卒業研究報告書

題目

並列差分進化法を用いた画像処理

指導教員

石水 隆　講師

報告者

11–1–037–0057

渋谷 俊樹

近畿大学理工学部情報学科

平成 24 年 1 月 31 日提出

概要

近年, 市販のデスクトップ PC やノート PC にもマルチコア・プロセッサが用いられている. しかし, 複数のコアによって並列処理を実行するためには, マルチコア CPU に対応した並列プログラムを作成する必要がある.[3]. 並列プログラムは複数のスレッドと呼ばれる処理単位から構成され, 異なるコアに割り当てられたスレッドは同時に実行される. 既存の逐次処理のプログラムを機械的に並列プログラムに変換するコンパイラも開発されているが, より効率的な並列プログラムを作成するには, アルゴリズムの段階から処理の並列化を図る必要がある.

並列プログラムによる高速化が求められる分野には様々あるが, その中の一つに最適化問題がある. 今日, 様々な種類の最適化問題に対してその問題の性質に応じた解放が考案されている. しかしながら, 制約条件が多い場合には問題の定式化が困難となる.

そこで, これらの問題を解決する手法として, 生物進化をヒントに得た探索手法である進化計算を用いることが近年注目されている. 遺伝的アルゴリズム (GA : Genetic Algorithm) に代表される進化計算の大きな特徴は, 解空間が非常に大きく, 従来の手法では高速計算機を用いても数十年かかるような問題や, 問題の構造が分からず数学的に定式化できないため解けないような問題に対しても, 近似解を比較的高速に得られるという点がある.

差分進化 (DE : Differential Evolution) とは, 決定変数が実数値を取る単目的の関数最適化問題を対象としたとした進化計算の一種である.[4] DE は微分不可能な多峰性関数の最適化問題にも有効に適用することができ, 多くのベンチマーク問題や現実的な最適化問題において, 優秀な最適化アルゴリズムの 1 つであることが示されている. また,DE はアルゴリズムが単純であるため実装が容易である. すなわち,DE の最大の利点は, 強力かつシンプルであるため, 使い勝手が良いことである.

このように進化計算, 取り分け差分進化計算は有効性が極めて高いため, 並列化により高速化することは重要である. しかし, 従来の進化計算の並列化手法は, そのプログラムを実行するハードウェアに大きく依存する. そこで本研究では, ハードウェアへの依存性が低い並列化手法を検討する.

本研究では, 上記の DE の並列化である, 並列差分進化計算 (PDE : Parallel Differential Evolution) を用いて画像処理を行う. 本研究で対象とする画像処理は, ぼやけた画像が与えられた際に, その元となる鮮明な画像を得るノイズ除去処理である. この処理を実装することによって, 画像の類似性を視覚的に認識することにより, 最適化問題における DE の有効性を考察する.

目次

1 序論

1.1 本研究の背景

近年, 市販のデスクトップ PC やノート PC にもマルチコア・プロセッサが用いられている．しかし, 複数のコアによって並列処理を実行するためには, マルチコア CPU に対応した並列プログラムを作成する必要がある.[3]. 並列プログラムは複数のスレッドと呼ばれる処理単位から構成され, 異なるコアに割り当てられたスレッドは同時に実行される．既存の逐次処理のプログラムを機械的に並列プログラムに変換するコンパイラも開発されているが, より効率的な並列プログラムを作成するには, アルゴリズムの段階から処理の並列化を図る必要がある．

並列プログラムによる高速化が求められる分野には様々あるが, その中の一つに最適化問題がある．今日, 様々な種類の最適化問題に対してその問題の性質に応じた解放が考案されている．

最適化問題に対する手法には, オペレーションズ・リサーチや数理的計画法の他に, 人工知能, エキスパート・システム, システム理論, ファジィ集合, ニュートラルネットワーク等, 様々なアルゴリズムが存在している．一般に, ある最適化問題を解くにはその問題の背景に十分注意を払って, 制約条件や目的関数の評価基準を定め, 数式もしくはグラフ等のモデルによって最適化問題を単純化する．そして、単純化された問題に対してあアルゴリズムを適用し, 主にコンピュータ上でシミュレーションを行うことによって最適化問題の解を得ることができる．

しかしながら，制約条件が多い場合には問題の定式化が困難となる．そこで，これらの問題を解決する手法として，生物進化をヒントに得た探索手法である進化計算を用いることが近年注目されている．遺伝的アルゴリズム (GA : Genetic Algorithm) に代表される進化計算の大きな特徴は，解空間が非常に大きく，従来の手法では高速計算機を用いても数十年かかるような問題や, 問題の構造が分からず数学的に定式化できないため解けないような問題に対しても，近似解を比較的高速に得られるという点がある．

差分進化 (DE : Differential Evolution) とは, 決定変数が実数値を取る単目的の関数最適化問題を対象としたとした進化計算の一種である [4]．DE は微分不可能な多峰性関数の最適化問題にも有効に適用することができ，多くのベンチマーク問題や現実的な最適化問題において, 優秀な最適化アルゴリズムの 1 つであることが示されている．また,DE はアルゴリズムが単純であるため実装が容易である．すなわち,DE の最大の利点は, 強力かつシンプルであるため, 使い勝手が良いことである．

1.2 本研究の目的

前節で述べたように，ように進化計算，取り分け差分進化計算は有用性が極めて高いため，並列化により高速化することは重要である．しかし，従来の従来の進化計算の並列化手法は, そのプログラムを実行するハードウェアに大きく依存する．そこで本研究では，ハードウェアへの依存性が低い並列化手法を検討する．

本稿では画像の類似性を視覚的に認識することで, 最適化問題における DE の有効性を検証する．また, 候補個体を得る戦略をいくつか用意した上で, どの戦略が最も優良であるかを検証するのも本研究の目的である．

1.3 本報告書の構成

以下に各章の簡単な構成を示す．

まず第 2 章では，本研究が対象とする最適化問題について説明し，続く第 3 章では，進化計算について説明する．第 4 章では本研究で対象とする画像処理について述べる．第 5 章で実験結果を示し，第 6 章で考察を行う．最後に第 7 章で結論および今後の課題を示す．

2 最適化問題

最適化問題は, 自然科学, 工学, 社会科学など様々な分野で発生する基本的な問題の 1 つである. 近年では, 最適化問題の大規模化と複雑化に伴い厳密な最適解を求めるのが難しくなっている. 必要十分な最適性を持つ近似解を実時間内に求める必要があり,GA や DE などメタヒューリスティックな最適化手法への関心が高まっている.[5] 本研究では, 最適化手法である DE と PDE を用いる. そのため, 本章では, 最適化問題の概要について記述する.

2.1 最適化問題概要

最適化とは複数の選択肢から最善のものを選ぶことであり, 最適化問題を数学的に表現すると, 与えられた制約条件を全て満たし, 目的関数 f(x) の値が最小または最大になるような決定変数 x の値を見つける問題である. この場合 x は最適解と呼ばれ, 最適解は 1 つしか存在しない場合もあるが, 反対に無限個存在する場合もある. 一般的に最適化問題は式 (1)(2) に記述する.

$$\text{目的関数 } f(x) \rightarrow \text{最小値 (最大値)} \tag{1}$$

$$\text{制約条件 } x \in F \tag{2}$$

このとき F は実行可能集合あるいは実行可能領域と呼ばれており, 制約条件を満たす実行可能解 (数理計画問題において, 実行可能領域上の点. 最適解とは限らない) の集合である.

