November 11, 2009 at 14:20

1. Introduction. In this program, a probably correct k-NN (shortly, PCNN) algorithm is implemented. A PCNN algorithm outputs the correct kNNs in high probability. Formally, the algorithm attains the "PCNN" criterion

$$P\left(D(X,\hat{X}_{kNN}) = D(X, X_{kNN})\right) > 1 - \epsilon$$

for a small ϵ, where $\hat{X}_{kNN}$ is the kth nearest neighbor of X returned by this algorithm and X_{kNN} is the correct answer.

In the algorithm implemented here, principle component analysis is applied to data first to obtain the largest variations in a few dimensions. Then it carries out a probabilistic variant of partial distance strategy, which is called a "marginal distance strategy (MDS)." The distance calculation of a query point q and one stored point x stops when $D_l^2(x,q) > \theta_l$ is satisfied for some l, judging x as not being the candidate. Here, $D_l^2(\cdot,\cdot)$ is the squared Euclidean distance measured in the first l dimensions. Let q be represented by a random variable X according to a density function f. In addition, we assume any other points are also generated from f. The key idea of the algorithm is, for a specified "error" ϵ, to find θ_l such that

$$P(D_l^2(X, X_{kNN}) > \theta_l) \leq \epsilon.$$

Then the partial distance judgement is carried out with θ_l for any point Y to be compared at lth dimension, that is, if $D_l^2(X,Y) > \theta_l$ holds, then the remaining distance calculation for Y up to m (the original dimension) is discarded. The probability that a point Y is calculated up to m dimensions is given by

$$\delta = P(D_l^2(X,Y) < \theta_l).$$

In other words, only at ratio of δ, full distance calculation is carried out.

To estimate these probabilities, we use the empirical percentile approach. First we estimate these probability distributions $F(D_l^2(X, X_{kNN}))$ and $G(D_l^2(X,Y))$ from a small subset of the stored data points for $l = 1, 2, \ldots, l_{max}$. Then according to given ϵ we find θ_l such that $\hat{F}(\theta_l) \simeq 1 - \epsilon$ for every l. In addition, assuming that the computational costs of inner product and (squared) distance are known, we find the optimal l_{opt} automatically.

For the detail, see the paper:

J. Toyama, M. Kudo and H. Imai, "Probably Correct k-Nearest Neighbor Search in High Dimensions", *Pattern Recognition*, in press, (DOI information: 10.1016/j.patcog.2009.09.026).

You can contact us at mine@main.ist.hokudai.ac.jp (Mineichi Kudo) and are welcome to visit our site http://prml.main.ist.hokudai.ac.jp (PRML laboratory).

2. Updated contents in this version. In the version (1.3), we also implemented a narrowing procedure to speed up the searching phase. The idea is that we execute our marginal distance strategy at the first dimension. In the first dimension, we find an interval around the query point as $[q_1 - \theta_1, q_1 + \theta_1]$. Then it suffices to check the samples falling in this interval. Such a narrowing is carried out very efficiently by a hash table or a binary search. Instead, we have to accept an additional error ϵ_1. The total error ϵ is added by this ϵ_1, so only a small value is allowed to ϵ_1. In this program, $\epsilon_1 = 0.1\%$.

3. Usage of this program. The program *kNN_MDS* can be executed in two ways:

A. (single path: design + search)
 % *kNN_MDS file1* (*training*) *file2* (*testing*) [*options*]

B. (sequential path: design and search) 1) a model generation process and 2) a searching process as follows.
 1. % *kNN_MDS file1* (*training*) *file2* (*testing*) −*O modelfile* [*options*]
 2. % *kNN_MDS file1* (*training*) *file2* (*testing*) −*I modelfile* [*options*]

In the first step, a model is generated from data of *file1* to calculate principal vectors and to find the empirical distributions. Then, in the second step, the model is read from *modelfile*, and find k-nearest neighbors of each data of *file2*.

4. Options. (Type the command without any option such as % *kNN_MDS* to see the actually implemented options.)
Options are

1. $-k\ k$: the number of nearest neighbors. The default value is one.
2. $-c\ c$: the number of classes. The default value is one, that is, no class label is assumed to be given.
3. $-tr\ n_1, n_2, \cdots, n_c$: the numbers of training samples in the class order. By default, the total number n (automatically determined from *file1*) is equally divided into n/c.
4. $-te\ m_1, m_2, \cdots, m_c$: the numbers of testing samples in the class order. By default, the total number m (automatically determined from *file2*) is equally divided into n/c.
5. $-A\ a$: the integer specifying the value of ϵ. In order, $a = 0, 1, 2, 3, 4$ and 5 mean $\epsilon = 0.0, 0.001, 0.005, 0.01, 0.05$ and 0.1, respectively. If you specify $a < 0$ then you will be prompted to select one of possible as after displaying the detail information.
6. $-Q\ q$: the number of maximum marginal dimension l_{max}. The default value is ten. This parameter has to be chosen appropriately so as to $l_{opt} < l_{max}$.
7. $-(N)T$: (non)activate the terminal condition. The default is inactive.
8. $-(N)R$: (non)activate the recovery process. The default is inactive.
9. $-(N)P$: (non)activate the partial distance strategy. The default is active.
10. $-L\ l$: specify the value of l manually. Unless this option is given, the optimal value l_{opt} is estimated from data.
11. $-S\ s$: the number of samples used for estimating the empirical distributions. The default value is 1000.
12. $-F\ f$: the index of narrowing procedure at 1st dimension. You can choose one of tree options. In default, a table lookup will be carried out. $f = 0, 1, 2, 3$ mean "no narrowing","filtering","binary search" and "table lookup", respectively

5. Output analysis. This program outputs 1) several setting environments, 2) estimated reduction rates with thresholds according to $l = 1, 2, \ldots, l_{max}$, 3) kth nearest neighbor with id, and 4) computational statistics. For 3), you will have something like

OUT	0	0.28356	49
OUT	0	0.11245	1
⋮			

A line above shows in order prefix `OUT`, the guessed class (starting from 0), the distance to the query points, the index of the kth nearest neighbor.

6. Data format. The data format used for *file1* and *file2* is the set of lines of m ASCII values (m-dimensional data) separated by spaces or tabs.

$$\begin{array}{cccc} x_{11} & x_{12} & \cdots & x_{1m} \\ x_{21} & x_{22} & \cdots & x_{2m} \\ \vdots & & & \end{array}$$

The data **have to be ordered** class by class, because the data will be separated by the numbers specified by the option $-tr$ or $-te$ or both.

7. How to compile from C++ files. All necessary C++ files are given in this distribution. In most cases, the following is sufficient to obtain the executable code with a C++ compiler, by default, g++,

% *make clean*

% *make*

8. How to compile from the source file. Almost all C++ files are generated from `CWEB` files with extension *.w* including this file. The `CWEB` program is one of literature programmings (developed by Silvio Levy and Donald E. Knuth) and allows the user to make a document in which source C or C++ code is embedded in the text describing the algorithm. A `CWEB` file is converted to a TeX file for documentation and a C++ code for execution. The TeX file is obtained by `CWEAVE` command such as

% *cweave kNN_MDS*

and the C++ file is obtained by `CTANGLE` command such as

% *ctangle kNN_MDS*

To do this, you need to download cweb-3.6 or its later versions.

Once you had the C++ code *kNN_MDS.C* or *kNN_MDS.c*, you can type

% *make kNN_MDS*

9. Simple test. This package includes two sample datasets. Try

% *kNN_MDS tr te*

and observe the output. To confirm if everything is perfectly made or not, compare the output with given *result*. The files *tr_1000* and *te_1000* are another dataset.

10. The structure of program. The C++ code is very simple.

In the following, blocks enclosed by `#ifdef DEBUG` and `#endif` are only activated when you compile this file with `-DDEBUG` (See `Makefile`).

⟨ Header Files 41 ⟩
⟨ Global Constants Declaration 40 ⟩
⟨ Function Declaration 33 ⟩
⟨ main 11 ⟩

11. The main function. The main function of this program is as follows.

```
⟨ main 11 ⟩ ≡
  int main(int argc, char **argv)
  {
    cout << "--(1)--␣Environmental␣setting␣" << endl;
    ⟨ Analysis of arguments 37 ⟩;
        /* ------------------------------------------------------ */
        /* Basic dimensions read from input data and options */
        /* n=n_tr : the number of stored samples */
        /* m : the number of features, dimensionality */
        /* nc : the number of classes */
        /* ------------------------------------------------------ */
    CELL *pair = Λ;
    int *iri;
    if (model_in_flag) {
      cout << "--(1.1a)--␣Reading␣of␣sorted␣data␣index␣" << endl;
      ⟨ Input of sorted data from a file 85 ⟩
      iri = new int[n];
      for (int i = 0; i < n; i++) iri[i] = pair[i].rid;
    }
    cout << "--(1.1)--␣Reading␣of␣the␣training␣data␣" << endl;
    ⟨ Read data for training 35 ⟩;
    n_tr = n;
    if (¬tr_flag) ⟨ Equal sample size is assumed in common to classes for training data 16 ⟩;
    cout << "--(1.2)--␣Find␣the␣empirical␣dist." << endl;
    ⟨ Find the empirical distributions F(X, X_kNN) and G(X, Y) 15 ⟩;
    if (¬model_in_flag)
      ⟨ Memory release of no more necessary variavles 78 ⟩;
    if (¬model_in_flag)
    {
      iri = new int[n];
      for (int i = 0; i < n; i++) iri[i] = i;
    }
    cout << "--(1.3)--␣Reading␣of␣a␣model␣file." << endl;
    if (model_in_flag)
      ⟨ Open a model file 24 ⟩;
    if (pair ≡ Λ) pair = new CELL[n];
    if (model_out_flag)
    {
      ⟨ Output of the model to a file 25 ⟩;
      ⟨ Output of sorted data to a file 84 ⟩;
      exit(0);
    }
          /* ------------------------------------------------------ */
          /* q : the marginal dimension */
          /* theta_q : the threshold value at qth dimension specified by ϵ */
          /* ------------------------------------------------------ */
    cout << "--(2)--␣Estimated␣percentile␣information␣" << endl;
    ⟨ Selection of q and θ_q 68 ⟩
    cout << "--(3)--␣Sorting␣on␣the␣1st␣principal␣axis␣" << endl;
    if (¬model_in_flag) ⟨ Sorting of data in the first principal axis 79 ⟩
```

```
int *start_data_no;
if (¬model_in_flag)
  ⟨ Data sorting on the basis of 1st principal dimension 81 ⟩;
int B = B_default;
REAL W;
REAL w;
REAL sample_min;
REAL sample_max;
REAL th1st;
int iw;
if (narrow ≡ Table_Lookup) ⟨ Establish a data structure for a serach of O(1) 82 ⟩
      /* ------------------------------------------------------------------ */
      /* n_te : the number of query (testing) samples */
      /* x : a query sample */
      /* knn_id : the indecies of K-NNs */
      /* knn_dis : the distances of K-NNs */
      /* ------------------------------------------------------------------ */
⟨ Read data for testing 36 ⟩;    /* Store into memory and let the number of samples be n_te */
if (¬te_flag) ⟨ Equal sample size is assumed in common to classes for testing data 17 ⟩;
if (n_te ≡ 0) {
  fprintf(stdout, "n_te␣=␣0\n");
  exit(0);
}
⟨ Memory setting for a query sample and K-NNs 14 ⟩;
time_t stime, etime;
printf("Start-time␣%u\n", stime = (unsigned) time(Λ));
cout ≪ "--(4)--␣kNN's␣(OUT␣class␣distance␣id)␣" ≪ endl;
for (int i = 0; i < n_te; i++)    /* Process all testing samples */
{
  isample = i;
  cl = class_from_num(i, te_pc, nc);
  ⟨ Get a test sample 42 ⟩;    /* Let the query sample x */
  ⟨ Search for a query sample x 12 ⟩    /* Get the K-NN and the guess class */
  if (cl ≡ guess_cl)    /* For classification */
  ncorrect++;
  else nwrong++;
}
cout ≪ "Complete␣!" ≪ endl;
if (nc > 1) cout ≪ "Recognition␣Rate␣" ≪ REAL(ncorrect)/REAL(n_te) ≪ endl;
cout ≪ "--(5)--␣k-NN's␣" ≪ endl;
⟨ Output computational infomation 13 ⟩;
printf("End-time␣%u\n", etime = (unsigned) time(Λ));
printf("...␣Used-time␣%u\n", etime − stime);
}
```

This code is used in section 10.

