1 目的

工学領域における多くの問題は最適化問題として定式化することができ，これに対する解法としてさまざまな手法が研究・開発されている．従来，最適化問題を解くための理論や技術は，数学的な理論に基づく最適化理論や数理計画法といった研究分野で発展してきた．しかし，現実世界の問題には，数学的に解析して解くことが非常に難しい問題や，限られた時間内で最適解を求めることが困難な問題も数多く存在している．このようなことから，こうした現実的な制約のある問題に対しては，進化計算と呼ばれる手法が効果的な解法の一つとして広く利用されるようになった．

進化計算の代表的な手法としては，Holland によって提案された遺伝的アルゴリズム（Genetic Algorithm, GA）や，Kennedy と Eberhart によって提案された粒子群最適化（Particle Swarm Optimization, PSO）などがある．これらの手法は，自然界の進化の仕組みや生物の振る舞いをモデルとした手法である [1].

本実験では，これらの進化計算アルゴリズムの理解を深めるため，基本的なベンチマーク問題を用いた数値実験を Python を用いて実装したプログラムを用いて行う．また，進化計算の工学設計問題の応用として圧力容器設計への応用についても数値実験を行う．

2 粒子群最適化

粒子群最適化（Particle Swarm Optimization, PSO）は，1995 年に Kennedy と Eberhart により提案された確率的な多点探索手法である．PSO のアルゴリズムは，鳥や魚など群れで行動する生物の情報共有の仕組みをモデルとしており，個々の粒子（解候補）が周囲の情報を交換しながら最適解を探索する [2]．この手法は，2000 年頃から多くの工学的問題に応用されるようになり，基礎研究から実用的な応用に至るまで幅広く研究されている．

2.1 PSO のアルゴリズム

PSO のアルゴリズムにはいくつかのバリエーションがあるが，ここでは最も一般的な標準的なアルゴリズムについて説明する．まず最初に，PSO を適用する最適化問題を定義する．簡単のため，以下のような n 次元の目的関数を最小化する問題 [3] を対象とする．この問題の目的は，決定変数 $x_i = [x_1, x_2, \cdots, x_n]^T$（$i = 1, 2, \cdots, n$）を最適化することである．

$$\min_{x_i \in \mathbb{R}^n} f(x_i) \tag{1}$$

PSO では，このような最適化問題を解くため，m 個の粒子が群れを形成し，解空間内を移動しながら最適化を行う．各粒子 j の位置は，問題の解候補に対応している．

$$x_{i,j} = [x_{1,j}, x_{2,j}, \cdots, x_{n,j}]^T \quad (j = 1, 2, \cdots, m) \tag{2}$$

標準的な PSO アルゴリズムでは，以下の更新式を用いて粒子の位置と速度を更新する．

$$v_{i,j}^{k+1} = \omega v_{i,j}^{k} + c^{(1)} r_{i,j}^{(1)k} (\mathrm{pbest}_{i,j}^{k} - x_{i,j}^{k}) + c^{(2)} r_{i,j}^{(2)k} (\mathrm{gbest}_{i}^{k} - x_{i,j}^{k}) \tag{3}$$

$$x_{i,j}^{k+1} = x_{i,j}^{k} + v_{i,j}^{k+1} \tag{4}$$

ここで，各添字 i, j, k はそれぞれ変数次元，粒子番号，および更新回数を表している．PSO では，粒子 $x_{i,j}^{k}$ の更新ごとに，目的関数値 $f(x_{i,j}^{k})$ が評価される．式 (3) の $\mathrm{pbest}_{i,j}^{k}$ は，粒子 j がこれまでの更新で発見した最良解を示し，「personal best」と呼ばれている．gbest_{i}^{k} は群れ全体で発見した最良解を示し，「global best」と呼ばれている．また，$\omega, c^{(1)}, c^{(2)}$ は PSO の制御パラメータであり，$r_{i,j}^{(1)k}, r_{i,j}^{(2)k}$ は [0,1] の範囲で一様に分布する乱数である．$v_{i,j}^{k} = [v_{1,j}^{k}, v_{2,j}^{k}, \cdots, v_{n,j}^{k}]^{T}$ は粒子の速度を示し，式 (3) の右辺第 1 項は「慣性項」とも呼ばれている．このように，PSO アルゴリズムはシンプルな更新ルールに基づき，目的関数の収束や最大更新回数などの終了条件が満たされるまで繰り返し処理を行う．