2.1.1 最適解と局所的最適解

式 (3) の条件を満たす実行可能解 $x^n \in F$ を最適解と呼ぶ

$$f(x^n) \leq f(x), \qquad x \in F \tag{3}$$

また実行可能 $x^n \in F$ を含む適当な集合 $\mathrm{U}(x^n)$ に対しては,

$$f(x^n) \leq f(x), \qquad x \in F \cap U(x^n) \tag{4}$$

が成り立つとき、x^n を局所的最適解という. 図**??**は最適解と局所的最適解を表している. $\mathrm{U}(x^n)$ は一般的には近傍と呼ばれ,x に少し変形を加えることによって得られる解集合のことである. なお,$U(x^n) \in F$ とは限らない. 最適化問題も目的関数や制約条件が複雑になり, 実行可能集合が膨大になる場合には最適解をみつけることは一般的に困難である. そのような場合には, 問題の大きさや問題を解くために与えられている時間等を考慮して, 適切な近傍を定義して局所的最適解を求める.

3 進化計算

3.1 差分進化計算 (DE)

DE は K.price, R.Storn らによって提案された, 確率的な探索法であり, 解集団を用いた多点探索をおこなう．DE の重要な特徴としては, 単純な数学的演算を用いることが挙げられる．このため, 制御パラメタの数が少なく, 設定が容易であり問題への実装も比較的容易におこなえる．DE において新たな解候補個体を得る戦略は, 基底個体の選び方および交叉の種類の組み合わせで定義され, [DE/best/1/bin] や [DE/rand/1/exp] などが代表される [2]．基底個体としては, 最適な関数値を持つ個体を選ぶ最適基底 (best) と, ランダムに選ぶランダム基底 (rand) の 2 通りが主に選択される．また, 代表的な交叉としては, 二項交叉 (bin) と指数交叉 (exp) の 2 通りが使用される．

3.1.1 DE の基本アルゴリズム

標準的な DE の処理手順を以下に示す．

Step1： N_P 個の個体を, 各次元の定義域においてランダムに生成して世代 g=1 とする．また, 最終世代, 収束の条件に設定する．

Step2： 各個体の関数値を計算する．

Step3： 各個体 $\mathbf{x}_i$(i = 1,2,...N_P) に対して以下の処理を行う．

Steo3.1： 最適個体またはランダム個体として個体 $\mathbf{x}_j(j \neq i)$ を選択する．

Step3.2： $\mathbf{x}_j$ から差分変異親個体 $\mathbf{v}_i$ を生成する．

Step3.3： 後述する交叉法により、対象親個体 $\mathbf{x}_i$ と差分変異親個体 $\mathbf{v}_i$ から子個体 $\mathbf{u}_i$ を生成する．

Step3.4： 対象親個体 $\mathbf{x}_i$ と子個体 $\mathbf{u}_i$ の関数値を比較し，良い方を次世代の $\mathbf{x}_i$ として残す．

Step6： 終了条件を満たしていなければ,g=g+1 として Step3 に戻る．

3.1.2 交叉

本節では、交叉について述べる．交叉とは、2 つの親個体から新たな子個体を生成する手続である．

DE では、まず対象親個体 $\mathbf{x}_i(1 \leq i \leq N_P)$ に対して個体 $\mathbf{x}_j(j \neq i)$ を選択する．$\mathbf{x}_j$ は、戦略 best では最も優れた個体，戦略 rand ではランダムに選んだ個体である．次に個体 $\mathbf{x}_j$ から差分変異親個体 $\mathbf{v}_i$ を生成する．$\mathbf{v}_i$ は以下の式で与えられる．ただし，$\mathbf{x}_k, \mathbf{x}_l$ はランダムに選択した個体，S_F は差分定数である．

$$\mathbf{v}_i := \mathbf{x}_j + S_F(\mathbf{x}_k - \mathbf{x}, l)$$

本研究では簡単のために差分定数 $S_F = 0$ としている．すなわち，$\mathbf{v}_i = \mathbf{x}_j$ である．

交叉により差分変異個体 $\mathbf{v}_i$ と対象親個体 $\mathbf{x}_i$ から子個体 $\mathbf{u}_i$ を生成する．二項交叉および指数交叉による生成は以下の手続き (5) および (6) で計算される．ただし，$v_{j,d}, x_{i,d}, u_{i,d}(1 \leq m \leq D)$ はそれぞれ $\mathbf{v}_i, \mathbf{x}_i, \mathbf{u}_i$ の d 番目の要素, d_r は $1 \leq d_r \leq D$ のランダムな整数であり，rand[0,1] は範囲 [0,1] の一様乱数である．

二項交叉

$$\left[\begin{array}{l} \text{for}(d := 1; j \leq D; ++d)\{ \\ \quad \text{if (rand } [0,1] < C_R \vee d = d_r) \\ \quad\quad u_{i,d} := v_{i,d}; \\ \quad \text{else } u_{i,d} := x_{i,d}; \\ \} \end{array}\right. \tag{5}$$

指数交叉

$$\left[\begin{array}{l} d := d_r; \\ \text{do } \{ \\ \quad u_{i,d} := v_i; \\ \quad d := (d+1)\%D; \\ \} \text{ while (rand } [0,1] < C_R \wedge d \neq d_r) \\ \text{while } (d \neq d_r)\{ \\ \quad u_{i,d} := x_{i,d}; \\ \quad d := (d+1)\%D; \\ \} \end{array}\right. \tag{6}$$

3.2 並列差分進化計算 (PDE)

並列差分進化計算 (PDE) は DE のプロセッサネットワーク上での並列化である. PDE では，各プロセッサが DE を並列に実行し，数世代に 1 回の割合で最も優れた個体を隣のプロセッサに送信する．また, 移民頻度と最適解への収束速度および最適解発見率は相関関係があり, 移民頻度を高くすると収束速度は速くなるが, 最適解発見率は低下する.[1]

3.2.1 ネットワーク

PDE で用いたネットワーク構造について記述する.

(1) リング型

複数のコンピュータあるいは接続機器を 1 本の環状のケーブルに接続する LAN 方式のことである.

(2) トーラス型

階層構造をもったネットワークである. 入力層, 中間層, 出力層の 3 つに分かれており, 入力層から入った情報は中間層, 出力層を通って出力される.

(3) ハイパーキューブ型

ハイパーキューブは,n-キューブ、あるいは超立方体とも呼ばれ, 汎用の (超) 並列計算の結合方法として注目されており, 実際にいくつかの並列システムに採用されている．ハイパーキューブはトーラス等のトポロジに比べて, プロセッサ数に対する直径を小さくできる.

図 1,2,3 にそれぞれリング型，トーラス型，ハイパーキューブ型のネットワークの概念図を示す.

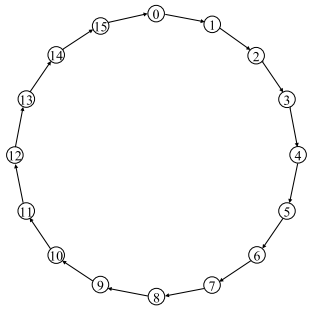

図 1　リング型

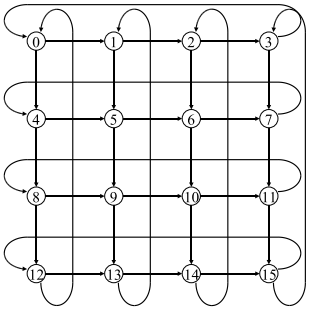

図 2　トーラス型

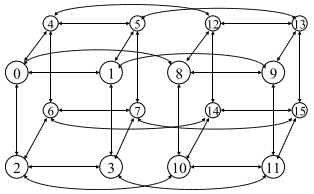

図 3　ハイパーキューブ型

3.2.2　PDE の戦略

本研究では, 基底個体の選び方 (mix は混在,rand はランダム,best は最適値) および交叉の種類 (mix は混在,bin は二項交叉,exp は指数交叉) の組み合わせた, 5 つの戦略で実験をおこなう. 本章では, その 5 つの戦略を記述する.

