手書き数字認識の
計算機シミュレーション

2018 年 8 月

法政大学情報科学部

若原徹

目次

1　開発環境

【プログラミング言語】

C 言語を用いる。下記 URL：

https://waka.cis.k.hosei.ac.jp

の C Programming Language Lecture Notes にある講義ノートを参考に勉強すること。

特に,第 8 回講義ノート Cpro_8.htm で取り上げたプログラミングスタイルが重要である。以下に列挙するが，詳細は講義ノートを参照のこと。

［1］ 名前のつけかた
- グローバル変数にはわかりやすい名前を，ローカル変数には短い名前を。
- 関連性のあるものには関連性のある名前をつけて，統一しよう。
- 関数には能動的な名前を。
- 名前は的確に。

［2］ 式と文
- 構造がわかるようにインデントしよう。
- 自然な形の式を使おう。
- カッコを使ってあいまいさを解消しよう。
- 複雑な式は分割しよう。
- 明解に書こう。
- 副作用（++などの演算子）に注意。

［3］ 一貫性と慣用句
- インデントとブレースのスタイルを統一しよう。
- 慣用句によって一貫性を確保しよう。
- 多分岐の判定には else-if を使おう。

［4］ マクロやリテラルな数値の使い方
- 関数マクロはなるべく使うな。
- マクロの本体と引数はかっこに入れよう。
- 数値はマクロでなく定数として定義しよう。
- 整数でなく文字定数を使おう。
- オブジェクトサイズは言語に計算させよう。

【プログラミング開発環境】

Linux 上の gcc を用いることを推奨する。また，拡張子.c が付く C 言語のソースコードファイルを作成するテキストエディタには emacs が高機能で使い易い。

Linux で用いるコマンドについては，例えば，

https://rat.cis.k.hosei.ac.jp/article/linux/command.html

を参照すること。良く使うコマンド rm，cp，mv については，ホームディレクトリにあるログインシェルファイル.bashrc を

```
$ emacs ~/.bashrc &↵
```

で開いて，動作確認のオプション-i を含むエイリアスを追記しておくと良い。

```
alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
```

作業環境の立ち上げでは，まず，Linux でログインしたホームディレクトリの下に作業ディレクトリ project を

```
$ mkdir project↵
```

によって作成する。次に，作業ディレクトリ project に

```
$ cd project↵
```

により移動する。以降，主にこのディレクトリでプログラミング開発を行う。

ソースコードファイル sample.c のエディット・コンパイル・リンク・実行のしかたは次のようになる。

```
$ emacs sample.c &↵      ←  ソースコードファイル sample.c を作成する。
$ gcc -c sample.c↵       ←  コンパイルが実行されて，オブジェクトコードファイル
                             sample.o ができる。
$ gcc sample.o↵          ←  リンクが実行されて，実行可能コードファイル a.out が
                             できる。実行は  $ ./a.out↵
$ gcc sample.c↵          ←  コンパイル・リンクが実行されて，a.out ができる。
                             実行は  $ ./a.out↵
$ gcc sample.c -o sample↵  ←  実行可能コードファイルの名前を指定して sample を
                             作る。実行は  $ ./sample↵
```

ただし，数学関数を利用するため，ヘッダファイル math.h を

```
#include <math.h>
```

でインクルードしてあるソースコードファイル sample.c をコンパイル･リンクする場合は，次のようにオプション-lm（ハイフンエルエム）を必ず付けること。

```
$ gcc sample.c -o sample -lm↵
```

※Windows 上で開発する場合

統合開発環境 Visual C++ または Visual C++ Express を用いて Win32 コンソールアプリケーションを立ち上げること。

2　手書き数字データの入手

【IPTP CDROM1B】

年賀葉書に記入された 3 桁郵便番号から採取した手書き数字の 2 値画像を整備したデータベース IPTP CDROM1B を用いる。

上記データベースは下記 URL:

https://waka.k.hosei.ac.jp

の IPTP CDROM1B から入手できる。

cdrom1b.txt には当該データベースの仕様に関する情報が記述されている。特に，学習用データ learn.pat とテスト用データ test.pat に含まれる 0～9 の各数字のサンプル数が重要である。

画像データ learn.pat と test.pat は数字の 2 値画像データを圧縮して格納してある。作業ディレクトリ project の下に新しいディレクトリ cdrom1b を作成して，

```
$ mkdir cdrom1b↵
```

これら learn.pat と test.pat をダウンロードする。

learn.pat と test.pat を解凍して，各サンプルの画像データファイルを作成するプログラムが PGM_digit.c である。これを project にダウンロードしてから，コンパイル・リンクを行い，実行可能ファイル PGM_digit を作成する。この PGM_digit をディレクトリ cdrom1b にコピーする。

```
$ cp PGM_digit ./cdrom1b↵
```

次に，ディレクトリ cdrom1b に移動して，学習用データ learn.pat を解凍する。

```
$ ./PGM_digit learn.pat↵
```

を実行すると，ldg 数字_通し番号.pgm の名前で各サンプルの画像がバイナリ pgm 形式で作成される。ldg0_0001.pgm～ldg9_1428.pgm までの総数 17,985 枚の画像となる。通し番号は左側 0 詰めの 4 桁数字である。

続いて，同様に，テスト用データ test.pat を解凍する。

```
$ ./PGM_digit test.pat↵
```

を実行すると，tdg 数字_通し番号.pgm の名前で各サンプルの画像がバイナリ pgm 形式で作成される。tdg0_0001.pgm～tdg9_1140.pgm までの総数 17,916 枚の画像となる。