PSO における粒子の移動の概略図を Fig.1 に示す．

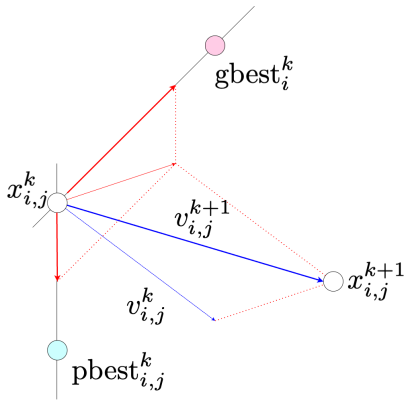

Fig 1　粒子の移動イメージ

標準的な PSO アルゴリズムの処理の流れを以下にまとめる．

STEP1　粒子数 m，重みパラメータ $\omega, c^{(1)}, c^{(2)}$，最大繰り返し回数 $k_{\max}$ を設定する．

STEP2　各粒子の位置 $x_{i,j}^{0}$ と速度 $v_{i,j}^{0}$ を解空間内にランダムに初期化する．

なお，$\mathrm{pbest}_{i,j}^{0} = x_{i,j}^{0}$，$\mathrm{gbest}_{j}^{0} = \mathrm{pbest}_{s,j}^{0}$，$s = \arg\min_{i} f(\mathrm{pbest}_{i,j}^{0})$ である．

STEP3　式 (3) と式 (4) を用いて粒子を更新する．

STEP4　もし $f(x_{i,j}^{k+1}) < f(\mathrm{pbest}_{i,j}^{k})$ なら，$\mathrm{pbest}_{i,j}^{k+1} = x_{i,j}^{k+1}$ とし，そうでなければ，$\mathrm{pbest}_{i,j}^{k+1} = \mathrm{pbest}_{i,j}^{k}$ とする．

また，$\mathrm{gbest}_{j}^{k+1} = \mathrm{pbest}_{s,j}^{k+1}$ とする．ここで，$s = \arg\min_{i} f(\mathrm{pbest}_{i,j}^{k})$ である．

STEP5　$k = k_{\max}$ となった場合，最終的に得られた gbest_{j}^{k+1} を解として出力し，処理を終了する．

そうでなければ，$k = k + 1$ として STEP2 に戻る．

2.2 Python による実装

ここでは、標準的な粒子群最適化（PSO）アルゴリズムを Python で実装する方法について説明する．まず最初に，最適化対象となる目的関数を定義する．代表的なベンチマーク問題である **Rastrigin 関数**を例とする．Rastrigin 関数は，関数値が多くの局所最小値を持つため，最適化アルゴリズムの性能評価に広く用いられている．この関数は次のように定義される．

$$f_{\text{Rastrigin}}(x_i) = \sum_{i=1}^{n} \left(x_i^2 - 10\cos(2\pi x_i) + 10\right) \tag{5}$$
$$(-5.12 \leq x_i \leq 5.12)$$

Rastrigin 関数の実装例を以下に示す．

```python
# ライブラリのインポート
from random import random
from matplotlib import pyplot
import math

# Rastrigin 関数の定義
def fitFunc1(xVals):
    # 初期値として 10 * 次元数の値を設定
    fitness = 10 * len(xVals)
    for i in range(len(xVals)):
        # 各次元の値に対する計算を加算
        fitness += xVals[i]**2 - (10 * math.cos(2 * math.pi * xVals[i]))
    return fitness
```

次に、粒子の位置情報と移動方向（速度）の初期化の実装例を示す．ここでは，粒子群の個数（Np）と次元数（Nd）に基づいて，位置と速度をランダムに設定する．位置は定義域 $[x_{\min}, x_{\max}]$ 内でランダムに生成され，速度は定義域 $[v_{\min}, v_{\max}]$ 内でランダムに生成される．

```python
# 粒子の位置情報の初期化
def initPosition(Np, Nd, xMin, xMax):
    R = [[xMin + random() * (xMax - xMin) for i in range(0, Nd)] for p in range(0, Np)]
    return R
```

```
5
6  # 粒子の移動方向の初期化
7  def initVelocity(Np, Nd, vMin, vMax):
8      V =[[vMin + random() * (vMax - vMin) for i in range(0, Nd)] for p in range(0, Np)]
9      return V
```