- 1 つ目の戦略 : [best/exp]
- 2 つ目の戦略 : [rand/exp]
- 3 つ目の戦略 : [best/bin]
- 4 つ目の戦略 : [rand/bin]
- 5 つ目の戦略 : [mix/mix]

3.2.3　PDE の基本アルゴリズム

PDE は, 局所メモリを持つ P 台のプロセッサ P_p $(0 \leq p < P)$ が, リング状に接続されたプロセッサネットワークを仮定する. PDE では, プロセッサ P_p $(0 \leq p < P)$ がそれぞれ集団 x_i^p $(0 \leq i < N)$ を保持する. PDE は以下の 2 つの操作を繰り返す. ここで G_L は移民頻度を表すパラメタである.

(1) **局所 DE 操作**　各プロセッサ $(0 \leq p < P)$ は保持する集団 x_i^p $(0 \leq i < N)$ に対して DE を G_L 世代計算する.

(2) **移民操作**　各プロセッサ $(0 \leq p < P)$ は保持する集団の中で最良の個体 x_{best}^p) を右隣のプロセッサ $P_{(p+1)modP}$ に送信し, 左隣のプロセッサ $P_{(p-1)modP}$ から受信した個体 $x_{best}^{(}(p-1)modP)$ を保持する集団 $x_i^{(}p)$ $(0 \leq i < N)$ の中の任意の個体と入れ替える.

4　画像処理

本研究では PDE を用いて画像処理を行う。本研究で対象とする画像処理は，ぼやけた画像が与えられたときに、その元となる鮮明な画像を得るノイズ除去処理である.

解となる画像については，以下の制限を設ける.

1. 元画像は 16 × 16 の 2 値画像 (1 または 0) とし，ぼやけた画像は，16 × 16 の実数値画像 (0〜1) と

する．

2. 処理中，解画像，およびそのぼやけた画像は，直接参照することはできず，ある画像がぼやけた画像にどの程度近いかを表す評価値のみを参照できる．

2. の制約のため，プログラムは直接画像を参照できず，候補画像を生成する際に，解画像およびそのぼやけた画像から新たな画像を生成することができない．この制約は，例えばモンタージュ画像を作るときのような状況を想定している．解となる画像は目撃者の頭の中しか無いため，モンタージュ画像作成者は元画像を見ることができない．よってモンタージュ画像作成者は，適当な絵を目撃者に見せて，それが解画像とどの程度近いかの評価してもらい，その評価値を元に新たな画像を作成することを繰り替えずことで解画像に近づけてゆく．

4.1 画像処理の手順

本研究で作成した画像処理の手順を以下に述べる．DE の部分については以下の手順で行う．

手順 1　候補画像を 20 個ほど生成する. 候補画像は白黒でランダムに生成する．
手順 2　候補画像同士を交叉させて新たな画像生成する．
手順 3　手順 2 の画像が親画像よりも解画像に近ければ親画像と入れ替える．
手順 4　手順 2 の画像が充分解画像に近ければ終了, それ以外は手順 2 に戻る．

PDE の部分については以下の手順で行う．

手順 1　候補画像を生成する．
手順 2　以下の処理を G_S 世代行う
手順 2.1　各プロセッサ $P_p(1 \leq p \leq Pr)$ で，DE を G_L 世代行う
手順 2.2　各プロセッサ $P_p(1 \leq p \leq Pr)$ は最も良い候補画像を隣接プロセッサ $P_q(q = p + 1 \, mod \, Pr)$ に送信する．
手順 2.3　各プロセッサ $P_p(1 \leq p \leq Pr)$ は 2.2 で受信した画像を，保持する画像の一つと置き換える．
手順 3.4　保持する画像のうち最も優れた画像が充分解画像に近ければ終了, それ以外は手順 2 に戻る．

4.2 解候補個体を得る戦略

本研究では，3.2.2 節で述べた 5 つの戦略で実験をおこなう．また，本研究では, リング型のネットワーク構造を用いて実験をおこなう．

5 実験結果

本研究では，5 つの戦略に対して, 移民頻度 8 で各 100 回の PDE を行った．表 1 にその実行結果を示す．また，付録に本研究で用いた画像処理の PDE プログラムのソースを示す．

また,5 つの戦略の中で, 最も発見率の高かった戦略 [rand/exp] で移民頻度を変えながら各 100 回の PDE を行った. 表 2 にその実行結果を示す

表 1　各戦略の解発見平均世代数と解発見率

戦略	best exp	rand exp	best bin	rand bin	mix mix
世代数	27	70	506	411	309
発見率	11%	70%	14%	16%	24%

表 2　各移民頻度での解発見平均世代数と解発見率

頻度	1	2	4	8	16
世代数	37	49	59	68	104
発見率	21%	41%	54%	76%	69%

6　考察

実験結果より, 今回用意した 5 つの戦略の中では,[rand/exp] が最も優良であることが分かった. これは,PDE を用いた画像処理は, 多峰性で変数間に依存関係がある問題といえ, 集団の多様性を確保するための処理として,「rand」が有用であったと思われる.

逆に,best では探索空間での位置によらず集団の中から淘汰する個体を選択するので, 全体としての収束性は向上するが, 今回の処理においては, 集団を局所解に陥らせたといえる.

7　結論と今後の課題

本研究では PDE を用いて, 画像処理をおこなった. 本研究において, 個体の選び方がランダム, 交叉の長さの決定方法が指数的な二点交叉である [rand/exp] が最適化問題に対し, 高確率で解を求めることが分かった. しかし, 予想していたよりも,best や mix を用いた戦略の解発見率が低かったので, 局所解に陥ってしまった場合, また, 大域世代から探索をやり直す等の改善があるように思われた.

謝辞

本論文を結ぶにあたり, 多忙にも関わらず親切な御指導, 御助言を賜りました石水隆講師に厚く御礼を申し上げます.

参考文献

[1] 石水隆, 田川聖治 : 並列差分進化計算の比較研究, 情報処理学会研究報告, Vol.2011-MPS-82 No.23 pp. 1-2 (2011)

[2] Storn, R. and Price, K. : Differential evolution - a simple and efficient heuristic for global optimization over continuous space, Journal of Global Optimization, Vol. 11, no. 4, pp. 341–359 (1997).

[3] Breshears, C.: The Art of Concurrency - A Thread Monkey's Guide to Writing Parallel Applications, O'REILLY (2009)

[4] Storn, R. and Price, K.: Differential evolution - a simple and efficient heuristic for global optimization over continuous space, Journal of Global Optimization, Vol. 11, No. 4, pp. 341—359 (1997)

[5] 伊藤 稔, 田中雅博 Particle Swarm Optimization の解探索性能に関する基礎的検討 第 32 回知能システムシンポジウム資料 (2005)

[6] 三宮信夫，喜多一，玉置久，岩本貴司，遺伝アルゴリズムと最適化，システム制御情報ライブラリー，浅草書店，(1998)

[7] 平野廣美，遺伝的アルゴリズムと遺伝的プログラミング，パーソナルメディア，(2000)

[8] 米沢保雄，遺伝的アルゴリズム 進化理論の情報学，森北出版，(1993)

[9] T.Ishimizu and K.Tagawa, Experiment Study of A Structured Differential Evolution with Mixed Strategies, Proc. the World Congress on Nature and Biologically Inspired Computing (NaBIC) , pp.598-603, (2010)

[10] Price,K.V.,Storn,R.M.andLampinen,J.A.:DifferentialEvolution—APractical Approach to Global Optimization, Springer (2005).