各画像データは 1 サンプルの手書き数字の 2 値画像を格納しており，画像サイズは縦 120，横 80 である。バイナリ pgm 形式の場合，各画素の濃淡値 0～255 は 8 ビットの 2 進数で表現されるため，1 バイト／画素となっている。ただし，今回の手書き数字画像は 2 値画像であり，文字部分が値 0（黒）で背景部分が値 255（白）となっている。

【ヘッダファイル mypgm.h とプログラミングの流れ】

下記 URL:

https://waka.cis.k.hosei.ac.jp

のC Program Source Codesの先頭にあるヘッダファイル

mypgm.h

を作業ディレクトリ project にダウンロードする。これから作成するプログラムには必ずこのヘッダファイルを

#include “mypgm.h”

でインクルードする。

mypgm.hには，#defineを用いたマクロの定義と，下記の外部変数の宣言および関数定義がなされている。

```
/* declaration of global variables */
unsigned char image1[1024][1024], image2[1024][1024];
int x_size1, y_size1, /* width & height of image1 */
  x_size2, y_size2; /* width & height of image2 */

/* prototype declaration of functions */
void load_image_data( ); /* image input */
void save_image_data( ); /* image output */
void load_image_file(char *); /* image input */
void save_image_file(char *); /* image output */
```

ヘッダファイルmypgm.hを用いたプログラミングは次のような流れで行う。

[1] 入力画像格納用の 2 次元配列 image1[][]に，関数 load_image_data()または load_image_file(filename)を用いて，ハードディスクから pgm 形式の画像を読み込む。読み込んだ画像の横および縦サイズはそれぞれ変数 x_size1, y_size1 に自動で入る。

[2] image1[][]に格納された入力画像に，目的とする画像処理を適当に関数で定義して施す。処理結果は何らかの数値データであったり，画像であったりする。処理結果の画像を出力する場合は，2次元配列 image2[][]に格納する。格納する画像の横および縦サイズはそれぞれ変数 x_size2, y_size2 に忘れずに必ず値を設定する。

[3] 出力画像格納用の2次元配列 image2[][]の内容をハードディスクにpgm形式で出力する場合は，関数 save_image_data()または save_image_file(filename)を用いる。変数 x_size2, y_size2 に正しい値が入っていないとエラーとなる。

プログラミング例としては，入力画像の反転画像を出力するプログラム inverse.c：

https://waka.cis.k.hosei.ac.jp/inverse.c

を参照のこと。

【画像の閲覧】

画像ビューワーには gimp を用いると良い。作業ディレクトリ project からは，次のようにして画像を表示する。

```
$ gimp ~/cdrom1b/ldg0_0001.pgm &
```

※Windows 上での画像の閲覧

次の URL:

https://forest.watch.impress.co.jp/library/software/irfanview/

から画像ビューワーIrfanView を入手すると良い。

3　位置と大きさの正規化

【前処理】

一般に，パターン認識の処理の流れは

パターンの観測　→　前処理　→　特徴抽出　→　識別処理　→　カテゴリ出力

となっている。手書き数字認識を例に取ると，「パターンの観測」は手書き数字の画像入力である。最後の「カテゴリ出力」は入力された手書き数字画像の認識結果であり，クラス名0～9の出力である。「特徴抽出」と「識別処理」については後述する。

「前処理」であるが，これは特徴抽出に先立って，雑音除去や平滑化処理，さらには整形操作を施す処理であり，認識精度の向上に大きく貢献する。

今回の手書き数字認識では，整形操作の一種である「位置と大きさの正規化」を前処理として施す。位置と大きさの正規化処理にも様々な手法があるが，ここではモーメント法に基づく手法を採用する。

【位置の正規化】

画像内の文字部分（黒画素領域）の重心(gx, gy)を求めて，重心が画像の中心(40, 60)にくるように文字部分を平行移動する。この操作が位置の大きさの正規化となる。また，重心を求める操作が画像の1次モーメント算出に相当する。

重心の算出は次のようなコーディングになる。

```
#define BLACK 0
#define WHITE 255
…
int main() {
  …
  int x, y;
  int count;
  double gx, gy;
  …
  gx = gy = 0.0;
  count = 0;
  for (y = 0; y < y_size1; y++) {
    for (x = 0; x < x_size1; x++) {
      if (image1[y][x] == BLACK) {
        gx += x;
        gy += y;
        count++;
```

```
      }
    }
  }
  gx /= (double)count;
  gy /= (double)count;
  …
}
```

【大きさの正規化】

画像内の文字部分（黒画素領域）の重心(gx, gy)を求めて，文字を構成する各黒画素から重心までの平均距離 rad を算出する。予め定めた正規化半径 RADIUS と rad の値が等しくなるように，文字部分を重心の周りに一様に伸縮する。この操作が大きさの正規化となる。また，文字を構成する各黒画素から重心までの平均距離 rad を求める操作が画像の 2 次モーメント算出に相当する。

各黒画素から重心までの平均距離 rad の算出は次のようなコーディングになる。gx, gy, count の値は既に算出されているとする。

```
#include <math.h>
…
int main() {
  …
  int x, y;
  double rad;
  …
  rad = 0.0;
  for (y = 0; y < y_size1; y++) {
    for (x = 0; x < x_size1; x++) {
      if (image1[y][x] == BLACK) {
        rad += sqrt((x-gx)*(x-gx)+(y-gy)*(y-gy));
      }
    }
  }
  rad /= (double)count;
  …
}
```

【双一次補間法と逆写像を用いた位置と大きさの正規化】

入力画像 image1[][]に位置と大きさの正規化を施した画像を image2[][]に格納するものとする。このとき，次のような計算手順を用いる。

[1] image2[][]の各位置(X,Y)が，位置と大きさの正規化処理の逆写像によって image1[][]のどの位置に対応するかを計算する。その位置を(tx, ty)と記すと，一般に tx, ty は整数ではなく実数となる。

