymoo[5] では 0.1) で施す。アルゴリズムを Algorithm10 に纏めた。また、図 2 は $x=0$ に対して PBM を施したときのバラつきをヒストグラムで表したものである。

2.5 生存選択

本節では最後の構成要素の生存選択について議論する。Algorithm1 で説明した通り、進化計算は親の集団 P_p と子孫の集団 P_o から次世代に継承される個体を選択する。原理的には 2.2 節で紹介した手法を利用できるが、多くの場合異なる手法が採用される (それゆえ生存選択と区別して名付けられている)。以下は有名な生存選択手法である。

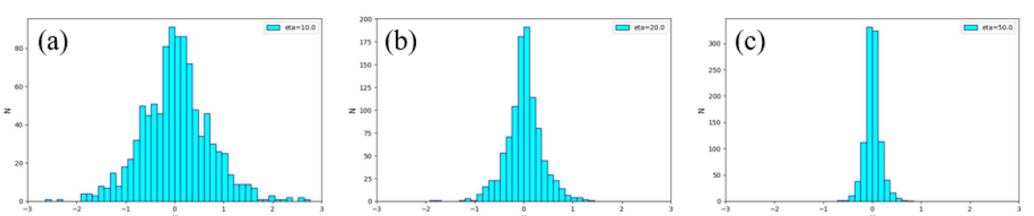

図 2　$x=0$ の個体に対して PBM を施したときのバラつき。(a)$\eta=10$, $\eta=20$, $\eta=50$

Algorithm 10 PBM

Require: 個体 $\boldsymbol{x}$, 各変数上限値 $\boldsymbol{x}_u$, 各変数下限値 $\boldsymbol{x}_l$, 突然変異確率 p
Ensure: 突然変異後の個体 $\boldsymbol{x}'$

```
任意の n 次元ベクトル x' を定義
for i = 1 to n do
    区間 [0, 1] の一様分布から p' をサンプリング
    if p' < p then
        区間 [0, 1] の一様分布から u をサンプリング
        式 (14) に従い δ_q を計算
        式 (15) に従い x'_i を計算
    else
        x'_i = x_i
    end if
end for
return (x' : σ')
```

- **年齢を基に選択する方法**: 最も単純な方法は各個体の年齢を基に選択する方法である。ここで年齢とは、個体が世代交代で生き延びた回数であり、世代数とは別の値である。生存選択では、$P_p \cup P_o$ 中の個体を年齢順で並び替え、最も若い N 個体を次世代に継承する (ここで N は各世代の個体数 (Algorithm1 参照))。
- **GENITOR**: 親の個体数 μ と子孫の個体数 λ について、$\mu > \lambda$ が成り立つとき GENITOR という手法が使える。これは、P_o 中のすべての個体と、P_p のうち最も優れた $\mu - \lambda$ の個体を生存選択する手法である。つまり、次世代に継承される個体数は μ となる (世代を経る度に集団の個体数が変動することは一般的でないので、GENITOR を使うときは $\mu = N$ とすることが多い)。
- $(\mu + \lambda)$ **選択**: 進化戦略の分野でよく用いられる手法に $(\mu + \lambda)$ 選択がある。これは親と子孫を平等に扱い、和集合 $P_p \cup P_o$ のうち最も優れた N 個体を次世代に継承する。
- (μ, λ) **選択**: 親の個体数 μ と子孫の個体数 λ について、$\mu < \lambda$ が成り立つとき (μ, λ) 選択という手法がよく使われる。これは、λ 個の子孫のうち、最も優れた μ 個の個体を次世代に継承する。

3 進化計算における制約の取り扱い

1.2 節で軽く触れたように、多くの問題では個体表現の取り得る値に制限がある。取り得る値の集合を実行可能領域 Ω と言い、進化計算は Ω の元から最適解を見つけ出そうとする。

進化計算の構成要素は 2 章で紹介した通りであるが、交叉や突然変異は制約について何ら配慮していなことに気付く。つまり、Ω の差集合を $\Phi(= \mathbb{R}^N - \Omega)$ とすると、進化計算は世代交代の際に $\boldsymbol{x} \in \Phi$ のような個体を生成できてしまう。そのため、2 章で紹介したものとは別に制約を加味するような機能が必要になる。

最も単純な方法は変数を変換する方法であろう。例えば、$x > 0$ なる制約がある場合、x の代わりに $y \in \mathbb{R}$ を個体表現に採用し、$x = e^y$ なる変換をしてから目的関数値を計算する。しかしながら、いつでも $\psi : \mathbb{R}^n \to \Omega$ なる全単射な関数 ψ が用意できるわけではないため、万能な手法とは言えない。そこで、最適化の分野では様々な手法が研究されてきた。本資料ではペナルティ関数を使う方法とヒューリスティックな最適化手法で適用可能な手法を紹介する。なお、後者は pymoo[5] の多くの最適化手法で採用されている。