12. Search algorithm. In the following a query sample x is processed.

In the following procedure, the distance calculation of a query point x and any stored point $x_i (i = 1, 2, \ldots, n)$ stops when $D_l^2(x, x_i) > \theta_l$ is satisfied for predetermined l, judging x_i as not being the candidate.

⟨ Search for a query sample x 12 ⟩ ≡

```
{
    /* ------------------------------------------------------------------ */
    /* In the following steps, */
    /* term_flag=1 shows that we can break the distance calculation of ith sample. */
    /* term_flag=2 shows that we can break the process of this x. */
    /* r2 : sqaured distance between x and ith sample (D_l^2(x, i)). */
    /* q : marginal dimension (l) */
    /* theta_q : marginal threshold in lth dimension (θ_l) */
    /* px[ ] : projected q-dimensional x (x^p) */
    /* ptrx[i][ ] : projected ith q-dimensional sample (x_i^p) */
    /* x[ ] : raw m-dimensional query sample (x) */
    /* trx[i][ ] : raw ith m-dimensional training sample (x_i) */
    /* current_d2 : squared distance of current kth NN to a query x (D^2(x, x̂_kNN)) */
    /* ------------------------------------------------------------------ */
  ⟨ Initialize of searching 43 ⟩
  static int int_st, int_ed;
  static REAL th1st, q1st, qst, qed;
  static REAL pp;
  if (narrow ≡ Binary_Search) ⟨ Determine the interval on the first principal axis 80 ⟩
  else if (narrow ≡ Table_Lookup) ⟨ Determine a small subset including the query point 83 ⟩
  else    /* Nothing or Filtering */
  {
    int_st = 0;
    int_ed = n − 1;
    th1st = sqrt(thresh_1st);
    q1st = px[0];
    qst = q1st − th1st;
    qed = q1st + th1st;
  }
#ifdef DEBUG
  cout ≪ "Interval␣on␣1st␣dim␣:␣for␣q_0=␣" ≪ px[0] ≪ "␣and␣th=␣" ≪ thresh_1st ≪
      "␣sorted_id␣" ≪ int_st ≪ "␣:␣" ≪ int_ed ≪ "␣value␣" ≪ pair[int_st].val ≪ "␣:␣" ≪
      pair[int_ed].val ≪ endl;
#endif
  ⟨ Counter of 1-dim marginal distance strategy 74 ⟩;
  int i;
  for (int i = int_st; i ≤ int_ed; i++)    /* Examine ith data */
  {
    term_flag = 0;    /* initialization */
    r2 = 0.0;
    if (q > 0)
        /* — MDS (marginal distance strategy) — */
    {
      REAL d;
      for (int j = 0; j < q; j++) {
        d = px[j] − ptrx[i][j];
        r2 += d ∗ d;
```

```
    }
    if (r2 > theta_q) {
      term_flag = 1;
      edim = q - 1;
      ⟨ Counter of marginal distance strategy 73 ⟩;
      continue;
    }
  }
  if (pflag)
      /* — PDS (partial distance strategy) — */
  {
    REAL d;
    if (allpca_flag ≡ 0) {
      r2 = 0.0;    /* restart from zero */
      for (int j = 0; j < m; j++) {
        d = x[j] - trx[i][j];
        r2 += d * d;
        if (r2 > current_d2/((1.0 + eta) * (1.0 + eta))) {
          term_flag = 1;
          edim = j;
          break;
        }
      }
    }
    else {
      for (int j = q; j < m; j++)    /* continue from q */
      {
        d = px[j] - ptrx[i][j];
        r2 += d * d;
        if (r2 > current_d2/((1.0 + eta) * (1.0 + eta))) {
          term_flag = 1;
          edim = j;
          break;
        }
      }
    }
    if (term_flag ≡ 1) {
      ⟨ Counter of partial distance strategy 75 ⟩;
      continue;
    }
    else    /* As a result, full distance calculation is made */
    {
      ⟨ Counter of full distance strategy 76 ⟩;
    }
  }
  else
      /* — Full distance calculation — */
  {
    r2 = DIS2(x, trx[i]);
    ⟨ Counter of full distance strategy 76 ⟩;
  }
  checked_sample++;
```

```
        /* Updating of current solutions */
    if (r2 < current_d2) {
      if (tflag)
          /* — Termination judgement — */
        ⟨ Check of possibility of termination 45 ⟩;
      ⟨ Update of kNNs 44 ⟩;
      if (q > 0 ∧ current_d2 < theta_q)
        theta_q = current_d2;
      if (term_flag ≡ 2) break;
    }
  }
  r2 = stock_k_nn(K, 0.0, 0, knn_id, knn_dis, 1);      /* Get the answer */
  if (checked_sample ≡ 0)      /* No sample was searched */
      /* — Recovery process — */
  {
    printf("*RECOVER*␣with␣no␣examined␣sample␣for␣%d\n", isample);
    ⟨ Recovery process with full distance search 77 ⟩
  }
  else if (rflag ∧ q > 0 ∧ ⟨ the found kth NN is irregularly large 47 ⟩) {
    printf("*␣Erly␣large:␣current_d2␣=%f␣>␣thresh_l[%d]=%f\n", r2, q, thresh_l[q]);
    ⟨ Recovery process with full distance search 77 ⟩
  }
  kNN = knn_id[K − 1];
  kNN = pair[kNN].id;
  if (kNN ≥ 0) {
    if (K ≡ 1) guess_cl = class_from_num(kNN, tr_pc, nc);
    else if (K > 1) {
      int *vote = new int[nc];
      for (int i = 0; i < nc; i++) vote[i] = 0;
      int iNN, gcl;
      for (int i = 0; i < K; i++) {
        iNN = knn_id[i];
        iNN = pair[iNN].id;
        gcl = class_from_num(iNN, tr_pc, nc);
        vote[gcl]++;
      }
      int maxgcl = 0;
      int vv = vote[0];
      for (int i = 1; i < nc; i++) {
        if (vote[i] > vv) {
          maxgcl = i;
          vv = vote[i];
        }
      }
#ifdef DEBUG
      printf("By␣the␣majority␣vote,␣guess_cl=␣%d␣with␣vote␣%d␣of␣%d\n", maxgcl, vote[maxgcl],
          K);
#endif
      guess_cl = maxgcl;
      delete[] vote;
    }
```

```
    if (LP ≡ 2)
      if (K ≡ 1) printf("OUT␣␣%d␣%f␣%d\n", guess_cl, sqrt(r2), kNN);
      else if (K > 0) {
        printf("OUT␣␣%d␣", guess_cl);
        int iNN, gcl;
        REAL iDD;
        for (int i = 0; i < K; i++) {
          iNN = knn_id[i];
          iNN = pair[iNN].id;
          iDD = knn_dis[i];
          gcl = class_from_num(iNN, tr_pc, nc);
          printf(",␣%d␣%f␣%d", gcl, sqrt(iDD), iNN);
        }
        printf("\n");
      }
      else printf("OUT␣␣%d␣%f␣%d\n", guess_cl, r2, kNN);
  }
  else {
    printf("*Error*␣for␣test␣sample␣%d␣:␣checked_sample␣=%d␣cand_d2␣=␣%f\n", isample,
        checked_sample, current_d2);
    for (int i = 0; i < K; i++) printf("knn_id[%d]=%d␣knn_dis[%d]=%f\n", i, knn_id[i], i, knn_dis[i]);
    exit(0);
  }
}
```

This code is used in section 11.

13. Preparation. From now on, we prepare several procedures and functions.

Here computational information is output. The table will be displayed to show how many times several kinds of calculation was made after searching all the query samples.

⟨ Output computational infomation 13 ⟩ ≡

```
  {
#ifdef COUNTER
    for (int i = 0; i < nkind; i++)
      printf("Kind␣%s␣:␣counter␣=␣%8ld␣(%f␣\%)␣for␣%d␣tr␣samples␣␣and␣%d␣te␣samples\n",
          kindname[i], kcounter[i], 100.0 * REAL(kcounter[i])/REAL(n)/REAL(n_te), n, n_te);
    REAL fulldisnum = 0.0;
    for (int i = 0; i < m; i++) {
      printf("Examined␣Dim␣%d␣:␣%u␣(%f␣\%)\n", i, examined_dim[i],
          100.0 * REAL(examined_dim[i])/REAL(n)/REAL(n_te));
      fulldisnum += examined_dim[i] * REAL(i + 1)/REAL(m);
    }
    printf("Corresponding␣number␣of␣full␣distance␣calculations␣=␣%f\n", fulldisnum);
#endif
  }
```

This code is used in section 11.

14. Memory setting for a query sample.

⟨ Memory setting for a query sample and K-NNs 14 ⟩ ≡

```
  REAL *x;     /* a testing data */
  int *knn_id = new int[K];
  REAL *knn_dis = new REAL[K];
  int cc = 0;
  int cl;     /* true class */
  int guess_cl;     /* guess class */
```

This code is used in section 11.

15. Empirical distributions. Two empirical distributions $F(X, X\_kNN)$ and $G(X, Y)$ are obtained. The empirical distribution $F(X, X\_kNN)$ is obtained from a set of sampled X and its kth NNs among all data. On the other hand, $G(X, Y)$ is obtained from all pairs of sampled X and Y.

```
⟨ Find the empirical distributions F(X, X_kNN) and G(X, Y) 15 ⟩ ≡
  if (m ≤ PCAdim ∧ n > m)
  ⟨ Use projected samples in m principal axes 18 ⟩
  ⟨ Assignment of several variables 19 ⟩
  if (¬model_in_flag)      /* When a model is already obtained and read from a file */
  {
    ⟨ Set up of sample matrix X and sampling matrix SX 64 ⟩
    ⟨ Analysis of statistics of training data 26 ⟩
    ⟨ Set up of thresholds 67 ⟩
  }
  ⟨ Parameters for Initializing of total environment 70 ⟩
  if (q > 0 ∨ allpca_flag)
    ⟨ Make ready of eigen vectors ev 61 ⟩
  if (¬model_in_flag) ⟨ Initialize of total environment 63 ⟩
  if (¬model_in_flag ∧ tflag)
  {
    ⟨ Memory allocation of terminal radii 58 ⟩;
    if (n_sampling < n)
      ⟨ For all training samples, find the kNNs 62 ⟩;
    ⟨ Calculation of Termination Distance 59 ⟩;
  }
```

This code is used in section 11.