[2] 位置(tx, ty)における image1[][]の濃淡値 gray を双一次補間法によって算出する。

[3] gray の値を，閾値 127.5 により，0 もしくは 255 に 2 値化する。この 2 値化された値 val を image2[Y][X]に代入する。

上記の処理は，以下のようなコーディングになる。gx, gy, rad の値は既に算出されているとする。ただし，正規化半径 RADIUS は 25.0 とした。

```
#define RADIUS 25.0
#define TH 127.5
#define WID 80
#define HEI 120
#define CX (WID/2)
#define CY (HEI/2)
…
int main() {
  …
  int X, Y;
  int m, n, val;
  double tx, ty, p, q, gray;
  …
  x_size2 = x_size1;
  y_size2 = y_size1;
  for (Y = 0; Y < y_size2; Y++) {
    for (X = 0; X < x_size2; X++) {
      /* inverse mapping by position and size normalization */
      tx = rad/RADIUS * (X-CX) + gx;
      ty = rad/RADIUS * (Y-CY) + gy;
      m = (int)floor(tx);
      p = tx - m;
      n = (int)floor(ty);
```

```
      q = ty - n;
      if (m >= 0 && m+1 < x_size1 && n >= 0 && n+1 < y_size1) {
        /* bilinear interpolation */
        gray = (1.0 - q)*((1.0 - p)*image1[n][m] + p*image1[n][m+1]) +
          q*((1.0 - p)*image1[n+1][m] + p*image1[n+1][m+1]);
        /* binarization */
        val = gray >= TH ? WHITE : BLACK;
      } else {
        val = WHITE;
      }
      image2[Y][X] = (unsigned char)val;
    }
  }
  …
}
```

4　特徴抽出

パターン認識では，特徴抽出が認識性能を大きく左右する。特徴空間が定まってしまえば，その特徴空間内でのサンプルの分布に基づき，統計的パターン認識理論により強力な識別器が構成できる。

しかし，最適な特徴抽出の理論は確立していない。このため，特に文字認識研究の分野で，これまでに多くの研究者の試行錯誤により様々な特徴抽出が検討されてきた。

ここでは，最も素朴ながら基本的な特徴としての「メッシュ特徴」，文字線の方向分布に着目した有力な「輪郭方向分布特徴」を取り上げる。

輪郭方向特徴の発展形として，2 値画像でなく濃淡画像に適用される，「濃淡勾配方向分布特徴」がある。これは，文字認識に限らず，広くコンピュータビジョンの分野で HOG(Histograms of Oriented Gradients)特徴量と呼ばれて活用されている。

なお，本節で扱う手書き数字画像は，前節の「位置と大きさの正規化」処理を施された後の画像であるとする。

4.1　メッシュ特徴

手書き数字画像を複数のブロック領域に分割し，各ブロック領域に含まれる黒画素の比率を算出する。この黒画素比率をブロック数分だけ並べたベクトルをメッシュ特徴と呼ぶ。

具体的には，縦 120 画素×横 80 画素の手書き数字画像を 10×10 画素の正方ブロック領域に分割することにより，縦 12×横 8 の総数 96 個のブロック領域を生成する。各ブロックの黒画素比率を並べた特徴ベクトルの次元数は 12×8＝96 となる。

メッシュ特徴 fv_mesh[96]の抽出処理は次のようなコーディングになる。

```
#define X_BLOCK 10
#define Y_BLOCK 10
#define DIM ((WID/X_BLOCK)*(HEI/Y_BLOCK))
…
int main() {
  …
  int x, y, k;
  double fv_mesh[DIM];
  …
  for (k = 0; k < DIM; k++) {
    fv_mesh[k] = 0.0;
  }
  for (y = 0; y < y_size2; y++) {
    for (x = 0; x < x_size2; x++) {
```

```
      if (image2[y][x] == BLACK) {
        k = y/Y_BLOCK*(WID/X_BLOCK)+x/X_BLOCK;
        fv_mesh[k] += 1.0;
      }
    }
  }
  for (k = 0; k < DIM; k++) {
    fv_mesh[k] /= (double)X_BLOCK*Y_BLOCK;
  }
  …
}
```

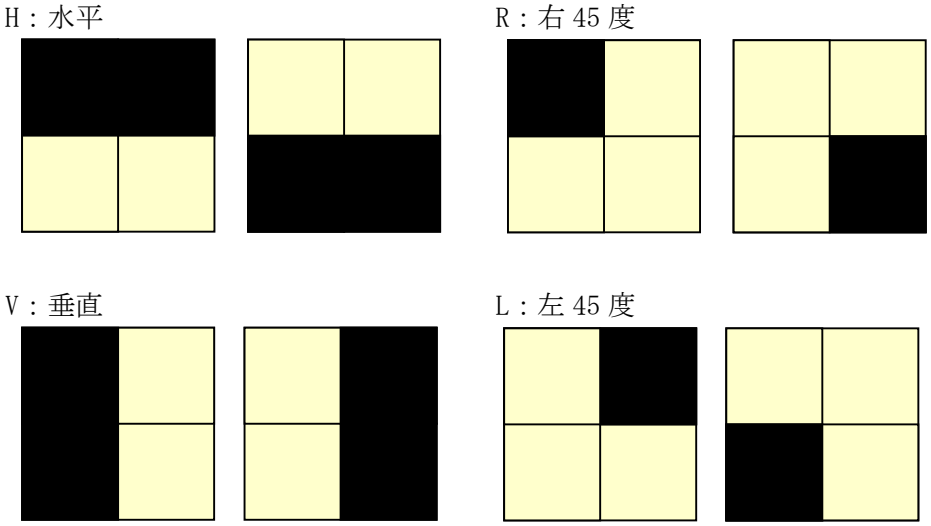
4.2　輪郭方向分布特徴

手書き数字画像を複数のブロック領域に分割し，各ブロック領域に含まれる黒連結領域の輪郭画素の方向を 4 方向(H：水平，R：右 45 度，V：垂直，L：左 45 度)に量子化して計数して，ブロック毎に 4 方向別の比率を算出する。この黒画素輪郭画素の比率をブロック数分だけ並べたベクトルを輪郭方向分布特徴と呼ぶ。