次に，粒子の速度ベクトルと位置ベクトルの更新式の実装例を示す．粒子の速度は，慣性項，個別の最良解（pBest），そして群れ全体の最良解（gBest）を基に更新される．これには，式（3）と式（4）の更新式を利用する．以下のプログラムでは，速度と位置の更新の際に，速度や位置が定義域を超えないように制限を加えている．

```
1  # 粒子の速度ベクトルの更新
2  def updateVelocity(R, V, Np, Nd, j, w, vMin, vMax, pBestPos, gBestPos):
3      for p in range(0, Np):
4          for i in range(0, Nd):
5              # ランダムな値r1, r2 を生成
6              r1 = random()
7              r2 = random()
8              # 速度ベクトルの更新
9              V[p][i] = w * V[p][i] + r1 * c1 * (pBestPos[p][i] - R[p][i])
10                                   + r2 * c2 * (gBestPos[i] - R[p][i])
11             # 速度制限
12             if V[p][i] > vMax: V[p][i] = vMax
13             if V[p][i] < vMin: V[p][i] = vMin
14
15 # 粒子の位置ベクトルの更新
16 def updatePosition(R, Np, Nd, xMin, xMax):
17     for p in range(0, Np):
18         for i in range(0, Nd):
19             R[p][i] = R[p][i] + V[p][i]
20             # 定義域外の場合には強制的に修正
21             if R[p][i] > xMax: R[p][i] = xMax
```

```
            if R[p][i] < xMin: R[p][i] = xMin
```

粒子の評価を行うため，各粒子の目的関数の値を計算し，個別の最良解（pBest）と群れ全体の最良解（gBest）を更新する．目的関数の評価は，fitFunc1() を用いて行う．

```
# 粒子の評価値の更新
def updateFitness(R, M, Np, pBestPos, pBestVal, gBestPos, gBestValue):
    for p in range(0, Np):
        # 目的関数の評価
        M[p] = fitFunc1(R[p])
        # gBest の更新
        if M[p] < gBestValue:
            gBestValue = M[p]
            gBestPos   = R[p]
        # pBest の更新
        if M[p] < pBestVal[p]:
            pBestVal[p] = M[p]
            pBestPos[p] = R[p]
    return gBestValue
```

最後に，PSO の制御パラメータの設定，メインプログラムの実装例を以下に示す．以下のコードでは，粒子数，次元数，世代数，慣性項，そして各係数を設定し，最適化を実行している．

```
if __name__ == "__main__":
    Np, Nd, Nt = 20, 20, 1000         # 粒子数、次元数、世代数
    c1, c2     = 2.05, 2.05           # 係数
    w, wMin, wMax = 0.0, 0.4, 0.9     # 慣性項の設定
    xMin, xMax = -5.12, 5.12          # 設計変数の定義域
    vMin, vMax = 0.25*xMin, 0.25*xMax  # 速度ベクトルの制限
    gBestValue = float("inf")         # gBest の初期評価値
    pBestValue = [float("inf")] * Np  # pBest の初期評価値
    pBestPos   = [[0]*Nd] * Np        # pBest の位置ベクトル
```

```
    gBestPos   = [0] * Nd            # gBest の位置ベクトル
    history    = []                  # gBest の履歴

    # 初期化
    R = initPosition(Np, Nd, xMin, xMax)
    V = initVelocity(Np, Nd, vMin, vMax)
    M = [fitFunc1(R[p]) for p in range(0, Np)]  # 目的関数の評価

    for j in range(0, Nt):
        # 粒子の位置更新
        updatePosition(R, Np, Nd, xMin, xMax)
        # gBest の評価値を更新
        gBestValue = updateFitness(R, M, Np, pBestPos, pBestValue, gBestPos, gBestValue)
        # 履歴に記録
        history.append(gBestValue)
        # 速度の更新
        updateVelocity(R, V, Np, Nd, j, w, vMin, vMax, pBestPos, gBestPos)
```