16. Determination of sample size in training samples. When no option `-tr` was given, equal size is considered in common to classes.

```
⟨ Equal sample size is assumed in common to classes for training data 16 ⟩ ≡
  {
    printf("\n␣*␣Wmmm␣...␣<␣Training␣Sample␣>set␣equivalent␣samples␣for␣each␣class␣by␣%d␣\n",
        n_tr/nc);
    fflush(stdout);
    for (int i = 0; i < nc; i++) tr_pc[i] = n_tr/nc;
  }
```

This code is used in section 11.

17. Determination of sample size in testing samples. When no option `-te` was given, equal size is considered in common to classes.

```
⟨ Equal sample size is assumed in common to classes for testing data 17 ⟩ ≡
  {
    printf("\n␣*␣Wmmm␣...␣<␣Testing␣Sample␣>set␣equivalent␣samples␣for␣each␣class␣by␣%d␣\n",
        n_te/nc);
    fflush(stdout);
    for (int i = 0; i < nc; i++) te_pc[i] = n_te/nc;
  }
```

This code is used in section 11.

18. Turn on the switch to all projected data.

⟨ Use projected samples in m principal axes 18 ⟩ ≡

```
{
  printf("\n␣*␣Because␣of␣small␣dimensions␣%d␣<=␣%d,␣we␣transform␣all␣data␣by␣PCA\n", m,
      PCAdim);
  allpca_flag = 1;
}
```

This code is used in section 15.

19. Assignment of several variables.

⟨ Assignment of several variables 19 ⟩ ≡

```
    /* ------------------------------------------------------------- */
    /* Key variables whose class are defined in matrix.h and matrix.C. */
    /* NN : an array of kth nearest neighbors */
    /* NNhalf : an array of k/2th nearest neighbors */
    /* select_level : the descritized level of error ϵ */
    /* thresh_l : an array of thresholds in each of l = 1, 2, ..., l_max */
    /* ------------------------------------------------------------- */
  int *NN;
  int *NNhalf;
  int select_level;
  REAL *thresh_l = new REAL[q + 1];    /* last one for full dimension */
  REAL thresh_1st;
```

See also sections 20, 21, 22, and 23.

This code is used in section 15.

20. Assignment of several variables.

⟨ Assignment of several variables 19 ⟩ +≡

```
    /* ------------------------------------------------------------- */
    /* Key variables whose class are defined in matrix.h and matrix.C. */
    /* X : training (stored) data (matrix) */
    /* M : the mean vector of training data (column vector) */
    /* Cov : the mean vector of training data (matrix) */
    /* Eval : the eigen values placed in the diagonal elements */
    /* (diagonal matrix) */
    /* Evec : the eigen (column) vectors ordered in the decreasing (matrix) */
    /* order of eigen values */
    /* ------------------------------------------------------------- */
  Matrix X(m, n);
  ColumnVector M(m);
  Matrix Cov(m, m);
  Matrix Eval(m, m);
  Matrix Evec(m, m);
```

21. Assignment of several variables.

⟨ Assignment of several variables 19 ⟩ +≡

```
    /* ------------------------------------------------------------ */
    /* Variables related to PCA. */
    /* PV : the principal variance */
    /* PX : the projected X on q principal axes */
    /* ptrx[ ] : the projected training data in array */
    /* ------------------------------------------------------------ */
  ColumnVector PV(m);    /* Principal Variance */
  Matrix PX(q, n);    /* projected X */
  if (allpca_flag) PX = Matrix(m, n);    /* projected X */
  REAL **ptrx;
```

22. Assignment of several variables.

⟨ Assignment of several variables 19 ⟩ +≡

```
    /* ------------------------------------------------------------ */
    /* Variables related to estimation of the empirical distributions. */
    /* n_sampling : the number of sampling points */
    /* n_tail : the number of end points corresponding to */
    /* specified ϵ */
    /* ------------------------------------------------------------ */
  int n_sampling;
  int sampling_skip = 1;
  int n_tail;
  ⟨ Set up of sampling parameters 65 ⟩
  Matrix SX(m, n_sampling + 1);
  Matrix PSX(q, n_sampling + 1);    /* projected SX */
```

23. Assignment of several variables.

⟨ Assignment of several variables 19 ⟩ +≡

```
/* ------------------------------------------------------------------- */
/* Variables related to estimation of the empirical distributions. */
/* n_tail : the number of end points corresponding to */
/* specified ε (error) at level 1, 2, ..., error_level */
/* XNtail(n_tail, q) : an array of tail samples */
/* gain[i][q] : reduction probabilities δ_q in q dimensions */
/* threshold[i][q] : reduction probabilities δ_q at level i */
/* P[i][q] : tail p-values in q dimensions at level i */
/* TermR2[] : the squared terminal radius of each sample */
/* ------------------------------------------------------------------- */
Matrix XNtail(n_tail, q + 1);    /* in q, hold the full dimensional tail */
static int n_error_level = 6;
REAL error_level[] = {0.0, 0.1, 0.5, 1.0, 5.0, 10.0};
REAL **gain = new REAL *[n_error_level];
REAL **threshold = new REAL *[n_error_level];
for (int i = 0; i < n_error_level; i++) {
  gain[i] = new REAL[q + 1];
  threshold[i] = new REAL[q + 1];
  gain[i][0] = 0.0;
}
Matrix P(n_tail + 1, q + 1);    /* Store tail p-value. In q, holds full */    /* dimensional p-value */
REAL *TermR2;
REAL **ev;
REAL *px;
```

24. File open of a model file.

```
⟨ Open a model file 24 ⟩ ≡
  {
    FILE *fp;
    fp = fopen(in_file, "r");
    if (tflag)
      try {
        TermR2 = new REAL[n];
      }
    catch(bad_alloc)
    {
      cerr ≪ "*E*␣cannot␣allocate␣memory␣for␣TermR2␣with␣size␣␣" ≪ n ≪ "␣for␣dimension␣" ≪
          m ≪ endl;
      exit(0);
    }
    int iallpca_flag, ipflag, itflag, irflag;    /* input flags */
    int iq, im, in;    /* input dimensions */
    if (allpca_flag) ptrx = trx;
    else {
      ptrx = new REAL *[n];
      for (int i = 0; i < n; i++) ptrx[i] = new REAL[q];
    }
    ⟨ Input of distribution parameters 69 ⟩
    if (0) {
      fp = stdout;
      ⟨ Output of distribution parameters 66 ⟩
    }
    if (iallpca_flag < allpca_flag) {
      printf("(Error)␣You␣cannot␣specify␣%s␣option␣that␣does␣not␣match␣th\
          e␣model␣file:␣flag␣=%d␣iflag=%d\n", "allpca_flag", allpca_flag, iallpca_flag);
      exit(0);
    }
    if (ipflag < pflag) {
      printf("(Error)␣You␣cannot␣specify␣%s␣option␣that␣does␣not␣match␣th\
          e␣model␣file:␣flag␣=%d␣iflag=%d\n", "pflag", pflag, ipflag);
      exit(0);
    }
    if (itflag < tflag) {
      printf("(Error)␣You␣cannot␣specify␣%s␣option␣that␣does␣not␣match␣th\
          e␣model␣file:␣flag␣=%d␣iflag=%d\n", "tflag", tflag, itflag);
      exit(0);
    }
    fclose(fp);
  }
```

This code is used in section 11.

25.

```
⟨ Output of the model to a file 25 ⟩ ≡
  {
    FILE *fp;
    fp = fopen(out_file, "w");
    ⟨ Output of distribution parameters 66 ⟩
    printf("All␣distribution␣parameters␣were␣held␣in␣file␣%s\n", out_file);
    fclose(fp);
  }
```

This code is used in section 11.

26. Let us analyze the statistics of kNNs on the basis of training samples.

⟨ Analysis of statistics of training data 26 ⟩ ≡

```
{
  int n_exam = 0;
  REAL p_mean;
  REAL p_var;
  ⟨ Find the mean and the variance of G(D^2(X,Y)) 27 ⟩
  cout << "(X,Y):␣Mean␣=␣" << p_mean << "\tVariance␣=␣" << p_var << endl;
  REAL nn_mean;
  REAL nn_var;
  NN = new int[n];
  if (tflag) NNhalf = new int[n];
  ⟨ Find the mean and the variance of F(D^2(X,X_kNN)) 28 ⟩
  cout << "(X,X_knn):␣Mean␣=␣" << nn_mean << "\tVariance␣=␣" << nn_var << endl;
#ifdef DEBUG
  cout << "In␣analysis:␣n_tail␣=␣" << n_tail << "␣for␣" << tail_percentage << "␣%␣of␣" <<
      n_sampling << "␣sampling␣data" << endl;
#endif
  if (q > m) {
    fprintf(stderr, "(Warning)␣q=%d␣>␣m=%d␣...␣q␣is␣replaced␣by␣m=%d\n", q, m, m);
    q = m;
  }
  REAL total_variance;
  REAL ratio;
  int n_sample_PCA;
  if (allpca_flag)
    if (sampled_eigen_flag) {
      cout << "In␣fast␣finding␣eigen␣vectors,␣use␣only␣sampled␣data" << endl;
      ⟨ Calculation of eigen vectors in m dimensions using sampled data 54 ⟩
    }
    else ⟨ Calculation of eigen vectors in m dimensions using all data 53 ⟩
  else if (sampled_eigen_flag) {
    cout << "In␣finding␣eigen␣vectors,␣use␣only␣sampled␣data" << endl;
    ⟨ Calculation of eigen vectors in q dimensions using sampled data 52 ⟩
  }
  else ⟨ Calculation of eigen vectors in q dimensions using all data 51 ⟩
  REAL cont_eff = 0.0;
  for (int i = 0; i < q; i++) {
    PV(i) = Eval(i, i);
    cont_eff += PV(i)/total_variance;
  }
  cout << "n_sample_PCA␣=␣" << n_sample_PCA << "␣cont_eff␣=␣" << cont_eff << endl;
  if (q > 0) {
    ⟨ Projection of data on q eigen vectors to obtain PX and PSX 55 ⟩
    REAL *p_p_mean = new REAL[q];
    REAL *p_p_var = new REAL[q];
    ⟨ Find the mean and variance of G(D_l^2(X,Y)) 30 ⟩
    for (int k = 0; k < q; k++) {
      cout << "(X,Y):␣Mean␣(" << k << ")␣=␣" << p_p_mean[k] << "\tVariance␣(" << k << ")␣=␣" <<
          p_p_var[k] << endl;
    }
```

```
      REAL *p_nn_mean = new REAL[q];
      REAL *p_nn_var = new REAL[q];
      REAL *p_nn_tail = new REAL[q];
      ⟨ Find the mean, variance, tail point of F(D_l^2(X, X_kNN)) 31 ⟩
      for (int k = 0; k < q; k++) {
        cout << "(X,X_knn):␣Mean␣(" << k << ")␣=␣" << p_nn_mean[k] << "\tVariance␣(" << k <<
            ")␣=␣" << p_nn_var[k] << endl;
      }
      ⟨ Estimate of tail probability of G(D_l^2(X, Y)) 32 ⟩
    }
  }
```

This code is used in section 15.

27.

⟨ Find the mean and the variance of $G(D^2(X,Y))$ 27 ⟩ ≡

```
  {
    int n_pair_sampling = n_sampling * (n_sampling - 1)/2;
    ColumnVector D(n_pair_sampling);     /* Store all distances */
    int nn = sampling_skip * n_sampling;
    int cc = 0;
    for (int i = 0; i < nn; i += sampling_skip)
      for (int j = i + sampling_skip; j < nn; j += sampling_skip) D(cc++) = DIS2(i, j);
#ifdef DEBUG
    cout << "Mean␣=␣" << D.Mean() << endl;
    cout << "Variance␣=␣" << D.Variance(D.Mean()) << endl;
#endif
    p_mean = D.Mean();
    p_var = D.Variance(D.Mean());
  }
```

This code is used in section 26.