付録 A　付録

本研究で用いた画像処理の PDE のプログラムのソースを示す．プログラムは 3 つのクラスからなる．

.1　DifferentialEvolution クラス

```
package de;
import java.io.*;

/**
 * 与えられたぼやけた値画像の元画像を進化計算により得る2
 */
public class DifferentialEvolution {
        static final int D = 32;           // 問題の次元(dimension of problem)
        static final int NP = 32;          // 母集団の大きさ(size of population)
        static final double F = 0.5;       // 差分定数(differential constant)
        static final double CR = 0.9;      // 交叉定数(crossover constant)
        static final double MT = 0.1;      // 突然変異定数(mutation constant)
        static int LGEN = 8;         // 局所世代数(number of local generations)
        static int SGEN = 1024/LGEN; // スーパー世代数(number of super generation)
        static MonochromePicture target;   // 目標画像
        static MonochromePicture targets[]; // 目標画像の行列
        static final int pictureSize = 16; // 画像のサイズ

        static final int CT = -1;     // 交叉のタイプ
        /**
         * -1: mix 混在
         *  1: bin 二項交叉
         * oh: exp 指数交叉
         */
        static final int IST = -1;      // インデックス選択のタイプ
        /**
         * -1: mix 混在
         *  1: rand ランダム
         * oh: best 最適値
         */
        static final int NWT = 1;        // ネットワークのタイプ
        /**
         *  1: ring network
         *  2: torus network
         *  3: hypercube network
         *  4: hierarchical network
         * oh: no network
         */
        static final int NTH = 4;        // 局所計算スレッド数
        static final int ROOPN = 8;      // ループ回数
        static final int GV = 1;
        static final double EPSILON = 0.1; // 適応値がこれより小さくなると解とみなす
        static LocalDifferentialEvolution de[]; // 進化計算スレッドの実装クラスのインスタンス
        static Thread th[];                     // 進化計算スレッドのスレッドのインスタンス
```

```
static FileWriter pass, rPass;
static PrintWriter fp, rfp;
static double fitnessRecord[][];      // 適応値記録用
static MonochromePicture optimumRecord[];      // 最適値記録用
static int roopIndex = 0;
static boolean isFindOptimum;   // 最適解発見？
static int findOptimum[];         // 最適解発見時
static int sgen, lgen;

public static void main (String[] args) {
        de = new LocalDifferentialEvolution[NTH];
        th = new Thread[NTH];



        //for (LGEN=1; LGEN<1024; LGEN*=2) {
        //      SGEN=2048/LGEN;

        lgen = LGEN;
        sgen = SGEN;
        if (LGEN > GV) {
                lgen = GV;
                sgen = SGEN * (LGEN / GV);
        }

        fitnessRecord = new double[ROOPN][sgen];
        optimumRecord = new MonochromePicture[ROOPN];
        targets = new MonochromePicture[ROOPN];
        roopIndex = 0;
        findOptimum = new int[ROOPN];

        String fname = recordFileName();
        try {
                //pass  = new FileWriter ("fitness.txt");
                //fp = new PrintWriter (pass);
                rPass = new FileWriter (fname);
                rfp = new PrintWriter (rPass);
        } catch (IOException e) {
                System.err.println (e);
        }

        for (roopIndex = 0; roopIndex<ROOPN; ++roopIndex) {
                targets[roopIndex] = target = new MonochromePicture (pictureSize);

                for (int i=0; i<NTH; ++i) {
                        de[i] = new LocalDifferentialEvolution (NP, F, CR, MT, lgen, CT, IST, t
                }

                isFindOptimum = false;
                findOptimum[roopIndex] = SGEN*LGEN; // SGEN*LGEN;

                switch (NWT) {
                case 1: // リング型ネットワーク
```

```
                        ringNetwork();
                        break;
                case 2: // トーラス型ネットワーク
                        torusNetwork();
                        break;
                case 3: // ハイパーキューブ型ネットワーク
                        hypercubeNetwork();
                        break;
                case 4: // 階層型ネットワーク
                        hierarchicalNetwork();
                        break;
                default: // ネットワーク無し
                        noNetwork();
                        break;
                }
                System.out.println (roopIndex + ": find at " + findOptimum[roopIndex] + ": ");
                printOptimum();
        }

        //fp.close();
        outputFitnessRecord();
        rfp.close();
        //}
}

/**
 * 局所進化計算
 * @param sg スーパー世代数
 */
static void localDifferentialEvolution (int sg) {
        for (int i=0; i<NTH; ++i) {
                th[i] = new Thread (de[i]);
        }
        for (int i=0; i<NTH; ++i) {
                th[i].start();
        }
        try {
                for (int i=0; i<NTH; ++i) {
                        th[i].join();
                }
        } catch (InterruptedException e) {
                System.err.println (e);
        }
        //for (int i=0; i<lg; ++i) {
        //      de[i].printOptimum();
        //}
        //printFitness (sg);
        //outputFitness (sg);
}

/**
 * 最適解を得る
 * @return  MonochromePicture[] : 各プロセッサの最適解画像
```

```
 */
static MonochromePicture[] getOptimum() {
 MonochromePicture opt [] = new  MonochromePicture[NTH];
        for (int i=0; i<NTH; ++i) {
                opt[i] = de[i].getOptimum();
        }
        return opt;
}

/**
 * 適応値を得る
 * @return double[] : 各プロセッサの適応値
 */
static double[] getFitness() {
        double fit [] = new double[NTH];
        for (int i=0; i<NTH; ++i) {
                fit[i] = de[i].getFitness();
        }
        return fit;
}

static void ringNetwork() {
        MonochromePicture opt[] = new MonochromePicture[NTH];
        double fit[] = new double[NTH];

        System.out.println ("Ring network");
        //fp.println ("Ring network");
        //fp.print ("\t");
        //for (int i=0; i<NTH; ++i)
        //      fp.print (i + "\t");
        //fp.println();

        for (int g=0; g<sgen; ++g) {
                if (!isFindOptimum) { // 最適値を発見するまで局所を計算するDE
                        localDifferentialEvolution (g);
                        opt = getOptimum();
                        fit = getFitness();

                        int gv = 1;
                        if (LGEN > GV) gv = LGEN/GV;
                        if ((g % gv) == (gv-1)) {
                                for (int i=0; i<NTH; ++i) {
                                        de[(i+1) % NTH].replaceData (opt[i], fit[i]);
                                        //System.out.print (i + "->" + ((i+1) % NTH) + " ");
                                }
                        }
                }
                //System.out.println();
                recordFitness(g);
        }
}

static void torusNetwork() {
```

```
        MonochromePicture opt[] = new MonochromePicture[NTH];
        double fit[] = new double[NTH];
        int width=1, hight=1;

        while (width*hight < NTH) {
                width *= 2;
                if (width*hight == NTH) break;
                hight *= 2;
        }

        System.out.println ("Torus network");
        //fp.println ("Torus network");
        //fp.print ("\t");
        //for (int i=0; i<NTH; ++i)
        //      fp.print (i + "\t");
        //fp.println();

        for (int g=0; g<sgen; ++g) {
                if (!isFindOptimum) { // 最適値を発見するまで局所を計算するDE
                        localDifferentialEvolution (g);