このコードを実行すると，Rastrigin 関数の最適解に収束していく過程がグラフで表示される．最良解（gBest）が世代ごとに更新される様子が履歴に記録され，最適化の進行具合を確認することができる．

3 数値実験

3.1 課題 1

最適化アルゴリズムの評価には，さまざまなテスト関数が利用されている．**Rastrigin 関数**以外の単一目的関数を調査し，それらの特徴を整理して Python で実装しなさい．

文献 [4] などを参考にし，最適化アルゴリズムのベンチマーク関数としてよく使用されるテスト関数について調査する．なお，テスト関数の性質としては以下のような要素がある．

- **単峰性関数（Unimodal functions）**：最適解がただ一つの谷に存在する関数
- **多峰性関数（Multimodal functions）**：最適解が複数の谷に存在する関数

- **変数間の依存関係の有無（Variable dependencies）**：変数が独立しているか，または相互に依存しているか

具体的なテスト関数として，次に示す 3 つの関数を紹介する．これらの関数は最適化問題において広く利用され，それぞれ異なる性質を持っている．

- **Sphere 関数**：この関数は，すべての変数が独立しており，最適解は原点（全ての変数が 0）にある．

$$f_{\text{Sphere}}(x_i) = \sum_{i=1}^{D} x_i^2, \quad (-10 \leq x_i \leq 10) \tag{6}$$

- **Rosenbrock 関数（Rosenbrock-Star）**：この関数は，変数間に依存関係がある．

$$f_{\text{Rosenbrock-Star}}(x_i) = \sum_{i=1}^{D-1} \left\{ 100(x_1 - x_i)^2 + (x_i - 1.0)^2 \right\}, \quad (-10 \leq x_i \leq 10) \tag{7}$$

- **Griewank 関数**：この関数は，複数の変数が相互に影響し合う形式で，非常に多くの局所最小値を持っている．

$$f_{\text{Griewank}}(x_i) = \frac{1}{4000} \sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1, \quad (-500 \leq x_i \leq 500) \tag{8}$$

各関数の特徴を理解し，それぞれ Python で実装する．そして，実装したテスト関数を利用し，PSO で最適化を行いなさい．これらの関数の実装例を以下に示す．

```python
# Sphere 関数
def fitFuncSphere(xVals):
    return sum([x**2 for x in xVals])

# Rosenbrock 関数 (Rosenbrock-Star)
def fitFuncRosenbrock(xVals):
    D = len(xVals)
    return sum([100 * (xVals[i] - xVals[i-1])**2 + (xVals[i] - 1.0)**2 for i in range(1, D)])

# Griewank 関数
def fitFuncGriewank(xVals):
    sum_term = sum([x**2 for x in xVals]) / 4000
    prod_term = 1
```

```
14      for i in range(len(xVals)):
15          prod_term *= math.cos(xVals[i] / math.sqrt(i+1))
16      return sum_term - prod_term + 1
```

実装したテスト関数を使用し，最適解に収束する過程を観察し，アルゴリズムの挙動や収束速度を比較しなさい．

3.2 課題 2

PSO（粒子群最適化）の標準的なパラメータが解探索に与える影響を調べ，数値実験を行いその結果を考察しなさい．

- **粒子数（Np）**：最適化を行う粒子の数
- **次元数（Nd）**：探索する解空間の次元数
- **繰り返し回数（Nt）**：アルゴリズムの更新を行う回数
- **更新式の係数（c1, c2）**：粒子の自己認識と社会的認識に関する重み
- **慣性項（w）**：粒子の速度を制御する係数

課題 2 では，テスト関数として **Rastrigin 関数**を使用し，PSO のパラメータを以下のように設定する．

- `Np=20`
- `Nd=20`
- `Nt=1000`
- `c1, c2=2.05`
- `w=0.75`
- `[xMax, xMin]=[-500, 500]`