28.

⟨ Find the mean and the variance of $F(D^2(X, X_{kNN}))$ 28 ⟩ ≡

```
{
  REAL pdisq;
  int cc = 0;
  REAL vmin;
  int nmin;
  ColumnVector D(n_sampling);
  Matrix DkNN;
  if (kNNoutput ≡ 1) DkNN = Matrix(n_sampling, m);
  int *kno = new int[K];
  REAL *kdis = new REAL[K];
  cc = 0;
  int nn = sampling_skip * n_sampling;
  for (int i = 0; i < nn; i += sampling_skip) {
    find_k_nn_in_training(K, i, n, trx, kno, kdis);
    NN[cc] = kno[K - 1];
    if (kNNoutput ≡ 1)
      for (int j = 0; j < m; j++) DkNN(cc, j) = (trx[NN[cc]][j] - trx[i][j]);
    D(cc++) = kdis[K - 1];
    if (tflag ∧ (nc > 1 ∨ K ≡ 1)) NNhalf[i] = kno[int((K + 1)/2)];
  }
  delete[] kno;
  delete[] kdis;
  nn_mean = D.Mean();
  nn_var = D.Variance(D.Mean());
  if (kNNoutput ≡ 1) {
    FILE *fp;
    fp = fopen("sampled_kNN", "w");
    Matrix DkNNM;
    DkNN.Output(fp, "kNN's");
    DkNNM = DkNN.t().MeanVec();
    DkNNM.Print("Mean␣of␣kNN");
    DkNN.t().VarianceMatrix(DkNNM).Print("Covariance␣matrix␣of␣kNN");
    fclose(fp);
  }
}
```

This code is used in section 26.

29. Empirical percentile method. Using the empirical distributions, we determine the threshold values in each q.

30. Mean and variance of $G(D_l^2(X,Y))$.

⟨ Find the mean and variance of $G(D_l^2(X,Y))$ 30 ⟩ ≡

```
{
  Matrix G(n_sampling * (n_sampling - 1)/2, q);    /* Store all distances */
  int cc = 0;
  for (int i = 0; i < n_sampling; i++) {
    for (int j = i + 1; j < n_sampling; j++) {
      for (int s = 0; s < q; s++)
        G(cc, s) = (PSX.Cols(i, i).Rows(0, s) - PSX.Cols(j, j).Rows(0, s)).Norm2();
      cc++;
    }
  }
  for (int k = 0; k < q; k++) {
    p_p_mean[k] = G.Cols(k, k).Mean();
    p_p_var[k] = G.Cols(k, k).Variance(p_p_mean[k]);
  }
  if (n_sampling * (n_sampling - 1)/2 ≠ cc)
    printf("*SSS*␣n_sampling*(n_sampling-1)/2=%d,␣cc=%d\n", n_sampling * (n_sampling - 1)/2, cc);
}
```

This code is used in section 26.

31. Mean, variance and thresholds of $F(D_l^2(X, X_{kNN}))$.

⟨Find the mean, variance, tail point of $F(D_l^2(X, X_{kNN}))$ 31⟩ ≡

```
{
  Matrix F(n_sampling, q);     /* Store all distances */
  int j = 0;
  REAL tmp;
  REAL dd;
  int cc = 0;
  int nn = sampling_skip * n_sampling;
  int ii;
  for (int i = 0; i < n_sampling; i++) {
    for (int k = 0; k < q; k++) {
        /* Be careful! PSX is already sampled and NN is for sampled ones */
      F(i, k) = (PSX.Cols(i, i).Rows(0, k) − PX.Cols(NN[i], NN[i]).Rows(0, k)).Norm2( );
      if (i < n_tail) {
        P(i, k) = F(i, k);
      }
      else if (i ≡ n_tail) {
        P(i, k) = F(i, k);     /* Bubble Sort in decreasing order */
        for (int i1 = 0; i1 < n_tail; i1++)
          for (int i2 = n_tail; i2 > i1; i2−−) {
            if (P(i2 − 1, k) < P(i2, k)) {
              tmp = P(i2, k);
              P(i2, k) = P(i2 − 1, k);
              P(i2 − 1, k) = tmp;
            }
          }
      }
      else {
        if (F(i, k) > P(n_tail − 1, k)) {
          P(n_tail, k) = F(i, k);
          for (int i2 = n_tail; i2 > 0; i2−−) {
            if (P(i2 − 1, k) < P(i2, k)) {
              tmp = P(i2, k);
              P(i2, k) = P(i2 − 1, k);
              P(i2 − 1, k) = tmp;
            }
          }
        }
      }
    }
         /* Full dimensional tail——— */
    ii = i * sampling_skip;
    dd = DIS2(ii, NN[i]);
    if (i < n_tail) {
      P(i, q) = dd;
    }
    else if (i ≡ n_tail) {     /* Bobble Sort */
      P(i, q) = dd;
      for (int i1 = 0; i1 < n_tail; i1++)
        for (int i2 = n_tail; i2 > i1; i2−−) {
```

```
            if (P(i2 − 1, q) < P(i2, q)) {
              tmp = P(i2, q);
              P(i2, q) = P(i2 − 1, q);
              P(i2 − 1, q) = tmp;
            }
          }
        }
        else {
          if (dd > P(n_tail − 1, q)) {
            P(n_tail, q) = dd;
            for (int i2 = n_tail; i2 > 0; i2 −−) {
              if (P(i2 − 1, q) < P(i2, q)) {
                tmp = P(i2, q);
                P(i2, q) = P(i2 − 1, q);
                P(i2 − 1, q) = tmp;
              }
            }
          }
        }    /* ------------------------------ */
        cc ++;
      }
      for (int k = 0; k < q; k++) {
        p_nn_mean[k] = F.Cols(k, k).Mean( );
        p_nn_var[k] = F.Cols(k, k).Variance(p_nn_mean[k]);
      }
      if (n_sampling ≠ cc) printf("*SST*␣n_sampling=%d,␣cc=%d\n", n_sampling, cc);
    }
```

This code is used in section 26.

32.

⟨ Estimate of tail probability of $G(D_l^2(X,Y))$ 32 ⟩ ≡

```
{
  Matrix G(n_sampling * (n_sampling - 1)/2, q);    /* Store all distances */
  int cc = 0;
  REAL dd;
  int ii, jj;
  int nn = sampling_skip * n_sampling;
  for (int i = 0; i < n_sampling; i++) {
    for (int j = i + 1; j < n_sampling; j++) {
      for (int k = 0; k < q; k++) {
        G(cc, k) = (PSX.Cols(i, i).Rows(0, k) - PSX.Cols(j, j).Rows(0, k)).Norm2();
        for (int l = 0; l < n_tail; l++)
          if (G(cc, k) > P(l, k)) XNtail(l, k) += 1.0;
      }    /* full dimensional tail */
      ii = i * sampling_skip;
      jj = j * sampling_skip;
      dd = DIS2(ii, jj);
      for (int l = 0; l < n_tail; l++)
        if (dd > P(l, q)) XNtail(l, q) += 1.0;
      cc++;
    }
  }
  XNtail /= cc;
}
```

This code is used in section 26.

33. Utility functions.

```
⟨ Function Declaration 33 ⟩ ≡
#define max(x, y) ((x) > (y) ? (x) : (y))
#define min(x, y) ((x) < (y) ? (x) : (y))
#define abs(x) ((x) > 0.0 ? (x) : (−(x)))
  int getword_of_line (FILE *fp, char * line , char word[][50], int max ) ;
      /* L2 (Euclidean) distance ——————————————- */
    inline REAL L2dis(REAL * a, REAL * b)
    {
      REAL sum = 0.0;
      REAL diff ;
      for (int i = 0; i < m; i++) {
        diff = a[i] − b[i];
        sum += diff * diff ;
      }
      return sqrt(sum);
    }
    inline REAL L2dis2(REAL * a, REAL * b)
    {
      REAL sum = 0.0;
      REAL diff ;
      for (int i = 0; i < m; i++) {
        diff = a[i] − b[i];
        sum += diff * diff ;
      }
      return sum;
    }
    inline REAL L2dis2(ColumnVector & a, ColumnVector & b)
    {
      REAL sum = 0.0;
      REAL diff ;
      for (int i = 0; i < m; i++) {
        diff = a(i) − b(i);
        sum += diff * diff ;
      }
      return sum;
    }
    inline REAL L2dis2(int k, int l)
    {
      REAL sum = 0.0;
      REAL diff ;
      for (int i = 0; i < m; i++) {
        diff = trx[k][i] − trx[l][i];
        sum += diff * diff ;
      }
      return sum;
    }   /* L1 (city block) distance ——————————————- */
    inline REAL L1dis(REAL * a, REAL * b)
    {
      REAL sum = 0.0;
      REAL diff ;
```

```
    for (int i = 0; i < m; i++) {
      diff = a[i] - b[i];
      sum += abs(diff);
    }
    return sum;
  }
  inline REAL L1dis2(REAL *a, REAL *b)
  {
    return L1dis(a, b);
  }
  inline REAL L1dis2(ColumnVector &a, ColumnVector &b)
  {
    REAL sum = 0.0;
    REAL diff;
    for (int i = 0; i < m; i++) {
      diff = a(i) - b(i);
      sum += abs(diff);
    }
    return sum;
  }
  inline REAL L1dis2(int k, int l)
  {
    return L1dis(trx[k], trx[l]);
  }    /* Linf (max) distance ------------------------- */
  inline REAL Linfdis(REAL *a, REAL *b)
  {
    REAL sum = 1. · 10^-50;
    REAL diff;
    for (int i = 0; i < m; i++) {
      diff = a[i] - b[i];
      sum = max(sum, abs(diff));
    }
    return sum;
  }
  inline REAL Linfdis2(REAL *a, REAL *b)
  {
    return Linfdis(a, b);
  }
  inline REAL Linfdis2(ColumnVector &a, ColumnVector &b)
  {
    REAL sum = 1. · 10^-50;
    REAL diff;
    for (int i = 0; i < m; i++) {
      diff = a(i) - b(i);
      sum = max(sum, abs(diff));
    }
    return sum;
  }
  inline REAL Linfdis2(int k, int l)
  {
```

```
  return Linfdis(trx[k], trx[l]);
}
inline REAL innerp(REAL * a, REAL * b)
{
  REAL sum = 0.0;
  for (int i = 0; i < m; i++) sum += a[i] * b[i];
  return sum;
}
inline REAL innerp(ColumnVector & a, ColumnVector & b)
{
  REAL sum = 0.0;
  for (int i = 0; i < m; i++) sum += a(i) * b(i);
  return sum;
}
inline REAL norm(REAL * a)
{
  REAL sum = 0.0;
  for (int i = 0; i < m; i++) sum += a[i] * a[i];
  return sqrt(sum);
}
REAL stock_k_nn(int k, REAL entry, int entry_no, int *ono, REAL * ostock, int set_flag = 0, int
    init_flag = 0)
    /* k ... holds up to k smallest values */
    /* entry ... new value */
    /* entry_no ... the sample number */
    /* ono[] ... set the inner contents (numbers of kNNs) to ono if set_flag = 1 */
    /* ostock[] ... set the inner contents (kNN distances) to ostock if set_flag = 1 */
    /* initflag ... 0: no initialize 1: initialize of memory */    /* */
{
  static int cc = 0;
  static REAL *stock = 0;    /* distance */
  static int *no;    /* member */
  static int num = 0;
  if (cc ≡ 0)    /* Memory setup */
  {
    try {
      stock = new REAL[k + 1];    /* plus one */
      for (int i = 0; i < k; i++) stock[i] = 1. · 10^33;
      no = new int[k + 1];
    }
    catch(bad_alloc)
    {
      cerr ≪ "Cannot␣alloc␣memory␣for␣stock␣and␣no␣" ≪ endl;
      exit(0);
    }
  }
  if (cc ≡ 0 ∨ init_flag ≠ 0)    /* Initialize */
  {
    for (int i = 0; i < k; i++) stock[i] = 1. · 10^33;
    for (int i = 0; i < k; i++) no[i] = (−1);
    num = 0;
```

```
      cc++;
      return stock[k − 1];
    }
    if (set_flag)     /* Return the results */
    {
      for (int i = 0; i < k; i++) {
        ostock[i] = stock[i];
        ono[i] = no[i];
      }
      return ostock[k − 1];
    }
    cc++;
    num++;     /* Updating */
    if (k ≡ 1) {
      if (entry < stock[0]) {
        stock[0] = entry;
        no[0] = entry_no;
      }
    }
    else {
      stock[k] = entry;
      no[k] = entry_no;
      REAL tmp;
      int tmpi;
      for (int i = k − 1; i ≥ 0; i−−) {
        if (stock[i] > stock[i + 1]) {
          tmp = stock[i];
          stock[i] = stock[i + 1];
          stock[i + 1] = tmp;
          tmpi = no[i];
          no[i] = no[i + 1];
          no[i + 1] = tmpi;
        }
        else break;
      }
    }
#ifdef DEBUG
    printf("Answer␣for␣%d␣data:␣", num);
    for (int i = 0; i < k; i++) printf("(%d,␣%f)␣", no[i], stock[i]);
    printf("\n");
    fflush(stdout);
#endif
    return stock[k − 1];
  }
  inline REAL find_k_nn_in_training(int k, int about, int n, REAL **tr, int *kno, REAL *kdis)
  {
    stock_k_nn(k, 0.0, 0, kno, kdis, 0, 1);     /* Initialize */
    REAL d;
    for (int i = 0; i < n; i++) {
      if (i ≡ about) continue;
      d = DIS2(about, i);
```

```
      stock_k_nn(k, d, i, kno, kdis);    /* updating */
    }
    stock_k_nn(k, 0.0, 0, kno, kdis, 1);    /* value setting; */
    return kdis[k - 1];
  }
  inline int comp_score(const void *p1, const void *p2){ CELL * s1 = ( CELL * ) p1; CELL * s2 =
        ( CELL * ) p2;
      double v = s1->val - s2->val;
      if (v > 0.) return 1;
      else if (v < 0.) return (-1);
      else return 0;
      }
```