                        opt = getOptimum();
                        fit = getFitness();

                        int gv = 1;
                        if (LGEN > GV) gv = LGEN/GV;
                        if ((g % gv) == (gv-1)) {
                                if ((g+1)%(2*gv) == 0) { // 横方向のスレッドと通信
                                        //System.out.println(g + "-");
                                        for (int h=0; h<hight; ++h) {
                                                for (int w=0; w<width; ++w) {
                                                        de[h*width + (w+1)%width].replaceData (
                                                        //System.out.print ((h*width + w) + "->
                                                }
                                        }
                                } else {        // 縦方向のスレッドと通信
                                        //System.out.println(g + "|");
                                        for (int h=0; h<hight; ++h) {
                                                for (int w=0; w<width; ++w) {
                                                        de[((h+1)%hight)*width + w].replaceData
                                                        //System.out.print ((h*width + w) + "->
                                                }
                                        }
                                }
                                //System.out.println();
                        }
                }
                recordFitness(g);
        }
}

static void hypercubeNetwork() {
        MonochromePicture opt[] = new MonochromePicture[NTH];
        double fit[] = new double[NTH];
```

```
        int neighbor = 1, pair = 2;

        System.out.println ("Hypercube network");
        //fp.println ("Hypercube network");
        //fp.print ("\t");
        //for (int i=0; i<NTH; ++i)
        //      fp.print (i + "\t");
        //fp.println();

        for (int g=0; g<sgen; ++g) {
                if (!isFindOptimum) { // 最適値を発見するまで局所を計算するDE
                        localDifferentialEvolution (g);
                        opt = getOptimum();
                        fit = getFitness();

                        int gv = 1;
                        if (LGEN > GV) gv = LGEN/GV;
                        if ((g % gv) == (gv-1)) {
                                for (int i=0; i<NTH; ++i) {
                                        if ((i%pair) < neighbor) {
                                                de[(i+neighbor) % NTH].replaceData (opt[i], fit
                                                //System.out.print(i+"->"+((i+neighbor) % NTH)+
                                        } else {
                                                de[(i-neighbor) % NTH].replaceData (opt[i], fit
                                                //System.out.print(i+"->"+((i-neighbor) % NTH)+
                                        }
                                }
                                //System.out.println();
                                neighbor *= 2;
                                pair *=2;
                                if (neighbor >= NTH) {
                                        neighbor = 1;
                                        pair = 2;
                                }
                        }
                }
                recordFitness(g);
        }
}

static void hierarchicalNetwork() {
        MonochromePicture opt[] = new MonochromePicture[NTH];
        double fit[] = new double[NTH];
        int neighbor = 1, pair = 2;

        System.out.println ("Hierarchical network");
        //fp.println ("Hierarchical network");
        //fp.print ("\t");
        //for (int i=0; i<NTH; ++i)
        //      fp.print (i + "\t");
        //fp.println();

        for (int g=0; g<sgen; ++g) {
```

```
                if (!isFindOptimum) { // 最適値を発見するまで局所を計算するDE
                        localDifferentialEvolution (g);
                        opt = getOptimum();
                        fit = getFitness();

                        int gv = 1;
                        if (LGEN > GV) gv = LGEN/GV;
                        if ((g % gv) == (gv-1)) {
                                neighbor = 1;
                                pair = 2;
                                while ((((g+1)/gv) % pair) == 0) {
                                        neighbor *= 2;
                                        pair *= 2;
                                }
                                while (pair > NTH) {
                                        neighbor = neighbor * 2/ NTH;
                                        pair = pair * 2 / NTH;
                                }
                                for (int i=0; i<NTH; ++i) {
                                        if ((i%pair) < neighbor) {
                                                de[(i+neighbor) % NTH].replaceData (opt[i], fit
                                                //System.out.print(i+"->"+((i+neighbor) % NTH)+
                                        } else {
                                                de[(i-neighbor) % NTH].replaceData (opt[i], fit
                                                //System.out.print(i+"->"+((i-neighbor) % NTH)+
                                        }
                                }
                                //System.out.println();
                        }
                }
                recordFitness(g);
        }
}

/**
 * ネットワーク無し
 * 局所計算スレッド間の通信を一切しない
 */
static void noNetwork() {
        System.out.println ("No network");
        //fp.println ("No network");
        //fp.print ("\t");
        //for (int i=0; i<NTH; ++i)
        //      fp.print (i + "\t");
        //fp.println();

        for (int g=0; g<sgen; ++g) {
                if (!isFindOptimum) { // 最適値を発見するまで局所を計算するDE
                        localDifferentialEvolution (g);
                }
                recordFitness(g);
        }
}
```

```
/**
 * 適合性表示
 */
static void printFitness() {
        System.out.print ("fitness : ");
        for (int i=0; i<NTH; ++i) {
                System.out.printf ("%4.2f ", de[i].getFitness());
        }
        System.out.println();
}

/**
 * 適合性表示
 * スーパー世代ごとの適合性を表示する
 * @param sg スーパー世代数
 */
static void printFitness (int sg) {
        System.out.printf ("fitness : %2d ", sg);
        for (int i=0; i<NTH; ++i) {
                System.out.printf ("%4.2f ", de[i].getFitness());
        }
        System.out.println();
}

static void printOptimum() {
        int iBest = 0;
        double fBest = de[0].getFitness();
        for (int i=1; i<NTH; ++i) {
                double f = de[i].getFitness();
                if (f < fBest) {
                        iBest = i;
                        fBest = f;
                }
        }
        System.out.println ("Target");
        target.showPicture();
        System.out.println ("Optimum");
        de[iBest].printOptimum();
        //de[iBest].outputOptimum (fp);
}

/**
 * 適合性出力
 * 適合性をファイルに出力する
 */
static void outputFitness() {
        fp.print ("fitness : ");