実験結果は，10 回実行した際の `gBestVal`（最良適応度）の値をプロットする．実験結果の示し方については，**Fig 2** を参考にすること．

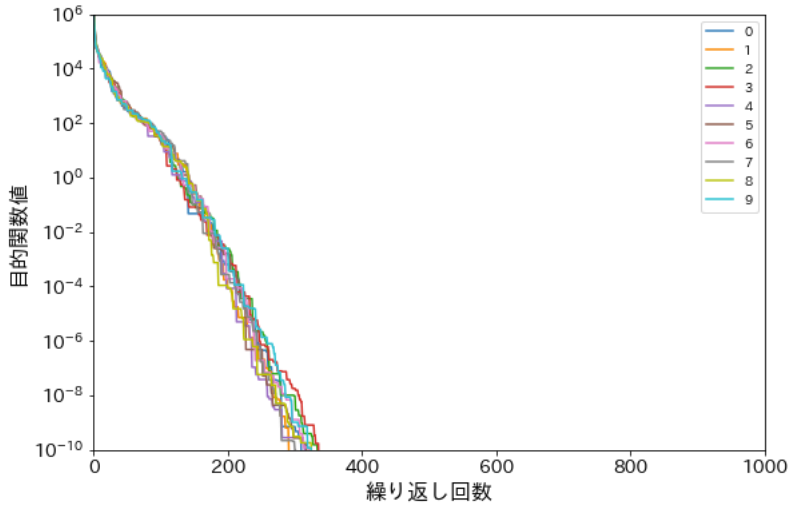

Fig 2　数値実験結果の一例 (Np=20, Nd=20, Nt=1000, c1, c2=2.05, w=0.75, [xMax, xMin]=[-500, 500])

Fig 2 のようなグラフを作成するには，サンプルコードの「if __name__ == "__main__":」部分を以下のように修正する．この修正では，追加のライブラリ pandas，numpy，およびグラフ中で日本語を使用するために japanize_matplotlib をインストールする必要がある．

```
if __name__ == "__main__":
    Np, Nd, Nt    = 20, 20, 1000      # 粒子数,次元数,世代数
    c1, c2        = 2.05, 2.05        # 係数
    w             = 0.75              # 慣性項
    xMin, xMax    = -500, 500         # 設計変数の定義域の最大・最小値
    vMin, vMax    = 0.25*xMin, 0.25*xMax # 速度ベクトルの最大・最小値

    # 試行回数 ITR と最良値を格納するリスト history
    ITR = 10
    history = np.empty((ITR, Nt))

    # 設定した試行回数繰り返す
    for i in range(0, ITR):
        gBestValue = float("inf")         # gBest（評価値）
        pBestValue = [float("inf")] * Np  # pBest（評価値）
        pBestPos   = [[0]*Nd] * Np        # pBest の位置ベクトル
```

```
        gBestPos    = [0] * Nd              # gBest の位置ベクトル
        R = initPosition(Np, Nd, xMin, xMax)
        V = initVelocity(Np, Nd, vMin, vMax)
        M = [fitFunc1(R[p]) for p in range(0, Np)]  # 目的関数
        for j in range(0, Nt):
            updatePosition(R, Np, Nd, xMin, xMax)
            gBestValue = updateFitness(R, M, Np, pBestPos, pBestValue, gBestPos, gBestValue)
            history[i][j] = gBestValue
            updateVelocity(R, V, Np, Nd, j, w, vMin, vMax, pBestPos, gBestPos)

    # グラフの描画 ライブラリの読み込みが必要
    df = pd.DataFrame(history).T              # pandas を利用しデータの変換 T は転置
    df.plot(logy=True,                        # プロット y 軸を対数スケールに変更
            xlim=[0,1000],                    # x 軸の範囲を指定
            ylim=[0.0000000001,1000000],      # y 軸の範囲を指定
            fontsize=14,                      # フォントサイズ
            figsize=(9, 6))                   # グラフの大きさ
    plt.xlabel(xlabel='繰り返し回数', size=16) # x 軸のラベル
    plt.ylabel(ylabel='目的関数値', size=16)   # y 軸のラベル
```

japanize_matplotlib を利用するには，以下のコマンドを使ってライブラリをインストールする．

```
!pip install japanize-matplotlib
```