See also sections 49, 50, and 72.

This code is used in section 10.

34. For reading the number of data n and the number of features m, read the file and count the number of "carriage return" and the number of columns in a line.

⟨ Determination of sizes 34 ⟩ ≡

```
{ FILE *fp_in;
  if ((fp_in = fopen(file_in, "r")) == NULL) {
    fprintf(stderr, " * file <%s> open error * \n", file_in);
    exit(0);
  }
  int nw;
  nread = 0; while ( ( nw = getword_of_line (fp_in, line , word, Max_Line ) ) != 0 )
  {
    if (n == 0) m = nw;
    nread++;
  }
  fclose(fp_in); }
```

This code is used in sections 35 and 36.

35. Read training data.

⟨ Read data for training 35 ⟩ ≡

```
char
line [Max_Line] ;
char word[Max_Line][50];
FILE *fp_tr;
int nread;
file_in = file_tr;
⟨ Determination of sizes 34 ⟩;
n_tr = n = nread;
cout << "n = " << n << " m = " << m << endl;
if ((fp_tr = fopen(file_tr, "r")) ≡ Λ) {
  fprintf(stderr, " * training file <%s> open error * \n", file_tr);
  exit(0);
}
try {
  trx = new REAL * [n];
  for (int i = 0; i < n; i++) trx[i] = new REAL[m];
}
catch(bad_alloc)
{
  cerr << "*E* cannot allocate memory for data with n,m= " << n << " " << m << endl;
  exit(0);
}
if (model_in_flag) { int i; for (int ii = 0; ii < n; ii++) { i = iri[ii];
int nw = 0; if ( ( nw = getword_of_line (fp_tr, line , word, Max_Line ) ) ≠ 0 )
{
  for (int k = 0; k < nw; k++) trx[i][k] = REAL(atof(word[k]));
}
} } else { for (int i = 0; i < n; i++) { int nw = 0; if ( ( nw = getword_of_line (fp_tr, line ,
    word, Max_Line ) ) ≠ 0 )
{
  for (int k = 0; k < nw; k++) trx[i][k] = REAL(atof(word[k]));
}
} } fclose(fp_tr);
```

This code is used in section 11.

36. Read data for testing.

⟨ Read data for testing 36 ⟩ ≡

```
{ cout << "In read testing data ! " << endl; char
line [Max_Line] ;
char word[Max_Line][50];
FILE *fp_tr;
int D = 0;
int nread;
file_in = file_te;
if (¬std_in_flag) {
  ⟨ Determination of sizes 34 ⟩;
  n_te = nread;
  cout << "n_te = " << n_te << " m = " << m << endl;
  if ((fp_tr = fopen(file_te, "r")) ≡ Λ) {
    printf(" * testing file < %s > open error * \n", file_te);
    exit(0);
  }
  te_data = new REAL * [n_te];
  {
    for (int i = 0; i < n_te; i++) te_data[i] = new REAL[m];
  }
}
else    /* Standard input */
{
  fp_tr = stdin;
  n_te = 1;
}
for (int i = 0; i < n_te; i++) { int nw = 0; if ( ( nw = getword_of_line (fp_tr, line , word, Max_Line )
    ) ≠ 0 )
{
  if (nw ≠ m) {
    cerr << "In reading test sample " << i << " size " << nw << " m= " << m << endl;
    exit(0);
  }
  for (int k = 0; k < nw; k++) te_data[i][k] = REAL(atof(word[k]));
}
} fclose(fp_tr); }
```

This code is used in section 11.

37. Analysis of argument.

```
⟨ Analysis of arguments 37 ⟩ ≡
  if (argc < 3) {
    ⟨ Usage 39 ⟩
    exit(0);
  }
  file_tr = argv[1];
  if ((fp_tr = fopen(argv[1], "r")) ≡ Λ) {
    printf("␣*␣training␣file␣<␣%s␣>␣open␣error␣*␣\n", argv[1]);
    exit(0);
  }
  argc −−;
  argv ++;
  if (argv[1][0] ≠ '-') {
    file_te = argv[1];
    argc −−;
    argv ++;
  }
  else    /* use standard input instead a file for query */
  {
    std_in_flag = 1;
    argc −−;
    argv ++;
  }    /* +++ option analyze +++ */
  char c;
  while (−− argc ∧ (∗++ argv)[0] ≡ '-') {
    switch (c = ∗++ argv[0]) {
    case 'p': printf("␣A␣priori␣probabilities␣are␣:␣");
      for (int i = 0; i < nc; i++) {
        p[i] = atof(∗++ argv);
        −− argc;
        printf("␣%f", p[i]);
      }
      putchar('\n');
      p_flag = 1;
      break;
    case 'Q': q = qdim = atoi(∗++ argv);
      −− argc;
      printf("(Option)␣the␣maximum␣dimension␣of␣marginal␣distance␣was␣set␣to␣%d\n", qdim);
      break;
    case 'L': qsdim = atoi(∗++ argv);
      −− argc;
      printf("(Option)␣the␣dimension␣of␣marginal␣distance␣was␣set␣to␣%d\n", qsdim);
      break;
    case 'E': eta = atof(∗++ argv);
      −− argc;
      printf("(Option)␣approximation␣degree␣eta␣was␣set␣to␣%f\n", eta);
      break;
    case 'P': pflag = 1;
      printf("(Option)␣use␣of␣partial␣distance␣strategy\n");
      break;
```

```
case 'R': rflag = 1;
  printf("(Option)␣use␣of␣recovery␣process␣for␣irregularly␣large␣kNN\n");
  break;
case 'T': tflag = 1;
  printf("(Option)␣use␣of␣termination␣strategy\n");
  break;
case 'N':
  switch (c = *++argv[0]) {
  case 'P': pflag = 0;
    printf("(Option)␣no␣use␣of␣partial␣distance␣strategy\n");
    break;
  case 'R': rflag = 0;
    printf("(Option)␣no␣use␣of␣recovery␣process␣for␣irregularly␣large␣kNN\n");
    break;
  case 'T': tflag = 0;
    printf("(Option)␣no␣use␣of␣termination␣strategy\n");
    break;
  default: break;
  }
  break;
case 't':
  switch (c = *++argv[0]) {
  case 'r': printf("␣Training␣sample␣numbers␣are␣:␣");
    for (int i = 0; i < nc; i++) {
      tr_pc[i] = atoi(*++argv);
      --argc;
      printf("␣%d", tr_pc[i]);
    }
    putchar('\n');
    tr_flag = 1;
    break;
  case 'e': printf("␣Test␣sample␣numbers␣are␣:␣");
    for (int i = 0; i < nc; i++) {
      te_pc[i] = atoi(*++argv);
      --argc;
      printf("␣%d", te_pc[i]);
    }
    putchar('\n');
    te_flag = 1;
    break;
  default: printf("*error*␣no␣option␣as␣-t%c␣\n", c);
    exit(0);
    break;
  }
  break;
case 'u': uflag = 1;
  break;
case 'k': K = kk = atoi(*++argv);
  --argc;
  if (kk > KMAX) {
    fprintf(stderr, "*␣error␣*␣␣number␣%d␣over␣max␣%d␣\n", kk, KMAX);
    exit(0);
```

```
    }
    break;
  case 'S': sampling_ceil = atoi(*++argv);
    --argc;
    printf("*␣sampling_ceil␣was␣set␣to␣%d␣\n", sampling_ceil);
    break;
  case 'A': A = atoi(*++argv);
    --argc;
    printf("*␣A␣was␣set␣to␣%d␣\n", A);
    break;
  case 'F': narrow = Narrowing(atoi(*++argv));
    --argc;
    printf("*␣narrow␣was␣set␣to␣%d␣\n", narrow);
    break;
  case 'z': tail_percentage = atof(*++argv);
    --argc;
    cut_off = tail_percentage/100.0;
    printf("␣Cut_off␣(the␣probability␣of␣error␣nearest␣neighbor)␣was␣set␣to␣%f␣\n", cut_off);
    break;
  case 'C': topS = atoi(*++argv);
    --argc;
    printf("␣topS␣was␣set␣in␣option␣to␣%d\n", topS);
    break;
  case 'b': switch_bisector = 1;
    printf("␣Bisectors␣will␣be␣used␣for␣narrowing␣the␣candidate␣region␣\n");
    break;
  case 'a': switch_ball = 1;
    printf("␣Balls␣will␣be␣used␣for␣narrowing␣the␣candidate␣region␣\n");
    break;
  case 'm': Default_max_nn = atof(*++argv);
    --argc;
    printf("␣Max_nn␣was␣set␣to␣%f␣\n", Default_max_nn);
    break;
  case 'O': out_file = *++argv;
    --argc;
    model_out_flag = 1;
    printf("Out_file␣was␣set␣to␣%s␣\n", out_file);
    break;
  case 'I': in_file = *++argv;
    --argc;
    model_in_flag = 1;
    printf("Model_in_file␣was␣set␣to␣%s␣\n", in_file);
    break;
  case 'c': nc = atoi(*++argv);
    --argc;
    printf("Class␣number␣is␣set␣to␣%d␣\n", nc);
    if (nc > CLASS) {
      fprintf(stderr, "*␣error␣*␣class␣number␣%d␣over␣max␣%d␣\n", c, CLASS);
      exit(0);
    }
    break;
  case 'f': sampled_eigen_flag = 1;
```

```
      break;
    case 'h': cout ≪ "with␣h" ≪ endl;
      ⟨ Usage 39 ⟩
      exit(0);
      break;
    default: fprintf(stderr, "␣*␣Error␣*␣Such␣an␣option␣does␣not␣exist␣!␣..␣%c␣\n", c);
      exit(0);
      break;
    }
  }
  ⟨ Modification of options 38 ⟩
  printf("(Option␣flags)␣pflag=%d␣tflag=%d␣rflag=%d␣allpca_flag=%d␣q=%d␣A=%d\n", pflag, tflag,
      rflag, allpca_flag, q, A);
  printf("*Nearest␣neighbor␣number␣is␣set␣to␣K=%d␣\n", kk);
  printf("*Narrowing␣:␣%d␣\n", int(narrow));
  if (A ≡ 0) {
    printf("*␣Exact␣search␣was␣chosen␣with␣A=%d␣\n", A);
    q = 1;
    narrow = Nothing;
  }
```