具体的には，縦 120 画素×横 80 画素の手書き数字画像を 20×20 画素の正方ブロック領域に分割することにより，縦 6×横 4 の総数 24 個のブロック領域を生成する。各ブロックの黒画素輪郭の 4 方向別の比率を並べた特徴ベクトルの次元数は 6×4×4＝96 となる。

黒輪郭画素の 4 方向量子化には，下図に示す 2×2 マスクを用いた判定処理を採用する。

図中，2×2 マスクの左上の画素が注目画素である。注目画素が下図の 8 種類のいずれかのパターンを満足する場合に，当該注目画素がそれぞれ指定された方向を持つ画素であると判定する。



輪郭方向分布特徴 fv_dir[24][4]の抽出処理は次のようなコーディングになる。

```
#define X_BLOCK 20
#define Y_BLOCK 20
#define BLOCKS ((WID/X_BLOCK)*(HEI/Y_BLOCK))
#define DIRECTION 4
…
/* prototype declaration of functions */
int get_dircode(int, int);
…
int main() {
  …
  int x, y, k;
  int code;
  double fv_dir[BLOCKS][DIRECTION];
  …
  for (k = 0; k < BLOCKS; k++) {
    for (code = 0; code < DIRECTION; code++) {
      fv_dir[k][code] = 0.0;
    }
  }
  for (y = 0; y < y_size2-1; y++) {
    for (x = 0; x < x_size2-1; x++) {
      code = get_dircode(y,x);
      if (code != -1) {
        k = y/Y_BLOCK*(WID/X_BLOCK)+x/X_BLOCK;
        fv_dir[k][code] += 1.0;
      }
    }
  }
  for (k = 0; k < BLOCKS; k++) {
    for (code = 0; code < DIRECTION; code++) {
      fv_dir[k][code] /= (double)X_BLOCK*Y_BLOCK;
    }
```

```
  }
  …
}

/* definition of functions */
int get_dircode(int y, int x)
/* 4-directional quantization of peripheral black pixels */
{
  int peri[4], sum;

  peri[0] = image2[y][x] == BLACK ? 1 : 0;
  peri[1] = image2[y][x+1] == BLACK ? 1 : 0;
  peri[2] = image2[y+1][x] == BLACK ? 1 : 0;
  peri[3] = image1[y+1][x+1] == BLACK ? 1 : 0;
  sum = peri[0] + peri[1] + peri[2] + peri[3];
  if (sum == 2) {
    if (peri[0] * peri[1] == 1 || peri[2] * peri[3] == 1) {
      return 0;  /* Horizontal */
    }
    if (peri[0] * peri[2] == 1 || peri[1] * peri[3] == 1) {
      return 2;  /* Vertical */
    }
    return -1;
  } else if (sum == 1) {
    if (peri[0] == 1 || peri[3] == 1) {
      return 1;  /* Right-diagonal */
    }
    if (peri[1] == 1 || peri[2] == 1) {
      return 3;  /* Left-diagonal */
    }
    return -1;
  } else {
    return -1;
  }
}
```

5　識別器の構成

学習データを用いて特徴空間での各クラスのサンプルの分布を求めて，その統計的ないし確率的振舞いを分析して，テストサンプルに対して誤り確率最小となるクラス名を出力する識別器を構成するのが，ベイズ則に基づく統計的パターン認識の考え方の基礎である。

統計的パターン認識の考え方の基礎については下記 URL：

https://waka.cis.k.hosei.ac.jp/patrec_master_1.pdf

を参照のこと。

学習サンプル数が十分にある場合，経験則としてはクラス当たりのサンプル数が特徴次元数の 2 桁以上ある場合，上記の統計的パターン認識に基づく識別器は極めて有力である。近年良く利用されるのが，次の 3 種類の識別器である。

[1] 各クラスの特徴空間での分布が正規分布であると仮定して，ベイズ則に基づく最適 2 次識別関数から導かれる擬似ベイズ識別関数

[2] 入力層に特徴ベクトルの成分を並べて出力層に各クラスの尤度を並べる feedforward 型の多層ネットワークで，中間層ユニットの活性化関数に sigmoid 関数を用いて非線形な決定境界を誤差逆伝播法で学習する 2 層ニューラルネット

[3] カーネル関数を用いて元の特徴空間から高次元（しばしば無限大）の特徴空間に写像して，その高次元空間でマージン最大となる線形な決定境界を凸 2 次計画法で決定するサポートベクターマシン

ここでは，次の 3 種類の識別器を取り上げる。

1 つ目は，最も素朴な識別器であり，テストサンプルと各クラスの平均ベクトルとの距離を調べて，最小距離値を持つクラスに分類する「最近傍平均分類」（Nearest-mean classification）である。

2 つ目は，学習サンプルの特徴ベクトルをすべて登録しておき，テストサンプルとの距離の小さい順に k 個の学習サンプルを選択し，それら k 個の中でのクラスの多数決を行う「k-NN 分類」（k-nearest-neighbor classification）である。