        for (int i=0; i<NTH; ++i) {
                fp.printf ("%4.2f\t", de[i].getFitness());
        }
        fp.println();
}
```

```
/**
 * 適合性出力
 * スーパー世代ごとの適合性をファイルに出力する
 * @param g スーパー世代数
 */
static void outputFitness (int sg) {
        fp.printf ("fitness\t%2d\t", sg);
        for (int i=0; i<NTH; ++i) {
                fp.printf ("%4.2f\t", de[i].getFitness());
        }
        fp.println();
}

/**
 * 全てのの中で最も高い適合性を返すLDE
 * @return double : 適合性最高値
 */
static double getBestFitness () {
        double fBest = de[0].getFitness();
        for (int i=1; i<NTH; ++i) {
                double f = de[i].getFitness();
                if (f<fBest) fBest = f;
        }
        return fBest;
}

/**
 * 全てのの中で最も高い適合性をもつ画像返すLDE
 * @return MonochromePicture : 適合性最高値の画像
 */
static MonochromePicture getBestOptimum (){
        double fBest = de[0].getFitness();
        MonochromePicture oBest = de[0].getOptimum();
        for (int i=1; i<NTH; ++i) {
                double f = de[i].getFitness();
                if (f<fBest) {
                        fBest = f;
                        oBest = de[i].getOptimum();
                }
        }
        return oBest;
}

/**
 * Fitness および Optimum の記録
 * @param g スーパー世代数
 */
static void recordFitness (int g) {
        double bestFitness = getBestFitness();
        fitnessRecord[roopIndex][g] = bestFitness;
        if (!isFindOptimum && bestFitness < EPSILON) {
                isFindOptimum = true;
```

```
                findOptimum[roopIndex] = g*lgen; // g*LGEN;
        }
        optimumRecord[roopIndex] = getBestOptimum();
}

/**
 * @return データ記録用ファイル名
 */
static String recordFileName () {
        String fname = "data/";
        if (CT == -1) {
                fname += "mix-";
        } else if (CT == 1) {
                fname += "bin-";
        } else {
                fname += "exp-";
        }
        if (IST == -1) {
                fname += "mix-";
        } else if (IST == 1) {
                fname += "rand-";
        } else {
                fname += "best-";
        }
        switch (NWT) {
        case 1:
                fname += "ring";
                break;
        case 2:
                fname += "torus";
                break;
        case 3:
                fname += "hypercube";
                break;
        case 4:
                fname += "hierarchical";
                break;
        default:
                fname += "no";
                break;
        }
        if (LGEN < 10) {
                fname += ("00"+LGEN);
        } else if (LGEN < 100) {
                fname += ("0"+LGEN);
        } else {
                fname += LGEN;
        }
        fname += ".txt";
        return fname;
}

static void outputFitnessRecord () {
```

```
if (CT == -1) {
        rfp.print ("Mix \t");
} else if (CT == 1) {
        rfp.print ("Bin \t");
} else {
        rfp.print ("Exp \t");
}
if (IST == -1) {
        rfp.print ("Mix\t");
} else if (IST == 1) {
        rfp.print ("Rand\t");
} else {
        rfp.print ("Best\t");
}
switch (NWT) {
case 1:
        rfp.print ("Ring network\t");
        break;
case 2:
        rfp.print ("Torus network\t");
        break;
case 3:
        rfp.print ("Hypercube network\t");
        break;
case 4:
        rfp.print ("Hierarchical network\t");
        break;
default:
        rfp.print ("No network\t");
        break;
}
rfp.println (LGEN);
rfp.print ("\t");
int gmax = sgen*lgen/GV;
if (gmax > 250) gmax = 250;
for (int g=0; g<gmax; ++g)
        rfp.printf ("%2d\t", g*GV);
rfp.println("findAt");
int count = 0;
int sum = 0;
int sqsum = 0;
int countTable[] = new int[8];
for (int i=0; i<countTable.length; ++i)
        countTable[i] = 0;
double fitnessSum[] = new double[gmax];
for (int i=0; i<fitnessSum.length; ++i) {
        fitnessSum[i] = 0;
}
int ct[] = new int[LGEN*SGEN];
for (int i=0; i<ct.length; ++i) {
        ct[i] = 0;
}
```

```
for (int r=0; r<ROOPN; ++r) {
        rfp.printf ("%2d\t", r);
        if (LGEN<GV) {
                for (int g=0; g<gmax; ++g) {
                        rfp.printf ("%4.3f\t", fitnessRecord[r][g*GV/LGEN]);
                        fitnessSum[g] += fitnessRecord[r][g*GV/LGEN];
                }
        } else {
                for (int g=0; g<gmax; ++g) {
                        rfp.printf ("%4.3f\t", fitnessRecord[r][g]);
                        fitnessSum[g] += fitnessRecord[r][g];
                }
        }
        if (findOptimum[r] < lgen*sgen) {
                rfp.println (findOptimum[r]);
                count++;
                sum += findOptimum[r];
                sqsum += findOptimum[r] * findOptimum[r];
                if (findOptimum[r] < 16) {
                        ++countTable[0];
                } else if (findOptimum[r] < 32) {
                        ++countTable[1];
                } else if (findOptimum[r] < 64) {
                        ++countTable[2];
                } else if (findOptimum[r] < 128) {
                        ++countTable[3];
                } else if (findOptimum[r] < 256) {
                        ++countTable[4];
                } else if (findOptimum[r] < 512) {
                        ++countTable[5];
                } else if (findOptimum[r] < 1024) {
                        ++countTable[6];
                } else if (findOptimum[r] < 2048) {
                        ++countTable[7];
                }
                ++ct[findOptimum[r]];
        } else {
                rfp.println ("not yet");
        }
}
rfp.print ("av\t");
for (int g=0; g<gmax; ++g) {
        rfp.printf ("%4.3f\t", (fitnessSum[g] / ROOPN));
}
rfp.println();
for (int i=1; i<countTable.length; ++i) {
        countTable[i] += countTable[i-1];
}
if (count > 0) {
        double av = (double) sum / count;
        if (count > 1) {
                double stdev = Math.sqrt ((double) (sqsum - av*av*count) / (count-1));
                rfp.print (count + "\t" + av + "\t" + stdev);
```

```
                        System.out.println (count + " av:" + av + " stdev:" + stdev);
                } else {
                        rfp.print (count + "\t" + av + "\tnon");
                        System.out.println (count + " av:" + av + " stdev:non");
                }
        } else {
                rfp.print ("0\tnon\tnon");
                System.out.println ("0 av:non stdev:non");
        }
        for (int i=0; i<countTable.length; ++i) {
                rfp.print ("\t" + countTable[i]);
        }
        rfp.println();
        for (int r=0; r<ROOPN; ++r) {
                rfp.println ("Loop : " + r);
                rfp.println ("Target");
                targets[r].printPicture (rfp);
                rfp.println ("Optimum");
                optimumRecord[r].printPicture (rfp);
                rfp.printf ("%4.3f", fitnessRecord[r][SGEN-1]);
                rfp.println();
        }
        for (int i=1; i<ct.length; ++i) {
                ct[i] += ct[i-1];
        }
        for (int i=0; i<512; i+=4) {
                rfp.print (i + "\t");
        }
        rfp.println();
        for (int i=0; i<512; i+=4) {
                rfp.print (ct[i] + "\t");
        }
        rfp.println();
    }
}
```