This code is used in section 11.

38. Modification of options. Some flags are modified so as to keep consistency.

```
⟨ Modification of options 38 ⟩ ≡
  if (A ≡ 0) allpca_flag = rflag = q = 0;
  if (nc ≡ 0 ∨ kk > 0) tflag = 0;
```

This code is used in section 37.

39. Message for usage.

⟨ Usage 39 ⟩ ≡

```
printf("<␣Usage␣of␣%s␣(ver.␣%s)␣as␣of␣%s␣>␣%s␣training-file␣test-file␣[-k␣k[%d]][-c␣c[%\
    d]][-tr␣n1␣n2␣...␣nc][-te␣m1␣m2␣...␣mc][-A␣select_number␣[%d]][-Q␣q(marginal␣dim\
    ension␣[%d])[-E␣eta][-P][-NP][-T][-NT][-R][-NR][-S␣sampling_no][-O(ut)␣Modelfile\
    ][-I(n)␣ModelFile][-F␣filter-mode][-u][-z␣cut-off][-f]\n", argv[0], ver, __DATE__, argv[0],
    K, nc, A, qdim);
printf("\t␣-k␣k␣....␣the␣number␣of␣nearest␣neighbors␣[Default␣:␣%d]␣\n", K);
printf("\t␣-c␣c␣....␣␣the␣number␣of␣classes␣[Default␣:␣%d]␣\n", nc);
printf("\t␣-tr␣n1␣....␣nc:␣the␣numbers␣of␣training␣samples␣in␣class␣order␣[Default␣:␣eq\
    ually]\n");
printf("\t␣-te␣m1␣....␣mc:␣the␣numbers␣of␣testing␣samples␣in␣class␣order␣[Default␣:␣equ\
    ally]\n");
printf("\t␣-A␣[%d]␣...␣error␣level␣(-1):␣interactive␣choice,␣0:0%,␣1:0.1%,␣2:0.5%,␣3:1%\
    ,␣4:5%,␣5:10%\n", A);
printf("\t␣-Q␣q␣....␣the␣maximum␣dimension␣of␣marginal␣distance␣strategy␣[Default␣%d]␣\n",
    qdim);
printf("\t␣-E␣eta␣....␣approximate␣nearest␣neighbor␣of␣(1+eta)␣times␣the␣nearest␣negihb\
    or␣distance␣[Defalt␣%f]␣\n", eta);
printf("\t␣-L␣q␣....␣the␣user␣specified␣dimension␣of␣marginal␣distance␣strategy␣[Defaul\
    t␣%d]␣\n", qsdim);
printf("\t␣-z␣cut-off␣....␣the␣probability␣of␣error␣nearest␣neighbor␣[Default␣%f␣\%]␣\n",
    tail_percentage);
printf("\t␣-P␣or␣-NP␣....␣:␣use␣or␣not␣of␣the␣partial␣distance␣strategy␣[Default␣%s]␣\n",
    (pflag) ? "ON" : "OFF");
printf("\t␣-T␣or␣-NT␣....␣:␣use␣or␣not␣of␣the␣termination␣criterion␣[Default␣%s]␣\n",
    (tflag) ? "ON" : "OFF");
printf("\t␣-R␣or␣-NR␣....␣:␣use␣or␣not␣of␣the␣recovery␣process␣for␣irregularly␣large␣kN\
    N␣[Default␣%s]\n", (rflag) ? "ON" : "OFF");
printf("\t␣-S␣s␣...:␣the␣maximum␣number␣of␣sampling␣points␣[Default␣%d]␣\n", sampling_ceil);
printf("\t␣-F␣fmode␣...␣0:␣Nothing,␣1:Filtering,␣2:Binary_search,␣3:␣Table-lookup␣[%d]\n",
    narrow);
printf("\t\t␣*Warning*␣if␣you␣choose␣2␣or␣more,␣then␣0.1%%␣error␣is␣added.\n");
printf("\t␣-f␣...:␣fast␣calculation␣of␣eigen␣vectors␣using␣sampled␣points␣[Default␣%d]␣\
    \n", sampled_eigen_flag);
```

This code is used in section 37.

40. Constants.

⟨ Global Constants Declaration 40 ⟩ ≡

```
#define REALdouble /* The data type of samples */
    //#define␣REAL␣float␣//␣The␣data␣type␣of␣samples␣
#define LP 2 /* Minkowski LP-norm. Specify one of 1,2,inf */
#define DIS L2dis /* Activate this one if you like to use Euclidean distance */
    //#define␣DIS␣L1dis␣␣␣␣␣//␣$L_1$␣distance␣
    //#define␣DIS␣Linfdis␣␣␣//␣$L_\infty$␣distance␣␣
#define DIS2 L2dis2 /* Squared distance. Active only one of these. */
    //#define␣DIS2␣L1dis2␣␣␣␣␣//␣Squared␣distance.␣Active␣one␣corresponding␣to␣
    //#define␣DIS2␣Linfdis2␣␣␣␣␣//␣Squared␣distance.␣Active␣one␣corresponding␣to␣
      const REALEPS = 1. · 10^-10;
    YESNOdo_orthonormal = NO;
    int sampling_ceil = 1000;
    float tail_percentage = 10.0; /* % */
    float cut_off = tail_percentage/100.0; /* Several flags */
    int dflag = 0; /* For debugging */
    int qflag = 1; /* Switch of marginal distance strategy (1: ON, 0: OFF) */
    int pflag = 1; /* Switch of partial distance strategy (1: ON, 0: OFF) */
    int tflag = 0; /* Switch of termination strategy (1: ON, 0: OFF) */
    int rflag = 0; /* Switch of recovery process for irregularly large (1: ON, 0: OFF) */
    int allpca_flag = 0; /* Switch of full transformation for small dimensions ¡= */
      /* 10 (1: ON, 0: OFF) */
    int sampled_eigen_flag = 0; /* Switch of use of sampled points only for PCA */
    int ocflag = 0; /* Default parameters */
    int PCAdim = 10;
    int qdim = 10; /* maximum marginal dimension */
    int qsdim = 0; /* specified marginal dimension */
    int q = qdim;
    float eta = 0.0; /* Exact k-NN */
    int K = 1; /* k- nearest */
    int kk = K; /* k- nearest */
    int nc = 1; /* the number of classes */ /* No use flag */
    int std_in_flag = 0;
    int uflag = 0; /*int A = (-1); */
    int A = 1;
    const int kNNoutput = 1; /* Output the statistics of kNNs */
      /* The bolow is the limitations of this program */
    const int KMAX = 1000; /* maximum value of k */
    const int CLASS = 256; /* maximum number of classes */
    const int DATAMAX = 150000; /* maximum number of samples */
    const int DIMMAX = 10000; /* maximum number of features (dimensions) */
    const int FMAX = DIMMAX;
    const int LINEMAX = 65000; /* maximim size (in byte) for a line of data */
    const int Max_Line = LINEMAX;
    const int BISECMAX = 100;
    const int max_checked_rank = 5;
    const int BS_FLAG = 1;
    const int B_default = 100;
    enum Narrowing {
      Nothing = 0, Filtering = 1, Binary_Search = 2, Table_Lookup = 3
```

```
};
enum Narrowing narrow = Filtering;
typedef struct cell {
  int id;
  int rid;
  REAL val;
} CELL;
REAL c1 = 3.0;   /* distance */
REAL c2 = 3.0;   /* inner product */
int checked_rank = max_checked_rank;
long *reject_count;
FILE *fp_tr;
FILE *fp_te;
FILE *fp_tmp;
char *file_tr;
char *file_te;
char *file_in;   /* For general use */
int n_tr = 0;   /* number of training samples */
int n_te = 0;   /* number of testing samples */
int n;   /* total sample number */
int m;   /* dimensionality */
int n_fea;
int tr_pc[CLASS];   /* Training : the data number for each class */
int te_pc[CLASS];   /* Test : the data number for each class */
int ncorrect;   /* # of correct */
int nwrong;   /* # of mistaken */
char
line [LINEMAX] ;
char w[DIMMAX][50];
REAL p[CLASS];   /* A priori probabilities */
int conf_matrix[CLASS][CLASS + 1];   /* Confusion matrix */
REAL **trx;   /* Sample */
REAL **te_data;   /* Sample */
REAL **pdis2;
REAL *tr_norm;
REAL xnorm;
REAL *zero;   /* null vector */
int tr_flag = 0;
int te_flag = 0;
int p_flag = 0;
int texflag = 0;
int testflag = 0;
REAL Default_max_nn = 1. · 10^10;
int level = 0;
REAL costsum;
REAL ans;
int nn_fea = 0;
REAL *var;
char *ver = "(1.3)";
```

```
int S = 10000;
int topS;
int switch_bisector = 0;
int switch_ball = 0;
int updated = 0;
REAL *disbset;
int *bestp;
REAL current_d = 1.e50;
REAL current_db = current_d;
int current_p = (-1);
int current_pb = (-1);
REAL e;
static char *in_file = "ModelFile";
static char *out_file = "ModelFile";
int model_in_flag = 0;
int model_out_flag = 0;
```

This code is used in section 10.

41. Header Files.

⟨ Header Files 41 ⟩ ≡

```
#include <iostream>
#include <cstdio>
#include <memory>
#include <string>
#include <time.h>
#include <fstream>
#include "math.h"
#include "matrix.h"
  using namespace std;
```

This code is used in section 10.

42. Get a test sample x.

⟨ Get a test sample 42 ⟩ ≡

```
  x = te_data[i];
```

This code is used in section 11.