3 つ目は，上記[1]の擬似ベイズ識別関数の近似として定式化される「改良投影距離」である。

上記[2]の 2 層ニューラルネットを試みる場合は別途指示する。

また，上記[3]のサポートベクターマシンの実装を試みる場合は次の URL：

http://www.csie.ntu.edu.tw/~cjlin/libsvm/

を参照のこと。

5.1　最近傍平均分類

まず，学習データを用いて，特徴空間での各クラスのサンプルの分布から各クラスの平均ベクトルを算出する。手書き数字認識の場合，カテゴリ 0～9 の分類であり，クラス数は 10

である。特徴空間の次元数を 96 とすると，算出された平均ベクトルは実数の 2 次元配列 fv_mean[10][96]に格納することができる。

次に，テストデータから 1 文字サンプルを入力して，そのテストサンプルから抽出された特徴ベクトルを fv[96]と記す。

続いて,テストサンプルの特徴ベクトル fv[96]と各クラスの平均ベクトル fv_mean[][96]とのユークリッド距離を算出して，小さい順にソートする。最小距離を持つクラスが当該テストサンプルの認識結果となる。

上記の最近傍平均分類の処理は次のようなコーディングになる。fv_mean[10][96], fv[96]の算出部分は省略する。また，平均ベクトルの格納には，2 次元配列を用いずに，double **fv_mean; の宣言により，動的なメモリ確保を利用している。

```
#include <stdlib.h>
#include <math.h>
…
#define N_CLASSES 10
#define DIM 96
…
/* prototype declaration of functions */
double calc_Euc_dist(double *, double *, int);
void bsort_ascend(double *, int *, int);
…
int main() {
  …
  int k;
  double fv[DIM];
  double **fv_mean;
  double dist[N_CLASSES];
  int label[N_CLASSES];
  …
  /* dynamic memory allocation */
  fv_mean = (double **)malloc(sizeof(double *) * N_CLASSES);
  for (k = 0; k < N_CLASSES; k++) {
    fv_mean[k] = (double *)malloc(sizeof(double) * DIM);
  }
  …
  /* calculation of Euclidean distances between input and mean feature vectors */
```

```
  for (k = 0; k < N_CLASSES; k++) {
    dist[k] = calc_Euc_dist(fv, fv_mean[k], DIM);
  }
  /* sorting of distances in the ascending order */
  for (k = 0; k < N_CLASSES; k++) {
    label[k] = k;
  }
  bsort_ascend(dist, label, N_CLASSES);

  ...
  /* release of memory allocation */
  for (k = 0; k < N_CLASSES; k++) {
    free(fv_mean[k]);
  }
  free(fv_mean);
  ...
}

/* definition of functions */
double calc_Euc_dist(double *fv1, double *fv2, int dim)
/* calculation of Euclidean distance between fv1 and fv2 */
{
  int k;
  double dist;

  dist = 0.0;
  for (k = 0; k < dim; k++) {
    dist += (fv1[k]-fv2[k])*(fv1[k]-fv2[k]);
  }
  return sqrt(dist);
}

void bsort_ascend(double *array, int *label, int dim)
/* bubble sorting in the ascending order */
{
  int i, j, k;
```

```
  double work;

  for (i = 0; i < dim-1; i++) {
    for (j = i + 1; j < dim; j++) {
      if (array[i] > array[j]) {
        work = array[i];
        array[i] = array[j];
        array[j] = work;
        k = label[i];
        label[i] = label[j];
        label[j] = k;
      }
    }
  }
}
```

5.2 k-NN 分類

k-NN 分類についての概説は下記 URL を参照のこと。

http://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

k-NN 分類の処理の流れは次のようになる。

① 全学習サンプルをテンプレートとして登録する。ここでは，学習データの全特徴ベクトルを登録することに相当する。それらを集合{ lfv[DIM] }と記す。

② テストデータから認識すべきサンプルを一つずつ順に入力する。テストサンプルから抽出された特徴ベクトルを tfv[DIM]とする。

③ テストサンプルの特徴ベクトル tfv[DIM]と集合{ lfv[DIM] }内の個々のテンプレートとのパターン間距離を算出して，値の小さい順にソートし，上位 k 個のテンプレートを選択する。