.1 LocalDifferentialEvolution クラス

```
package de;
import java.util.Random;
import java.io.*;

/**
 * 与えられたぼやけた値画像の元画像を進化計算により得る2
 */
public class LocalDifferentialEvolution implements Runnable {
        MonochromePicture pop[];  // 母集団(population)
        double fit[];             // 母集団の適合性(fitness of the population)
        int sizePop;              // 母集団の大きさ(size of population)
        double diffCons;          // 差分定数(differential constant) 現在は未使用拡張用(:)
        double crossCons;         // 交叉定数(crossover constant)
        double mutationCons;      // 突然変異定数(mutation constant)
        int generation;           // 世代数(number of generations)
```

```
int iBest;                  // 最適解のインデックス
Random rd;                  // 乱数発生用クラスのインスタンス
MonochromePicture target; // 目標となる値画像2
int pictureSize;            // 画像のサイズ

int crossType;          // 交叉のタイプ
/**
 *  1: bin
 * oh: exp
 */
int indexSelectType;    // インデックス選択のタイプ
/**
 *  1: rand
 * ot: best
 */
int thNum;              // スレッド番号

/**
 * コンストラクタ
 * @param int s 母集団のサイズ
 * @param double dc 差分定数
 * @param double cc 交叉定数
 * @param double mc 突然変異定数
 * @param int gen 世代数
 * @param int ct 交叉のタイプ
 * @param int ist : インデックスの選択タイプランダム（ベストor）
 * @param MonochromePicture mp : 目標画像
 * @ramam int tn スレッド番号
 */
LocalDifferentialEvolution (int s, double dc, double cc, double mc, int gen, int ct, int ist, M
        sizePop = s;
        diffCons = dc;
        crossCons = cc;
        mutationCons = mc;
        generation = gen;
        if (ct == -1) {
                if (tn % 2 == 0) {
                        crossType = 0;
                } else {
                        crossType = 1;
                }
        } else {
                crossType = ct;
        }
        if (ist == -1) {
                if (tn % 4 < 2) {
                        indexSelectType = 0;
                } else {
                        indexSelectType = 1;
                }
        } else {
                indexSelectType = ist;
        }
```

```
        rd = new Random();
        target = mp;
        pictureSize = mp.size;
        pop = makePopulation();              // 解の母集団設定
        fit = new double[sizePop];
        setInitialFitness();                // 適合性の初期値設定

        thNum = tn;
}

/**
 * 解の母集団表示関数
 */
public void printPopulation () {
        for (int i=0; i<sizePop; ++i) {
                if (i == iBest) {
                        System.out.println("best");
                } else {
                        System.out.println();
                }
                pop[i].showPicture();
        }
}

/**
 * 最適解表示
 */
public void printOptimum () {
        pop[iBest].showPicture();
        System.out.println ("fitness : " + pop[iBest].getValue());
}

/**
 * 最適解出力
 * @param printWriter fp : 出力ファイルへのポインタ
 */
public void outputOptimum (PrintWriter fp) {
        pop[iBest].printPicture (fp);
        fp.println ("fitness\t" + pop[iBest].getValue());
}

/**
 * 最適解出力
 * @return MonochromePicture : 最適解
 */
public MonochromePicture getOptimum () {
        return pop[iBest];
}

/**
 * 適合性出力
 * @return doulbe 適合性
```

```
 */
public double getFitness () {
        return fit[iBest];
}

/**
 * データ入れ替え
 * 入力されたデータを母集団の最適解を除く解つと入れ替える1
 * @param MonochromePicture mp : 入れ替えデータ
 */
public void replaceData (MonochromePicture mp) {
        int r;
        do {
                r = rd.nextInt(sizePop);
        } while (r == iBest);
        pop[r] = mp.clone();
        fit[r] =  mp.getValue();
        if (fit[r] <= fit[iBest]) iBest = r;
}

/**
 * データ入れ替え
 * 入力されたデータを母集団の最適解を除く解つと入れ替える1
 * @param MonochromePicture mp : 入れ替えデータ
 * @param doulbe fit : 適応値
 */
public void replaceData (MonochromePicture mp, double f) {
        int r;
        do {
                r = rd.nextInt(sizePop);
        } while (r == iBest);
        pop[r] = mp.clone();
        fit[r] = f;
        if (f <= fit[iBest]) iBest = r;
}


public void run () {
        double f;                               // fitness of the trial individual
        int ri;                                 // ランダム選択インデックス(randomly selected indices)

        for (int g=0; g<generation; ++g) {
                /*System.out.printf("tn = %d, g = %3d ", thNum, g);
                printOptimum();
                System.out.printf (" Fobj = %4.3f\n", fit[iBest]);*/

                for (int i=0; i<sizePop; ++i) {
                        MonochromePicture mp;
                        if (indexSelectType == 1) { // ランダム要素選択
                                do {
                                        ri = rd.nextInt (sizePop); // ランダムに要素
を選択
                                } while (i == ri);
```

```
                                } else {                        // 最適要素選択
                                        if (i != iBest) {
                                                ri = iBest;                     // 現時点の最適要
素を選択
                                        } else {
                                                do {
                                                        ri = rd.nextInt (sizePop); // ラ
ンダムに要素を選択
                                                } while (i == ri);
                                        }
                                }

                                if (crossType == 1) { // 二項交叉
                                        mp = binCross (pop[i], pop[ri]);
                                } else { // 指数交叉
                                        mp = expCross (pop[i], pop[ri]);
                                }

                                /* より良い解が見つかれば値を更新 */
                                f = mp. evaluateValue (target);
                                if (f <= fit[i]) {
                                        pop[i] = mp;
                                        fit[i] = f;

                                        if (f <= fit[iBest])
                                                iBest = i;
                                }
                                //System.out.println (g+":"+i+"---");
                                //pop[i].showPicture();
                        }
                        mutation(); // 一定確率で突然変異を起こす

                        /* 途中経過出力 */
                        //if (g%32 == 0) {
                        //      System.out.println (g+":"+iBest+"---");
                        //      printOptimum();
                        //}
                }
/*
                fnc.printType();
                System.out.printf("g = %3d ", generation);
                printOptimum ();
                System.out.printf (" Fobj = %4.3f\n", fit[iBest]);*/
        }

        /**
         * 解の母集団作成
         * @return 解の母集団
         */
        private MonochromePicture[] makePopulation () {
                MonochromePicture pop[] = new MonochromePicture[sizePop];

                for (int i=0; i<sizePop; ++i) {
```

```
                pop[i] = new MonochromePicture (pictureSize);
        }
        return pop;
}

/**
 * fitness の初期値設定
 */
private void setInitialFitness () {
        for (int i=0; i<sizePop; ++i) {
                fit[i] = pop[i].evaluateValue (target);
                        //System.out.println (i + ": " + fit[i]);
        }

        iBest = 0;
        for (int i=1; i<sizePop; ++i) {
                if (fit[i] < fit[iBest])
                        iBest = i;
        }
}

/**
 * つの画像から交叉画像二項交叉を求める2()
 */
MonochromePicture binCross (MonochromePicture mp1, MonochromePicture mp2) {
        double[][] p1 = mp1.getPicture();
        double[][] p2 = mp2.getPicture();
        double[][] p = new double [pictureSize][pictureSize];

        for (int x=0; x<pictureSize; ++x)
                for (int y=0; y<pictureSize; ++y) {
                        if (rd.nextDouble() <crossCons) // 一定確率でどちらかの親父(母or
を選択)
                                p[x][y] = p2[x][y];
                        else
                                p[x][y] = p1[x][y];
                }
        return new MonochromePicture (p);
}

/**
 * つの画像から交叉画像指数交叉を求める2()
 */
MonochromePicture expCross (MonochromePicture mp1, MonochromePicture mp2) {
        double[][] p1 = mp1.getPicture();
        double[][] p2 = mp2.getPicture();
        double[][] p = new double [pictureSize][pictureSize];

        boolean expSwitch = (rd.nextDouble() <crossCons);

        for (int x=0; x<pictureSize; ++x)
                for (int y=0; y<pictureSize; ++y) {
                        if (expSwitch)
```

```
                                        p[x][y] = p2[x][y];
                                else
                                        p[x][y] = p1[x][y];
                                if (rd.nextDouble() <crossCons) expSwitch = !expSwitch; //
一定確率で親父⇔母を入れ替える()
                        }
                return new MonochromePicture (p);
        }

        /**
         * 一定確率で要素の一つに突然変異を起こす
         */
        void mutation() {
                if (rd.nextDouble() <mutationCons) {
                        int ri;
                        do {
                                ri = rd.nextInt (sizePop); // ランダムに要素を選択
                        } while (iBest == ri);
                        MonochromePicture mp = new MonochromePicture (pictureSize);
                        pop[ri] = mp;
                }
        }

        /**
         * テスト用メインメソッド
         * ローカルに進化計算を行い画像を求める。
         */
        public static void main (String[] args) {
                LocalDifferentialEvolution de;
                MonochromePicture tar = new MonochromePicture(16);

                de = new LocalDifferentialEvolution (16, 0.8, 0.5, 0.3, 1024, 1, 1, tar, 0);

                de.run();
                System.out.println 目標画像("");
                tar.showPicture();
                System.out.println 進化計算により得られた画像("");
                de.printOptimum();
        }
}
```