43. Initialization of searching.

⟨ Initialize of searching 43 ⟩ ≡

```
  if (allpca_flag) {
    for (int j = 0; j < m; j++) px[j] = innerp(x, ev[j]);
    if (q > 0) theta_q = thresh_l[q - 1];
  }
  else if (q > 0) {
    for (int j = 0; j < q; j++) px[j] = innerp(x, ev[j]);
    theta_q = thresh_l[q - 1];
  }
  checked_sample = 0;
  current_d2 = stock_k_nn(K, 0.0, 0, knn_id, knn_dis, 0, 1);    /* Initialize */
```

This code is used in section 12.

44. Updating of the current k-nearest neighbors.

⟨ Update of kNNs 44 ⟩ ≡

```
{ #
ifdef DEBUG cerr << "In␣>␣Update␣of␣kNN␣with␣current_d2=␣" << current_d2 << "␣in␣sample␣" <<
        i << "␣rid=" << pair[i].id << endl;
# endif current_d2 = stock_k_nn(K, r2, i, knn_id, knn_dis); }
```

This code is used in sections 12 and 48.

45. Termination judgement. In this program, a terminal condition can be applied with option "-T." The idea is simple. When a query point q falls in the ball of radius $D(x, x_{1NN})/2$ centered at x, then the true nearest neighbor of q is x and we do not need to check the other points. The correctness is easily verified. Some extentions are described in our another paper: M. Kudo, N. Masuyama and M. Shimbo, "Simple termination conditions for k-nearest neighbor method." *Pattern Recognition Letters*, 24(2003), pp. 1213-1223.

```
⟨ Check of possibility of termination 45 ⟩ ≡
  {
#ifdef DEBUG
    cerr << "In␣>␣Termination␣check␣with␣i␣" << i << endl;
#endif
    if (r2 < TermR2[i]) {
      kcounter[RadiusTermination] += (n − i);    /* Count no examined number */
      term_flag = 2;
    }
  }
```

This code is used in sections 12 and 48.

46. Full distance calculation in the original m-dimension.

```
⟨ Do full distance calculation using raw data 46 ⟩ ≡
#ifdef DEBUG
  cerr << "In␣>␣Full␣distance␣with␣allpca_flag␣" << allpca_flag << endl;
#endif
  if (allpca_flag) r2 = DIS2(px, ptrx[i]);
  else r2 = DIS2(x, trx[i]);
```

This code is used in section 48.

47.

```
⟨ the found kth NN is irregularly large 47 ⟩ ≡
  ((r2 > thresh_l[q]) ? 1 : 0)
```

This code is used in section 12.

48. Full distance calculation in the original m-dimension.

```
⟨ Do full search for every data 48 ⟩ ≡
#ifdef DEBUG
  cerr << "In␣>␣Full␣distance␣for␣every␣data" << endl;
#endif
  for (int i = 0; i < n; i++)    /* Examine ith data */
  {
    ⟨ Do full distance calculation using raw data 46 ⟩
    if (r2 < current_d2) {
      if (tflag) ⟨ Check of possibility of termination 45 ⟩
      ⟨ Update of kNNs 44 ⟩
      if (term_flag ≡ 2) break;
    }
  }
  kcounter[RFullDistance]++;
```

This code is used in section 77.

49. Functions.

```
⟨ Function Declaration 33 ⟩ +≡
  int class_from_num(int ii, int *trnum, int cn)
  {
    int cl = 0;
    int tnum, j;
    int i = ii;
    while ((i -= trnum[cl]) ≥ 0) {
      cl++;
    }
    if (ii < 0 ∨ cl ≥ cn) {
      tnum = 0;
      for (j = 0; j < cn; j++) tnum += trnum[cl];
      fprintf(stderr,
          "*E*␣number␣%d␣is␣over␣%d␣of␣stored␣samples␣cl=%d␣cn=%d␣(trnum[%d]=%d)\n", ii,
          tnum, cl, cn, cl, trnum[cl]);
      fflush(stderr);
      exit(0);
    }
    return cl;
  }
```

50. Majority vote. This function is not used currently. Just for the future use.

```
⟨ Function Declaration 33 ⟩ +≡
  int output(int k, int *knn_id, REAL *knn_dis, int tn)
  {
    static int cc = 0;
    static int *num;
    if (cc ≡ 0) num = new int[nc];
    for (int i = 0; i < nc; i++) num[i] = 0;
    int cl;
    for (int i = 0; i < k; i++) {
      cl = class_from_num(knn_id[i], tr_pc, nc);
      num[cl]++;
    }    /* majority vote */
    int maxc = (−1);
    int maxv = (−1);
    for (int i = 0; i < nc; i++) {
      if (num[i] > maxv) {
        maxv = num[i];
        maxc = i;
      }
    }    /* calc of score for the majority class */
    REALv = 0.0;
    int vc = 0;
    int first;
    for (int i = 0; i < k; i++)
      if (class_from_num(knn_id[i], tr_pc, nc) ≡ maxc) {
        v += knn_dis[i];
        vc++;
      }
    first = knn_id[0];
    if (vc > 0) v /= (REAL)vc;
    if (ocflag) {
      printf("OUT␣␣%d␣%f␣%d\n", maxc, v, first);
      fflush(stdout);
    }
    int tcl;
    if (¬std_in_flag) tcl = class_from_num(tn, te_pc, nc);
    else tcl = (−1);
#ifdef DEBUG
    printf("*␣%s␣*␣Test␣sample␣%d␣(class␣%d)␣is␣assigned␣to␣class␣%d␣with␣value␣%f\n",
        (maxc ≡ tcl) ? "Correct" : "Wrong", tn, tcl, maxc, v);
#endif
    if (maxc ≡ tcl) ncorrect++;
    else nwrong++;
    cc++;
    return maxc;
  }
```

51. Calculation of eigen values and eigen vectors of all data in q dimensions.

⟨ Calculation of eigen vectors in q dimensions using all data 51 ⟩ ≡

```
  {
#ifdef DEBUG
    cout << "In␣calc,␣using␣all␣data,␣find␣|q|␣eigen␣values␣q␣=" << q << endl;
#endif
        /* ——————————————————— */
        /* Mean vector */
        /* ——————————————————— */
    M *= 0.0;
    for (int i = 0; i < n; i++) M += X.Cols(i, i);
    M *= 1.0/(Real)n;
#ifdef DEBUG
    cout << "mean␣M␣was␣calculated" << endl;
#endif
        /* ——————————————————— */
        /* Covariance matrix */
        /* ——————————————————— */
    Cov = X * X.t() * 1.0/(Real)n − M * M.t();
#ifdef DEBUG
    cout << "covariance␣matrix␣Cov␣was␣calculated" << endl;
#endif
    total_variance = Cov.Tr();
    cout << "Total␣variance␣=␣" << total_variance << endl;
#ifdef DEBUG
    cout << "In>␣Cov.Eigen␣" << endl;
#endif
    ratio = Cov.Eigen(Eval, Evec, q);
    n_sample_PCA = n;
  }
```

This code is used in section 26.

52. Calculation of eigen values and eigen vectors of sampled data in q dimensions.

```
⟨ Calculation of eigen vectors in q dimensions using sampled data 52 ⟩ ≡
  {
#ifdef DEBUG
    cout << "In␣calc,␣using␣sampled␣data,␣find␣|q|␣eigen␣values␣q␣=" << q << endl;
#endif
        /* ------------------------------ */
        /* Mean vector */
        /* ------------------------------ */
    M *= 0.0;
    for (int i = 0; i < n_sampling; i++) M += SX.Cols(i, i);
    M *= 1.0/(Real)n_sampling;
#ifdef DEBUG
    cout << "mean␣M␣was␣calculated" << endl;
#endif
        /* ------------------------------ */
        /* Covariance matrix */
        /* ------------------------------ */
    Cov = SX * SX.t() * 1.0/(Real)n_sampling - M * M.t();
#ifdef DEBUG
    cout << "covariance␣matrix␣Cov␣was␣calculated" << endl;
#endif
    total_variance = Cov.Tr();
    cout << "Total␣variance␣=␣" << total_variance << endl;
#ifdef DEBUG
    cout << "In>␣Cov.Eigen␣" << endl;
#endif
    ratio = Cov.Eigen(Eval, Evec, q);
    n_sample_PCA = n_sampling;
  }
```

This code is used in section 26.

53. Calculation of eigen values and eigen vectors of all data in m dimensions.

⟨ Calculation of eigen vectors in m dimensions using all data 53 ⟩ ≡

```
  {
#ifdef DEBUG
    cout << "In␣calc␣|m|␣eigen␣values␣m␣=" << m << endl;
#endif
        /* ------------------------------- */
        /* Mean vector */
        /* ------------------------------- */
    M *= 0.0;
    for (int i = 0; i < n; i++) M += X.Cols(i, i);
    M *= 1.0/(Real)n;
        /* ------------------------------- */
        /* Covariance matrix */
        /* ------------------------------- */
    Cov = X * X.t() * 1.0/(Real)n - M * M.t();
    total_variance = Cov.Tr();
    cout << "Total␣variance␣=␣" << total_variance << endl;
#ifdef DEBUG
    cout << "In>␣Cov.Eigen␣" << endl;
#endif
    ratio = Cov.Eigen(Eval, Evec, m);
    n_sample_PCA = n;
  }
```

This code is used in section 26.

54. Calculation of eigen values and eigen vectors of sampled data in m dimensions.

⟨ Calculation of eigen vectors in m dimensions using sampled data 54 ⟩ ≡

```
  {
#ifdef DEBUG
    cout << "In␣calc␣|m|␣eigen␣values␣m␣=" << m << endl;
#endif
        /* ------------------------------- */
        /* Mean vector */
        /* ------------------------------- */
    M *= 0.0;
    for (int i = 0; i < n_sampling; i++) M += SX.Cols(i, i);
    M *= 1.0/(Real)n_sampling;
        /* ------------------------------- */
        /* Covariance matrix */
        /* ------------------------------- */
    Cov = SX * SX.t() * 1.0/(Real)n_sampling - M * M.t();
    total_variance = Cov.Tr();
    cout << "Total␣variance␣=␣" << total_variance << endl;
#ifdef DEBUG
    cout << "In>␣Cov.Eigen␣" << endl;
#endif
    ratio = Cov.Eigen(Eval, Evec, m);
    n_sample_PCA = n_sampling;
  }
```

This code is used in section 26.

55. Projection of data on the q principal axes to have PX and PSX.

```
⟨ Projection of data on q eigen vectors to obtain PX and PSX 55 ⟩ ≡
  {    /* X: mx(nsample)¬qx */     /* (nsample) */
    PX = Evec.t().Rows(0, q − 1) ∗ X;
    PSX = Evec.t().Rows(0, q − 1) ∗ SX;
  }
```

This code is used in section 26.

56. Projection of all data on the q principal axes.

```
⟨ Projection of all the data on q eigenvectors 56 ⟩ ≡
  {    /* X: mx(nsample)¬qx */
    ptrx = new REAL ∗ [n];
    for (int i = 0; i < n; i++) ptrx[i] = new REAL[q];
    for (int i = 0; i < n; i++)
      for (int j = 0; j < q; j++) ptrx[i][j] = PX(j, i);
  }
```

This code is used in section 63.

57. Projection of all data on the m principal axes.

```
⟨ Projection of all the data on m eigenvectors 57 ⟩ ≡
  {
    PX = Evec.t() ∗ X;
    for (int i = 0; i < n; i++)
      for (int j = 0; j < m; j++) trx[i][j] = PX(j, i);
    ptrx = trx;
  }
```

This code is used in section 63.