④ それら k 個のテンプレートでクラスの多数決を行い，最大個数のテンプレートを含むクラスを当該テストサンプルの認識結果として出力する。

上記の k-NN 分類の処理は次のようなコーディングになる。1 つ 1 つのテストサンプルの認識毎に，全テンプレートの読み込みが必要となる。また，上記③のソーティングでは，実際は，上位 k 個のみ求めればよい。

```
#include <stdlib.h>
```

```
#include <math.h>
…
#define N_CLASSES 10
#define DIM 96
#define MAX_CANDIDATES 20000
#define KNN 3
…
/* prototype declaration of functions */
double calc_Euc_dist(double *, double *, int);
void quicksort(double *, int *, int, int);
void dswap(double *, int, int);
void iswap(int *, int, int);

…
int main() {
   FILE *fp;
   char filename[256];
   int i, j, k;
   double lfv[DIM];
   double tfv[DIM];
   int l_digit, l_serial;
   int l_samples;
   int t_digit, t_serial;
   int hit[N_CLASSES];
   double *dist;
   int **sample_id;
   int *label;
   …
   static int l_pat[10] = {2535, 1691, 1658, 1736, 1188, 1996, 1977, 1410, 2366, 1428};
   static int t_pat[10] = {1485, 2392, 1939, 2043, 2403, 1641, 1831, 2056, 986, 1140};
   …
   /* dynamic memory allocation */
   dist = (double *)malloc(sizeof(double) * MAX_CANDIDATES);
   label = (int *)malloc(sizeof(int) * MAX_CANDIDATES);
   sample_id = (int **)malloc(sizeof(int *) * 2);
   for (k = 0; k < 2; k++) {
```

```
    sample_id[k] = (int *)malloc(sizeof(int) * MAX_CANDIDATES);
}
…
/* input of one test feature vector */
while (1) {
    do {
        printf("¥ndigit (exit = -1) = ");
        scanf("%d", &t_digit);
        if (t_digit == -1 || t_digit < 0 || t_digit >= 10) exit(1);
        printf("serial = ");
        scanf("%d", &t_serial);
    } while (t_serial < 1 || t_serial > t_pat[t_digit]);
    sprintf(filename, "./fv_mesh/tfv%d_%04d", t_digit, t_serial);
    fp = fopen(filename, "rb");
    fread(tfv, sizeof(tfv), 1, fp);
    fclose(fp);
    /* k-NN classification starts */
    l_samples = 0;
    /* input of all learning feature vectors one by one */
    for (l_digit = 0; l_digit < N_CLASSES; l_digit++) {
        for (l_serial = 1; l_serial <= l_pat[l_digit]; l_serial++) {
            sample_id[0][l_samples] = l_digit;
            sample_id[1][l_samples] = l_serial;
            /* input of one learning feature vector */
            …
            /* calculation of Euclidean distances between two feature vectors */
            dist[l_samples] = calc_Euc_dist(tfv, lfv, DIM);
            l_samples++;
        }
    }
    /* sorting of distances in the ascending order */
    for (k = 0; k < l_samples; k++) {
        label[k] = k;
    }
    quicksort(dist, label, 0, l_samples-1);
```

```
      /* counting of class occurrences in KNN candidates */
      for (k = 0; k < N_CLASSES; k++) {
         hit[k] = 0;
      }
      for (k = 0; k < KNN; k++) {
         i = label[k];
         j = sample_id[0][i];
         hit[j]++;
      }
      /* majority voting in hit[ ] */
      …
      /* display of recognition result */
      …
   }
   /* release of memory allocation */
   for (k = 0; k < 2; k++) {
      free(sample_id[k]);
   }
   free(sample_id);
   free(label);
   free(dist);
   …
}

/* definition of functions */
double calc_Euc_dist(double *fv1, double *fv2, int dim)
/* calculation of Euclidean distance between fv1 and fv2 */
{
   int k;
   double dist;

   dist = 0.0;
   for (k = 0; k < dim; k++) {
      dist += (fv1[k]-fv2[k])*(fv1[k]-fv2[k]);
   }
   return sqrt(dist);
```

```
}


void quicksort(double *array, int *label, int left, int right)
{
   int i, last;

   if (left >= right) return;
   dswap(array, left, (left+right)/2);
   iswap(label, left, (left+right)/2);
   last = left;
   for (i = left+1; i <= right; i++) {
      if (array[i] < array[left]) {
         ++last;
         dswap(array, last, i);
         iswap(label, last, i);
      }
   }
   dswap(array, left, last);
   iswap(label, left, last);
   quicksort(array, label, left, last-1);
   quicksort(array, label, last+1, right);
}


void dswap(double *array, int i, int j)
{
   double temp;

   temp = array[i];
   array[i] = array[j];
   array[j] = temp;
}


void iswap(int *array, int i, int j)
```

```
{
  int temp;

  temp = array[i];
  array[i] = array[j];
  array[j] = temp;
}
```

5.3　改良投影距離

改良投影距離の導出と理論的考察については次の文献に詳しい。

・韓雪仙，若林哲史，木村文隆，三宅康二：“ベイズアプローチによる最適識別系の有限標本効果に関する考察－学習標本の大きさがクラス間で異なる場合－”，信学論 D-II，Vol. J82-D-II，No. 4，pp. 621-630，1999.

改良投影距離の計算式を以下に掲げる。

入力サンプルの特徴ベクトルを$\boldsymbol{x}$（次元数d）と記し，クラスωとの改良投影距離を$g(\boldsymbol{x}|\omega)$と記す。また，クラスωの平均ベクトルを$\boldsymbol{\mu}_\omega$，共分散行列をΣ_ω，共分散行列Σ_ωのd個の固有値と固有ベクトルの組を$\{\lambda_{i,\omega}, \Phi_{i,\omega}\}(i = 1,2,\dots,d)$と記す。$\sigma^2$は$d$個の特徴要素の全クラスでの平均分散値である。$\alpha$は実数で$\alpha \in [0,1)$の範囲のパラメータであり，認識実験より識別率が最大となるように調整する。Tはベクトルの転置操作を表す。

$$g(\boldsymbol{x}|\omega) = \| \boldsymbol{x} - \boldsymbol{\mu}_\omega \|^2 - \sum_{i=1}^{k} \frac{\lambda_{i,\omega}}{\lambda_{i,\omega} + \frac{\alpha}{1-\alpha}\sigma^2} \left\{ {\Phi_{i,\omega}}^{\mathrm{T}} (\boldsymbol{x} - \boldsymbol{\mu}_\omega) \right\}^2. \tag{1}$$

さらに，kは切断次元数と呼ばれ，クラスω毎に定まり，次式で固有値の累積寄与率$\rho_{k,\omega}$が一定値βを越える最小値として決定される。βの値としては，0.90や0.95がよく用いられる。

$$\rho_{k,\omega} = \frac{\sum_{i=1}^{k} \lambda_{i,\omega}}{\sum_{j=1}^{d} \lambda_{j,\omega}} \geq \beta.$$

改良投影距離の式(1)で，第2項を除くと，最近傍平均分類と同等となる。

さて，改良投影距離を実装するには，まず，共分散行列の固有値・固有ベクトルの計算が必要である。これには線形計算ライブラリ CLAPACK を用いると良い。

インストールの手順を以下に示す。

まず，下記 URL：

http://www.netlib.org/clapack/

から

clapack.tgz

をホームディレクトリにダウンロードする。2011.9時点で最新バージョンは3.2.1である。

続いて，次のコマンドシーケンスを実行する。

```
$ tar zxvf clapack.tgz
$ cd CLAPACK-3.2.1
$ cp make.inc.example make.inc
$ make f2clib
$ make blaslib
$ make
```