また，必要な以下のライブラリを追加する．

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import japanize_matplotlib
```

3.3 課題 3

課題 1 と**課題 2** を活用し，PSO による解探索が，簡単な問題と難しい問題に対してどのように異なるかを考察しなさい．

3.4 課題 4

課題 2 で探索性能を評価した PSO アルゴリズムは，慣性項付きの標準的なアルゴリズムである．PSO にはこのアルゴリズム以外にも様々な改良手法が存在する．これらのアルゴリズムや改良手法について調査し，その特徴をまとめなさい．

何から手をつけて良いかわからない人は，代表的な改良手法である **LDIWM**（Linearly Decreasing Inertia Weight Method）について調べ，Python で実装しなさい．**LDIWM** は，式 **(1)** のパラメータ w を以下のように変更する改良手法である．

$$w^k = w^0 - \frac{(w^0 - w^{k_{\max}}) \times k}{k_{\max}} \tag{9}$$

ここで，w^k は k 回目の繰り返しにおける慣性項を表し，w^0 は初期の慣性項（$k = 0$ での値），$w^{k_{\max}}$ は最大繰り返し回数 $k_{\max}$ での慣性項である．この手法では，探索の序盤に大きめの慣性項を与え，探索の進行にともない慣性項を減少させることで，解探索の効率を向上させる．

Python での実装方法は，サンプルコードの「`if __name__ == "__main__":`」の部分を以下のように変更する（実際には 2 行の追加のみ）．ここで，`wMin` が w^0 に，`wMax` が $w^{k_{\max}}$ に対応する．多くの論文で用いられているパラメータは，`wMin=0.4`，`wMax=0.9` などである．

```python
if __name__ == "__main__":
    Np, Nd, Nt    = 20, 20, 1000        # 粒子数,次元数,世代数
    c1, c2        = 2.05, 2.05          # 係数
    w             = 0.75                # 慣性項
    wMin, wMax = 0.4, 0.9               # この行を追加 LDIWM の場合の慣性項
    xMin, xMax = -5.12, 5.12            # 設計変数の定義域の最大・最小値
    vMin, vMax = 0.25*xMin, 0.25*xMax   # 速度ベクトルの最大・最小値
    gBestValue = float("inf")           # gBest（評価値）
    pBestValue = [float("inf")] * Np    # pBest（評価値）
    pBestPos   = [[0]*Nd] * Np          # pBest の位置ベクトル
    gBestPos   = [0] * Nd               # gBest の位置ベクトル
    history    = []                     # gBest の履歴
```