3.1 ペナルティ関数による制約の考慮

まず、実行可能領域 Ω を $\{\boldsymbol{x} | g_i(\boldsymbol{x}) < 0, \ \ (i = 1, ..., m)\}$ に置き換える。ここで各 g_i は制約関数である。すべての制約関数は $g_i(\boldsymbol{x}) < 0$ の形で表現できる。例えば $h_i(\boldsymbol{x}) > 0$ なる制約は、$g_i(\boldsymbol{x}) = -h_i(\boldsymbol{x})$ とすればよい。等式制約 $g_i(\boldsymbol{x}) = 0$ は $\boldsymbol{x}$ のうちの適当な成分 x_i について $x_i = h_i(x_1, ..., x_{i-1}, x_{i+1}, ..., x_n)$ なる陽関数を導出し x_i を最適化対象から取り除く。

定義より、$g_i(\boldsymbol{x})$ の値が大きいほど $\boldsymbol{x}$ は i 番目の制約について大きく反していると言える。そこで、制約を満たしていない度合に関する測度、つまりペナルティ関数

$$p_i(\boldsymbol{x}) = \begin{cases} 0 & \text{if} \ \ g_i(\boldsymbol{x}) < 0 \\ |g_i(\boldsymbol{x})|^k & \text{otherwise} \end{cases} \tag{16}$$

を導入する。上式より、$\boldsymbol{x}$ が制約を満たす場合 $p_i(\boldsymbol{x})$ はゼロで、$g_i(\boldsymbol{x})$ が正のとき、$p_i(\boldsymbol{x})$ は $g_i(\boldsymbol{x})$ の k 乗となる (一般的に k には 1 もしくは 2 を用いる)。これを目的関数に足した

$$f'(\boldsymbol{x}) = f(\boldsymbol{x}) + \sum_i w_i p_i(\boldsymbol{x}) \tag{17}$$

を新たな目的関数として最適化を実行する。ここで w_i は正のハイパーパラメータである。

ペナルティ関数を用いた最適化の場合、解の探索範囲は $\mathbb{R}^n$ 全体となる。従って交叉や突然変異の際に $\boldsymbol{x} \notin \Omega$ なる解が生成されても問題ない。ただし、制約を満たさない解は式 (17) の右辺第二項が正である分、$f'(\boldsymbol{x})$ が大きくなり生存競争に負けやすくなる。負けやすさの度合いは $f(\boldsymbol{x})$ と $p_i(\boldsymbol{x})$ の値の相対的な大小に依存するため、問題毎に適切な w_i を設定しなければならない。非常に大きな w_i を設定すれば、制約を満たさない解はほぼ淘汰されるようになる。これは多くの場合望ま

しい傾向であるが、真の最適解が Ω の”端”近くにあるような問題、つまり最適解 $\boldsymbol{x}^*$ について $g_i(\boldsymbol{x}^*)$ がかなりゼロに近い問題の場合、非常に大きな w_i だと解の探査に苦労する。

3.2 Deb らによる手法

前節の手法はやはり w_i の設定が難しい。ペナルティ関数が十分に機能するためには $f(\boldsymbol{x})$ の値にも注意しなければならないためである。そこで、Deb ら [9] はパラメータレスに制約を加味するために、

$$f'(\boldsymbol{x}) \begin{cases} f(\boldsymbol{x}) & \text{if feasible} \\ f_{max} + \sum_i p_i(\boldsymbol{x}) & \text{otherwise} \end{cases} \tag{18}$$

なる代わりの目的関数を提案した。ここで f_{max} は制約を満たす個体のうち、$f(\boldsymbol{x})$ の最大値である。f_{max} の計算が可能なヒューリスティックな最適化らしい手法で、pymoo[5] の多くのアルゴリズムで採用されている。

また、最適化問題によっては制約を満たさない解の目的関数値が求まらないこともある (例えば CAE を介して目的関数値を計算する場合。最適化対象として密度 ρ を考えており、$\rho > 0$ なる制約を設定しているとする。もしも $\rho < 0$ の条件で CAE の計算を始めたら、恐らく発散して目的関数値の計算まで至らないだろう)。このような問題に対して、本手法は制約を満たす解の目的関数値のみ計算すればよいので、うまく対処できる。

参考文献

[1] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Springer, 2015.
[2] K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms*. WILEY, 2009.
[3] B. Xin J. Li and J. Chen. *Decomposition-based Evolutionary Optimization in Complex Environments*. World Scientific, 2021.
[4] M. Preuss. *Multimodal Optimization by Means of Evolutionary Algorithms*. Springer, 2014.
[5] Julian Blank and Kalyanmoy Deb. Pymoo: Multi-objective optimization in python. *IEEE Access*, Vol. 8, pp. 89497–89509, 2020.
[6] Larry J. Eshelman and J. David Schaffer. Real-coded genetic algorithms and interval-schemata. In L. DARRELL WHITLEY, editor, *Foundations of Genetic Algorithms*, Vol. 2 of *Foundations of Genetic Algorithms*, pp. 187–202. Elsevier, 1993.
[7] K DEB. Simulated binary crossover for continuous search space. *Complex Systems*, Vol. 9, pp. 115–148, 1995.
[8] N. Hansen. The cma evolution strategy: A tutorial. *arxiv preprinted*, 2016.
[9] Kalyanmoy Deb and Samir Agrawal. A niched-penalty approach for constraint handling in genetic algorithms. In *Artificial Neural Nets and Genetic Algorithms*, pp. 235–243, Vienna, 1999. Springer Vienna.