さらに，ルート権限になって

```
$ sudo cp lapack_LINUX.a /usr/local/lib/liblapack.a
$ sudo cp blas_LINUX.a /usr/local/lib/libblas.a
$ sudo cp F2CLIBS/libf2c.a /usr/local/lib
$ sudo /sbin/ldconfig
```

これで線形計算ライブラリ CLAPACK が使えるようになる。

参考に，固有値・固有ベクトルを計算するサンプルプログラム eigen.c を以下に示す。

```
/* eigen.c */
#include <stdio.h>

#define N 3

double A[N*N];
double w[N];
double work[3*N];

int main()
{
  int i;
  char jobz = 'V';
  char uplo ='U';
```

```
  int n = N, lda = N, lwork = 3*N, info;

  A[0]=9.;A[1]=999.;A[2]=999.;  /* 999が入っている配列の値は利用されない□*/
  A[3]=1.;A[4]=9.;A[5]=999.;
  A[6]=1.;A[7]=1.;A[8]=9.;

  dsyev_(&jobz, &uplo, &n, A, &lda, w, work, &lwork, &info);

  for (i = 0; i < N; i++) {
    printf("%lf : vector=[ %lf %lf %lf ]¥n",w[i] ,A[i*N], A[i*N+1], A[i*N+2]);
  }

  return 0;
}
```

CLAPACK を利用したプログラムのコンパイル・リンクは次のようになる。

```
$ gcc -o eigen eigen.c -llapack -lblas -lf2c -lm↵
```

実行は次の通りである。

```
$ ./eigen↵
```

最後に，式(1)の改良投影距離の計算のコーディングのしかたを示す。ただし，平均ベクトル fv_mean[10][96]，共分散行列 fv_cov[10][96][96]，固有値 eigval[10][96]，固有ベクトル eigvec[10][96][96]，切断次元数 cutdim[10]の算出は完了しているとして，外部変数として宣言してある。

```
#include <stdlib.h>
#include <math.h>
…
#define N_CLASSES 10
#define DIM 96
#define ALPHA 0.5
…
/* global variables */
double fv_mean[N_CLASSES][DIM], fv_cov[N_CLASSES][DIM][DIM];
double eigval[N_CLASSES][DIM], eigvec[N_CLASSES][DIM][DIM];
int cutdim[N_CLASSES];
```

```
…
/* prototype declaration of functions */
double calc_projection_dist(double *, int, double, double, int);
…
int main() {
  …
  int i, j, k;
  double fv[DIM];
  double sigma2;
  double dist[N_CLASSES];
  int label[N_CLASSES];
  …
  /* calculation of average variance of feature vector elements */
  sigma2 = 0.0;
  for (k = 0; k < N_CLASSES; k++) {
    for (i = 0; i < DIM; i++) {
      sigma2 += fv_cov[k][i][i];
    }
  }
  sigma2 /= (double)N_CLASSES*DIM;

  /* calculation of modified projection distances */
  for (k = 0; k < N_CLASSES; k++) {
    dist[k] = calc_projection_dist(fv, k, sigma2, ALPHA, DIM);
  }
  /* sorting of distances in the ascending order */
  for (k = 0; k < N_CLASSES; k++) {
    label[k] = k;
  }
  bsort_ascend(dist, label, N_CLASSES);

  …
}

/* definition of functions */
double calc_projection_dist(double *fv, int k, double sigma2,
```

```
                                double alpha, int dim)
{
  int i, j;
  double dist1, dist2;
  double sum;
  double factor;

  /* calculation of the first term */
  dist1 = 0.0;
  for (i = 0; i < dim; i++) {
    dist1 += (fv[i]-fv_mean[k][i])*(fv[i]-fv_mean[k][i]);
  }

  /* calculation of the second term */
  dist2 = 0.0;
  factor = alpha/(1.0-alpha)*sigma2;
  for (j = 0; j < cutdim[k]; j++) {
    sum = 0.0;
    for (i = 0; i < DIM; i++) {
      sum += eigvec[k][j][i]*(fv[i]-fv_mean[k][i]);
    }
    sum = sum*sum;
    dist2 += eigval[k][j]/(eigval[k][j]+factor)*sum;
  }

  return (dist1-dist2);
}
```

6　認識実験

認識実験では，学習→テストの順に実験を進める。

学習では，学習用データ ldg0_0001.pgm～ldg9_1428.pgm を用いて前処理・特徴抽出と識別器の構築を行う。続いて，テストでは，テスト用データ tdg0_0001.pgm～tdg9_1140.pgm を 1 つずつ前処理・特徴抽出してから識別器で分類して 1 位から 10 位までの候補クラス名を出力する。

実験結果の整理では，次のような項目を評価する。

[1] 認識率：正解が第 1 位候補となった率

[2] 累積分類率：正解が上位第 k 位候補までに含まれた率（第 1 位累積分類率＝正解率）

[3] Confusion matrix（混合行列）：行方向に入力サンプルのクラス名，列方向に出力の第 1 候補クラス名を記し，各セルに出現サンプル数を挿入したもの

ただし，テスト用データのみでなく，学習用データについても識別器で分類を行って，学習用データとテスト用データでの認識精度の比較を行うことが多い。一般に，後者の方が認識精度は低くなる。