```
    R = initPosition(Np, Nd, xMin, xMax)
    V = initVelocity(Np, Nd, vMin, vMax)
    M = [fitFunc1(R[p]) for p in range(0, Np)]  # 目的関数
    for j in range(0, Nt):
        updatePosition(R, Np, Nd, xMin, xMax)
        gBestValue = updateFitness(R, M, Np, pBestPos, pBestValue, gBestPos, gBestValue)
        history.append(gBestValue)
        w = wMax - ((wMax - wMin) / Nt) * j  # この行を追加
        updateVelocity(R, V, Np, Nd, j, w, vMin, vMax, pBestPos, gBestPos)
```

3.5 課題 5

課題 4 で実装した改良アルゴリズムを用いて数値実験を行い，その結果を**課題 2** で得られた結果と比較しなさい．比較の際には，それぞれのアルゴリズムの性能や探索効率について考察を行い，改良の効果を評価しなさい．

3.6 課題 6

Fig 3 に示すように，半球状のキャップが両橋に付いている円筒状の容器において，材料，形成，溶接に必要なコストを最小化する圧力容器の最適設計問題 [5, 6] に PSO を応用する．シェルの厚み T_s，キャップの厚み T_h，内径 R，円筒状の長さ L の 4 つの変数を設計する．以下に示す定式化では $\{T_s, T_h, R, L\} = \{x_1, x_2, x_3, x_4\} = \boldsymbol{x}$ としている．なお，内径と円筒状の長さは連続値だが，シェルの厚みとキャップの厚みは ASME 規格により 0.0625[inch] 間隔の離散値となる．

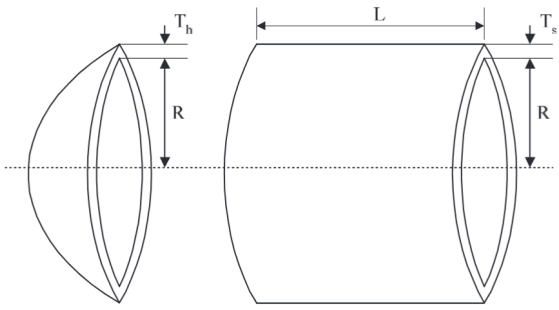

Fig 3 圧力容器設計問題 (図引用 [6])

$$\text{最小化} \quad f(\boldsymbol{x}) = 0.6224x_1x_3x_4 + 1.7781x_2x_3^2 + 3.1661x_1^2x_4 + 19.84x_1^2x_3 \tag{10}$$

$$\text{制約条件} \quad g_1(\boldsymbol{x}) = -x_1 + 0.0193x_3 \leq 0, \tag{11}$$

$$g_2(\boldsymbol{x}) = -x_2 + 0.00954x_3 \leq 0, \tag{12}$$

$$g_3(\boldsymbol{x}) = -\pi x_3^2x_4 - 4\pi/3x_3^3 + 1296000 \leq 0, \tag{13}$$

$$g_4(\boldsymbol{x}) = x_4 - 240 \leq 0, \tag{14}$$

$$\{x_1, x_2\} = 0.0625i, i \in \{1, 2, \cdots, 99\}, 10 \leq \{x_3, x_4\} \leq 200.$$

制約違反を許容しないために，ペナルティ関数を定義する．ペナルティ関数の一例を以下に示す．

$$P(\boldsymbol{x}) = f(\boldsymbol{x}) + \text{penalty\_factor} \times \sum_i \max(0, g_i(\boldsymbol{x})) \tag{15}$$

以下に示す実装例では，ペナルティ係数 penalty_factor を 10^6 としている．制約 $g_i(\boldsymbol{x})$ が正のとき (すなわち制約違反しているとき) だけペナルティが加算される．

```python
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import japanize_matplotlib

# 目的関数と制約条件
def objective_function(x):
    x1, x2, x3, x4 = x
    return (
        0.6224 * x1 * x3 * x4 +
        1.7781 * x2 * x3**2 +
        3.1661 * x1**2 * x4 +
        19.84 * x1**2 * x3
    )

def constraints(x):
    x1, x2, x3, x4 = x
```

```
    g1 = -x1 + 0.0193 * x3
    g2 = -x2 + 0.00954 * x3
    g3 = -np.pi * x3**2 * x4 - (4/3) * np.pi * x3**3 + 1296000
    g4 = x4 - 240
    return np.array([g1, g2, g3, g4])

# ペナルティ関数( 例)
def penalty_function(x):
    penalty_factor = 1e6
    constraint_values = constraints(x)
    penalty = np.sum(np.maximum(0, constraint_values)) * penalty_factor
    return objective_function(x) + penalty

# PSO アルゴリズム
def pso(objective, bounds, discrete_indices, num_particles=30, max_iter=100, w=0.5, c1=2, c2
    =2):
    num_dimensions = len(bounds)
    positions = np.array([
        [np.random.uniform(low, high) for low, high in bounds]
        for _ in range(num_particles)
    ])

    for i in discrete_indices:
        positions[:, i] = np.round(positions[:, i] / 0.0625) * 0.0625

    velocities = np.zeros((num_particles, num_dimensions))
    personal_best_positions = positions.copy()
    personal_best_values = np.array([objective(pos) for pos in positions])
    global_best_position = positions[np.argmin(personal_best_values)]
```

```
    global_best_value = np.min(personal_best_values)

    best_values = []
    iteration_best_values = []

    for iteration in range(max_iter):
        iteration_best_values.append(global_best_value)  # 世代ごとの最適化結果を記録

        for i in range(num_particles):
            r1 = np.random.rand(num_dimensions)
            r2 = np.random.rand(num_dimensions)
            velocities[i] = (
                w * velocities[i]
                + c1 * r1 * (personal_best_positions[i] - positions[i])
                + c2 * r2 * (global_best_position - positions[i])
            )
            positions[i] += velocities[i]
            for j, (low, high) in enumerate(bounds):
                positions[i, j] = np.clip(positions[i, j], low, high)

            for j in discrete_indices:
                positions[i, j] = round(positions[i, j] / 0.0625) * 0.0625

            current_value = objective(positions[i])

            if current_value < personal_best_values[i]:
                personal_best_positions[i] = positions[i].copy()
                personal_best_values[i] = current_value

```