.1 MonochromePicture クラス

```
package de;

import java.util.Random;
import java.io.*;
import java.awt.Graphics;
import java.awt.Font;
import java.awt.Color;

/**
 * 値画像を作成するクラス2
```

```
 * 値画像と、そのぼやけた画像を作成する2
 */
public class MonochromePicture implements Cloneable {
        double[][] picture;         // 元画像
        double[][] dimPicture;      // ぼやけた画像
        Random rd;                  // 乱数生成用
        int size;                   // 画像サイズ
        double value;               // 対象画像との距離

        /**
         * コンストラクタ
         * 画像中にランダムな大きさの黒い正方形を描いた図を作成する
         * @param int s : 画像のサイズ
         */
        MonochromePicture (int s) {
                size = s;
                rd = new Random();
                picture = new double[size][size];
                dimPicture = new double[size][size];

                /* 画像を白で初期化 */
                for (int x=0; x<size; ++x)
                        for (int y=0; y<size; ++y)
                                picture[x][y] = 0;

                /* 画像中にランダムな大きさの黒い正方形を描く */
                for (int i=0; i<5; ++i) {
                        int sqsize = rd.nextInt(8);
                        int left = rd.nextInt(size-sqsize);
                        int right = left+sqsize;
                        int top = rd.nextInt(size-sqsize);
                        int bottom = top+sqsize;

                        for (int x=left; x<right; ++x)
                                for (int y=top; y<bottom; ++y)
                                        picture[x][y] = 1;
                }
                dim();  // ぼやけた画像を描く
                value = Double.MAX_VALUE;
        }

        /**
         * コンストラクタ
         * 引数で指定した図を作成する
         * @param double[][] p : 指定する図
         */
        MonochromePicture (double[][] p) {
                size = p.length;
                picture = new double[size][size];
                dimPicture = new double[size][size];
                for (int x=0; x<size; ++x)
                        for (int y=0; y<size; ++y)
                                picture[x][y] = p[x][y];
```

```
        dim();
        value = Double.MAX_VALUE;
}

/**
 * コンストラクタ
 * 引数で指定した図を作成する
 * @param MonochromePicture mp : 指定する図
 */
MonochromePicture (MonochromePicture mp) {
        size = mp.size;
        picture = new double[size][size];
        dimPicture = new double[size][size];
        for (int x=0; x<size; ++x)
                for (int y=0; y<size; ++y) {
                        picture[x][y] = mp.picture[x][y];
                        dimPicture[x][y] = mp.dimPicture[x][y];
                }
        value = mp.getValue();;
}

/**
 * インスタンスのクローンを作成する
 * @return MonochromePicture : インスタンスのクローン
 */
public MonochromePicture clone() {
        return new MonochromePicture (this);
}

/**
 * 元画像を得る
 * @return double[][] : 元画像
 */
double[][] getPicture () {
        return picture;
}

/**
 * ぼやけた画像を得る
 * @return double[][] : ぼやけた画像
 */
double[][] getDimPicture () {
        return dimPicture;
}

/**
 * 評価値を得る
 * @return double : 評価値
 */
double getValue() {
        return value;
}
```

```
/**
 * 画像をぼやけさせる
 * 各画素に対して、周囲画像の平均値を取る8
 */
void dim() {
        double pixel;

        pixel = 0;
        for (int vx=0; vx<=1; ++vx)
                for (int vy=0; vy<=1; ++vy)
                        pixel += picture[vx][vy];
        dimPicture[0][0] = pixel/4;

        for (int y=1; y<size-1; ++y) {
                pixel = 0;
                for (int vx=0; vx<=1; ++vx)
                        for (int vy=-1; vy<=1; ++vy)
                                pixel += picture[vx][y+vy];
                dimPicture[0][y] = pixel/6;
        }

        pixel = 0;
        for (int vx=0; vx<=1; ++vx)
                for (int vy=-1; vy<=0; ++vy)
                        pixel += picture[vx][size-1+vy];
        dimPicture[0][size-1] = pixel/4;

        for (int x=1; x<size-1; ++x) {

                pixel = 0;
                for (int vx=-1; vx<=1; ++vx)
                        for (int vy=0; vy<=1; ++vy)
                                pixel += picture[x+vx][vy];
                dimPicture[x][0] = pixel/6;

                for (int y=1; y<size-1; ++y) {
                        pixel = 0;
                        for (int vx=-1; vx<=1; ++vx)
                                for (int vy=-1; vy<=1; ++vy)
                                        pixel += picture[x+vx][y+vy];
                        dimPicture[x][y] = pixel/9;
                }

                pixel = 0;
                for (int vx=-1; vx<=1; ++vx)
                        for (int vy=-1; vy<=0; ++vy)
                                pixel += picture[x+vx][size-1+vy];
                dimPicture[x][size-1] = pixel/6;
        }

        pixel = 0;
        for (int vx=-1; vx<=0; ++vx)
                for (int vy=0; vy<=1; ++vy)
```

```
                        pixel += picture[size-1+vx][vy];
        dimPicture[size-1][0] = pixel/4;

        for (int y=1; y<size-1; ++y) {
                pixel = 0;
                for (int vx=-1; vx<=0; ++vx)
                        for (int vy=-1; vy<=1; ++vy)
                                pixel += picture[size-1+vx][y+vy];
                dimPicture[size-1][y] = pixel/6;
        }

        pixel = 0;
        for (int vx=-1; vx<=0; ++vx)
                for (int vy=-1; vy<=0; ++vy)
                        pixel += picture[size-1+vx][size-1+vy];
        dimPicture[size-1][size-1] = pixel/4;
}

/**
 * 対象画像との近さを評価する
 * @param double[][] p : 対象画像
 * @return double : 対象画像との距離
 */
double evaluateValue (double[][] p) {
        value = 0;
        for (int x=0; x<size; ++x)
                for (int y=0; y<size; ++y) {
                        double diff = dimPicture[x][y] - p[x][y];
                        double sq = diff * diff;
                        value += sq;
                }
        return value;
}

/**
 * 対象画像との近さを評価する
 * @param MonochromePicture mp : 対象画像
 * @return double : 対象画像との距離
 */
double evaluateValue (MonochromePicture mp){
        double[][] p = mp.getDimPicture();
        value = 0;
        for (int x=0; x<size; ++x)
                for (int y=0; y<size; ++y) {
                        double diff = dimPicture[x][y] - p[x][y];
                        double sq = diff * diff;
                        value += sq;
                }
        return value;
}

/**
 * 画像間の近さを評価する
```

```
 * @param double[][] p1 : 対象画像
 * @param double[][] p2 : 対象画像
 * @return double : 対象画像間の距離
 */
static double evaluateValue (double[][] p1, double[][] p2) {
        double v = 0;
        int size = p1.length;
        for (int x=0; x<size; ++x)
                for (int y=0; y<size; ++y) {
                        double diff = p1[x][y] - p2[x][y];
                        double sq = diff * diff;
                        v += sq;
                }
        return v;
}

/**
 * 画像間の近さを評価する
 * @param MonochromePicture mp1 : 対象画像
 * @param MonochromePicture mp2 : 対象画像
 * @return double : 対象画像間の距離
 */
static double evaluateValue (MonochromePicture mp1, MonochromePicture mp2) {
        double[][] p1 = mp1.getDimPicture();
        double[][] p2 = mp2.getDimPicture();
        double v = 0;
        int size = mp1.size;
        for (int x=0; x<size; ++x)
                for (int y=0; y<size; ++y) {
                        double diff = p1[x][y] - p2[x][y];
                        double sq = diff * diff;
                        v += sq;
                }
        return v;
}

/**
 * 元画像を表示する
 */
void showPicture () {
        for (int x=0; x<size; ++x) {
                for (int y=0; y<size; ++y) {
                        if (picture[x][y] == 0) System.out.print ("  ");
                        else System.out.print ("* ");
                }
                System.out.println ();
        }
}

/**
 * ぼやけた画像を表示する
 */
void showDimPicture () {
```

```
        for (int x=0; x<size; ++x) {
                for (int y=0; y<size; ++y) {
                        double d = dimPicture[x][y];
                        d *= 8;
                        int i = (int) d;
                        System.out.print(i+" ");
                }
                System.out.println ();
        }
}

/**
 * 元画像を出力する
 * @param printWriter fp : 出力ファイルへのポインタ
 */
void printPicture (PrintWriter fp) {
        for (int x=0; x<size; ++x) {
                for (int y=0; y<size; ++y) {
                        if (picture[x][y] == 0) fp.print ("  ");
                        else fp.print ("* ");
                }
                fp.println ();
        }
}

/**
 * ぼやけた画像を表示する
 * @param printWriter fp : 出力ファイルへのポインタ
 */
void printDimPicture (PrintWriter fp) {
        for (int x=0; x<size; ++x) {
                for (int y=0; y<size; ++y) {
                        double d = dimPicture[x][y];
                        d *= 8;
                        int i = (int) d;
                        fp.print(i+" ");
                }
                fp.println ();
        }
}



/**
 * テスト用メインメソッド
 */
public static void main (String[] args) {
        MonochromePicture mp = new MonochromePicture (16);
        mp.showPicture();
        mp.dim();
        mp.showDimPicture();
        MonochromePicture mp2 = new MonochromePicture (16);
        mp2.showPicture();
```

```
                mp2.dim();
                mp2.showDimPicture();
                double value = mp.evaluateValue (mp2);
                System.out.println (value);

        }
}
```