以下では，認識実験を進める上での基本的な処理のいくつかについてコーディングのしかたを示す。これらを参考に認識実験のプログラムを完成すること。

【手書き数字画像ファイルの連続読み込み】

個々のファイルの読み込みではファイル名を指定する必要がある。これをキーボードから入力するのでなく，プログラミングでファイル名を指定するには次のように行う。ただし，学習用データおよびテスト用データに含まれる各数字のサンプル数を予め配列に格納しておく（cdrom1b.txt 参照のこと）。

```
#include <stdio.h>
#include "mypgm.h"
…
/* global variables */
l_pat[10] = {2535, 1691, 1658, 1736, 1188, 1996, 1977, 1410, 2366, 1428};
t_pat[10] = {1485, 2392, 1939, 2043, 2403, 1641, 1831, 2056, 986, 1140};
…
int main() {
  …
  char filename[256];
  int digit, serial;
```

```
  …
  for (digit = 0; digit <= 9; digit++) {
    for (serial = 1; serial <= l_pat[digit]; serial++) {
      sprintf(filename, "./cdrom1b/ldg%d_%04d.pgm", digit, serial);
      /* input of learning sample data */
      load_image_file(filename);
      …
    }
  }
  …
}
```

【抽出特徴のバイナリデータのファイル出力】

識別器の構築や認識実験を行うには，学習用およびテスト用の各サンプルから抽出された特徴データを何度も使うことになる。これを毎回計算するのは無駄であるため，抽出特徴をバイナリデータとしてファイル出力しておくと良い。これには，<stdio.h>で定義されている直接入出力関数 fread, fwrite を用いる。

次のようなコーディングになる。予め特徴データ用ディレクトリ fv_data を作成しておき，そこに出力することにする。

```
#include <stdio.h>
#include "mypgm.h"
…
#define DIM 96
…
/* global variables */
l_pat[10] = {2535, 1691, 1658, 1736, 1188, 1996, 1977, 1410, 2366, 1428};
t_pat[10] = {1485, 2392, 1939, 2043, 2403, 1641, 1831, 2056, 986, 1140};
…
int main() {
  …
  FILE *fp;
  char infile[256], outfile[256];
  int digit, serial;
  double fv[DIM];
  …
```

```
  for (digit = 0; digit <= 9; digit++) {
    for (serial = 1; serial <= l_pat[digit]; serial++) {
      sprintf(infile, "./cdrom1b/ldg%d_%04d.pgm", digit, serial);
      /* input of learning sample data */
      load_image_file(infile);
      …
      /* position and size normalization */
      …
      /* feature extraction */
      …
      /* output of binary feature data */
      sprintf(outfile, "./fv_data/lfv%d_%04d.pgm", digit, serial);
      fp = fopen(outfile, "wb");
      fwrite(fv, sizeof(fv), 1, fp);
      fclose(fp);
      …
    }
  }
  …
}
```

※特徴データファイルの読み込みは次のように行う。

```
      /* input of binary feature data */
      sprintf(infile, "./fv_data/lfv%d_%04d.pgm", digit, serial);
      fp = fopen(infile, "rb");
      fread(fv, sizeof(fv), 1, fp);
      fclose(fp);
```

【累積分類率の計算】

テストでは，テスト用データ tdg0_0001.pgm～tdg9_1140.pgm を 1 つずつ前処理・特徴抽出してから識別器で分類して 1 位から 10 位までの候補クラス名を出力する。

その際，正解が第何位に出現したかを調べて集計すると，累積分類率を評価することができる。

次のようなコーディングになる。ここでは，テスト用データの特徴データファイルを読み込んで認識実験を行う場合について示す。

```
#include <stdio.h>
#include "mypgm.h"
…
#define N_CLASSES 10
#define DIM 96
…
/* global variables */
l_pat[10] = {2535, 1691, 1658, 1736, 1188, 1996, 1977, 1410, 2366, 1428};
t_pat[10] = {1485, 2392, 1939, 2043, 2403, 1641, 1831, 2056, 986, 1140};
…
int main() {
  …
  FILE *fp;
  char filename[256];
  int digit, serial;
  int t_total;
  int k;
  int rank;
  double fv[DIM];
  double dist[N_CLASSES];
  int label[N_CLASSES];
  double cum_rate[N_CLASSES+1][N_CLASSES] = {0.0};
  …
  t_total = 0;
  for (digit = 0; digit <= 9; digit++) {
    t_total += t_pat[digit];
    for (serial = 1; serial <= t_pat[digit]; serial++) {
      /* input of binary feature data */
      sprintf(filename, "./fv_data/tfv%d_%04d.pgm", digit, serial);
      fp = fopen(filename, "rb");
      fread(fv, sizeof(fv), 1, fp);
      fclose(fp);
      …
      /* calculation of distances */
      …
      /* sorting of distances in the ascending order */
```

```
      for (k = 0; k < N_CLASSES; k++) {
        label[k] = k;
      }
      bsort_ascend(dist, label, N_CLASSES);
      /* check where the correct answer appears */
      for (k = 0; k < N_CLASSES; k++) {
        if (label[k] == digit) {
          rank = k;
          break;
        }
      }
      cum_rate[digit][rank] += 1.0;
      cum_rate[N_CLASSES][rank] += 1.0;
    }
    /* calculation of cumulative classfication rates for each digit */
    for (k = 0; k < N_CLASSES; k++) {
      cum_rate[digit][k] *= 100.0/t_pat[digit];
    }
    for (k = 1; k < N_CLASSES; k++) {
      cum_rate[digit][k] += cum_rate[digit][k-1];
    }
  }
  /* calculation of cumulative classfication rates for all digits */
  for (k = 0; k < N_CLASSES; k++) {
      cum_rate[N_CLASSES][k] *= 100.0/t_total;
  }
  for (k = 1; k < N_CLASSES; k++) {
    cum_rate[N_CLASSES][k] += cum_rate[N_CLASSES][k-1];
  }
  …
}
```

完