```
        global_best_position = personal_best_positions[np.argmin(personal_best_values)]
        global_best_value = np.min(personal_best_values)

    return global_best_position, global_best_value, iteration_best_values

# 問題設定
bounds = [(0.0625, 6.1875), (0.0625, 6.1875), (10, 200), (10, 200)]
discrete_indices = [0, 1]

# 試行回数
num_trials = 10
all_iteration_values = []
best_positions = []
best_values = []

# 10試行
for trial in range(num_trials):
    best_position, best_value, iteration_best_values = pso(penalty_function, bounds,
    discrete_indices)
    all_iteration_values.append(iteration_best_values)
    best_positions.append(best_position)
    best_values.append(best_value)

# 世代ごとの最適化結果をプロット
plt.figure(figsize=(8, 6))

for i in range(num_trials):
    # 各試行に異なる色と線のスタイルを指定
    plt.plot(range(1, len(all_iteration_values[i]) + 1),
```

```
103             all_iteration_values[i],
104             label=f"試行 {i+1}",
105             linestyle='-')
106
107 plt.xlabel("世代（繰り返し回数）", fontsize=14)
108 plt.ylabel("目的関数値", fontsize=14)
109 plt.grid(True)
110 plt.legend(fontsize=12)
111 plt.tight_layout()  # レイアウト調整
112 plt.show()
113
114 # 結果をpandas で表示
115 trial_results = {
116     "試行": [],
117     "T_s": [],
118     "T_h": [],
119     "R": [],
120     "L": [],
121     "目的関数値": [],
122 }
123
124 for i in range(num_trials):
125     trial_results["試行"].append(i + 1)
126     trial_results["T_s"].append(best_positions[i][0])
127     trial_results["T_h"].append(best_positions[i][1])
128     trial_results["R"].append(best_positions[i][2])
129     trial_results["L"].append(best_positions[i][3])
130     trial_results["目的関数値"].append(best_values[i])
131
```

```
# pandas DataFrame に変換
df_results = pd.DataFrame(trial_results)

# 表示
pd.set_option('display.float_format', '{:.4f}'.format)  # 小数点以下 4桁に設定
print(df_results)
```

3.7 課題 7

課題 6 の「圧力容器設計問題」以外の設計問題を調べレポート (報告書) に 1 つ示しなさい．なお，余力のある人は，その問題を PSO を用いて最適化するコードを実装し示しなさい．

出典・参考文献

[1] 相吉英太郎, 安田恵一郎編著, “メタヒューリスティクスと応用”, 電気学会, 2007.

[2] J. Kennedy and R. Eberhart, “Particle Swarm Optimization”, Proceedings of the 1995 IEEE International Conference on Neural Networks, pp. 1942–1948, 1995.

[3] 茨木俊秀, 福島雅夫, “最適化の手法”, 共立出版, 1993.

[4] Axel Thevenot, “Optimization & Eye Pleasure : 78 Benchmark Test Functions for Single Objective Optimization”, https://github.com/AxelThevenot/Python_Benchmark_Test_Optimization_Function_Single_Objective

[5] B. K. Kannan, S. N. Kramer, “An Augmented Lagrange Multiplier Based Method for Mixed Integer Discrete Continuous Optimization and Its Applications to Mechanical Design”, ASME. Journal of Mechanical Design, Vol. 116, No. 2, pp. 405–411, 1994.

[6] 阪井節子, 高濱徹行, “集団的降下法に対するペナルティ係数の適応的調整法の提案”, 数理解析研究所講究録, Vol. 2126, pp. 53–62, 2019.

[7] 若佐裕治, “粒子群最適化の安定性と性能の解析”, システム/制御/情報, Vol. 57, No. 5, pp. 201–206, 2013.