58. Memory allocation of terminal radii is firstly made.

```
⟨ Memory allocation of terminal radii 58 ⟩ ≡
  try {
    TermR2 = new REAL[n];
  }
  catch(bad_alloc)
  {
    cerr ≪ "*E*␣cannot␣allocate␣memory␣for␣TermR2␣with␣size␣␣" ≪ n ≪ "␣for␣dimension␣" ≪
        m ≪ endl;
    exit(0);
  }
```

This code is used in section 15.

59. Calculation of Termination Distance.

```
⟨ Calculation of Termination Distance 59 ⟩ ≡
    /* Calculate termination radius of each sample */
    /* and keep it termd2 in square distance */
  {
    cout << "IN␣>␣Termination␣Distance␣" << endl;
    if (nc > 1)    /* multi-class case */
    {
      int pk2 = int(kk/2);
      int nk2 = int((kk + 1)/2);
      int *knn_id = new int[nk2];
      REAL *knn_dis = new REAL[nk2];
      REAL pd2;    /* Squared Distance to kth NN in the same class */
      REAL nd2;    /* Sqaured Distance to kth NN in the different classes */
      REAL dd;
      int icls;
      for (int i = 0; i < n; i++) {    /* The same class */
        if (K > 1) pd2 = DIS2(i, NNhalf[i]);
        else pd2 = 0.0;    /* Different classes */    /* 1st part */
        icls = class_from_num(i, tr_pc, nc);
        stock_k_nn(nk2, 1. · 10^33, 0, knn_id, knn_dis, 0, 1);    /* Initialize */
        for (int jcls = 0; jcls < icls; jcls++) ⟨ Find unlike neighbor 60 ⟩;    /* 2nd part */
        for (int jcls = icls + 1; jcls < nc; jcls++) ⟨ Find unlike neighbor 60 ⟩;
        nd2 = stock_k_nn(nk2, 0.0, 0, knn_id, knn_dis, 1);
        TermR2[i] = 0.25 * (sqrt(nd2) − sqrt(pd2)) * (sqrt(nd2) − sqrt(pd2));
        if (TermR2[i] ≤ 0) {
          cerr << "*Strange*␣nd2=" << nd2 << "␣pd2=" << pd2 << endl;
        }
      }
      delete[] knn_id;
      delete[] knn_dis;
    }
    else    /* Single class */
    {
      for (int i = 0; i < n; i++) TermR2[i] = 0.25 * DIS2(i, NN[i]);
    }
  }
```

This code is used in section 15.

60. Finding of unlike kNNs. We have to find the *unlike nearest neighbor* of every samples. The unlike nearest neighbor of a sample is the nearest neighbor of it belonging to a class different from that of the sample.

```
⟨ Find unlike neighbor 60 ⟩ ≡    /* For class j(≠ i) */
  {
    int st = 0;
    int ed;
    for (int j = 0; j < jcls; j++) st += tr_pc[j];
    ed = st + tr_pc[jcls];
    for (int j = st; j < ed; j++) {
      dd = DIS2(i, j);
      stock_k_nn(nk2, dd, j, knn_id, knn_dis);
    }
  }
```

This code is used in section 59.

61.

```
⟨ Make ready of eigen vectors ev 61 ⟩ ≡
  if (allpca_flag) {
    ev = new REAL *[m];
    for (int i = 0; i < m; i++) {
      ev[i] = new REAL[m];
      for (int j = 0; j < m; j++) ev[i][j] = Evec(j, i);
    }
  }
  else if (q > 0) {
    ev = new REAL *[q];
    for (int i = 0; i < q; i++) {
      ev[i] = new REAL[m];
      for (int j = 0; j < m; j++) ev[i][j] = Evec(j, i);
    }
  }
  px = new REAL[m];
```

This code is used in section 15.

62. Finding of kNNs for all the training points.

⟨For all training samples, find the kNNs 62⟩ ≡

```
{
  REAL pdisq;
  int cc = 0;
  REAL vmin;
  int nmin;
  int *kno = new int[K];
  REAL *kdis = new REAL[K];
  cc = 0;
  for (int i = 0; i < n; i++) {
    find_k_nn_in_training(K, i, n, trx, kno, kdis);
    NN[i] = kno[K - 1];
    if (tflag ∧ K > 1) NNhalf[i] = kno[K/2];
  }
  delete[] kno;
  delete[] kdis;
}
```

This code is used in section 15.

63.

⟨Initialize of total environment 63⟩ ≡

```
if (allpca_flag) ⟨Projection of all the data on m eigenvectors 57⟩
else if (q > 0) ⟨Projection of all the data on q eigenvectors 56⟩
```

This code is used in section 15.

64. Obtain a sample matrix SX using sampled points.

⟨Set up of sample matrix X and sampling matrix SX 64⟩ ≡

```
{
  ColumnVector x(m);
  int cc = 0;
#ifdef DEBUG
  cout << "X:␣rows=" << X.Nrows() << "␣cols=" << X.Ncols() << endl;
  cout << "SX:␣rows=" << SX.Nrows() << "␣cols=" << SX.Ncols() << endl;
#endif
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) x(j) = trx[i][j];
    X.SetCols(i, i, x);
    if ((i % sampling_skip) ≡ 0 ∧ cc < n_sampling) {
      SX.SetCols(cc, cc, x);
      cc++;
    }
  }
  if (cc ≠ n_sampling) {
    cerr << "*Sampling␣is␣strange␣in␣SX␣setting" << endl;
    cerr << "cc␣=␣" << cc << "␣n_sampling␣=␣" << n_sampling << endl;
    exit(0);
  }
}
```

This code is used in section 15.

65.

```
⟨ Set up of sampling parameters 65 ⟩ ≡
  {
    n_sampling = min(n, sampling_ceil);
    int on_sampling = n_sampling;
    sampling_skip = 1;
    if (n > n_sampling)
      sampling_skip = (n % n_sampling ≡ 0) ? n/n_sampling : int(REAL(n)/REAL(n_sampling));
    if (n % sampling_skip ≠ 0) n_sampling = n/sampling_skip + 1;
    printf("For␣n=%d␣and␣n_sampling␣=%d␣...␣sampling_skip␣=%d␣n_sampling␣=%d\n", n,
        on_sampling, sampling_skip, n_sampling);
    n_tail = int(tail_percentage * 0.01 * n_sampling) + 1;
    n_tail = min(n_tail, n);
    cout ≪ "Set␣of␣sampling␣n_sampling␣=␣" ≪ n_sampling ≪ "␣with␣sampling_skip␣=␣" ≪
        sampling_skip ≪ endl;
    cout ≪ "n_sampling=␣" ≪ n_sampling ≪ endl;
    Matrix SX(m, n_sampling + 1);
    Matrix PSX(q, n_sampling + 1);     /* projected SX */
  }
```

This code is used in section 22.

66.

⟨ Output of distribution parameters 66 ⟩ ≡

```
{
  fprintf(fp, "#␣Version:␣%s\n", ver);
  fprintf(fp, "#␣Data␣parameter␣n,␣m\n");
  fprintf(fp, "%d␣%d\n", n, m);
  fprintf(fp, "#␣parameter␣q\n");
  fprintf(fp, "%d\n", q);
  fprintf(fp, "#␣flag␣parameters:␣allpca_flag,␣pflag,␣tflag,␣rflag\n");
  fprintf(fp, "%1d␣%1d␣%1d␣%1d\n", allpca_flag, pflag, tflag, rflag);
  fprintf(fp, "#␣thresholds,␣error_level,␣gain_level\n");
  int pp;
  for (int i = 0; i < n_error_level; i++) {
    fprintf(fp, "%f␣", error_level[i]);
    for (int j = 0; j ≤ q; j++) fprintf(fp, "%15.12f␣%15.12f␣", threshold[i][j], gain[i][j]);
    fprintf(fp, "\n");
  }
  if (allpca_flag) {
    fprintf(fp, "#␣Number␣of␣eigen␣vectors\n");
    fprintf(fp, "%d\n", m);
    fprintf(fp, "#␣Eigen␣vectors\n");
    for (int i = 0; i < m; i++) {
      for (int j = 0; j < m; j++) fprintf(fp, "%f␣", ev[i][j]);
      fprintf(fp, "\n");
    }
    fprintf(fp, "#␣Transformed␣data\n");
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) fprintf(fp, "%f␣", ptrx[i][j]);
      fprintf(fp, "\n");
    }
  }
  else {
    fprintf(fp, "#␣Number␣of␣eigen␣vectors\n");
    fprintf(fp, "%d\n", q);
    fprintf(fp, "#␣Eigen␣vectors\n");
    for (int i = 0; i < q; i++) {
      for (int j = 0; j < m; j++) fprintf(fp, "%f␣", ev[i][j]);
      fprintf(fp, "\n");
    }
    fprintf(fp, "#␣Transformed␣data\n");
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < q; j++) fprintf(fp, "%f␣", ptrx[i][j]);
      fprintf(fp, "\n");
    }
  }
  if (tflag) {
    fprintf(fp, "#␣Termination␣radius\n");
    for (int i = 0; i < n; i++) fprintf(fp, "%f\n", TermR2[i]);
  }
  ⟨ Estimation of time coefficients 71 ⟩
  fprintf(fp, "#␣Estimated␣execution␣costs:␣c1&c2=␣%f␣%f\n", c1, c2);
```

```
  }
```

This code is used in sections 24 and 25.

67.

⟨ Set up of thresholds 67 ⟩ ≡

```
  {
    int pp;
    for (int i = 1; i < n_error_level; i++) {
      pp = int(n_sampling * error_level[i] * 0.01);
      for (int k = 0; k <= q; k++) {
        threshold[i][k] = P(pp, k);
        gain[i][k] = XNtail(pp, k);
        printf("[Set up of threholds] threshold[%d][%d]=%e gain[%d][%d]=%e\n", i, k,
            threshold[i][k], i, k, gain[i][k]);
      }
    }
  }
```

This code is used in section 15.

68.

⟨ Selection of q and θ_q 68 ⟩ ≡

```
{    /* User selection of error probability */
  cout ≪ "␣---␣User␣Selection␣of␣Error␣Probability␣---" ≪ endl;
  cout ≪ "\t\t␣Expected␣Distance␣Reduction␣Rate\n";
  cout ≪ "No.␣␣Error(\%)␣\t";
  for (int k = 0; k < q; k++) cout ≪ "␣␣␣q=␣␣" ≪ k + 1 ≪ "\t";
  cout ≪ "␣␣␣q=␣␣" ≪ m ≪ "\t";
  cout ≪ endl;
  int *opt_q = new int[n_error_level];
  int pp;
  for (int i = 1; i < n_error_level; i++) {
    pp = int(n_sampling * error_level[i] * 0.01);
    printf("%1d␣-␣%f␣:\t", i, error_level[i]);
    for (int k = 0; k ≤ q; k++) printf("%7.3f␣\t", gain[i][k]);
    printf("\n");
    int mk;
    REAL rk = 1. · 10^33;
    REAL r;
    for (int k = 0; k < q; k++) {
      r = estimated_reduction(n, m, k + 1, gain[i][k]);
      if (r < rk) {
        mk = k;
        opt_q[i] = mk;
        rk = r;
      }
    }
    printf("Predic,␣R-rate\t");
    for (int k = 0; k ≤ q; k++)
      printf("%7.3f%c␣\t", estimated_reduction(n, m, k + 1, gain[i][k]), (k ≡ mk) ? '*' : '␣');
        /* mk is the optimal q */
    printf("\n");
  }
  int select_level;
  if (A < 0) {
    cout ≪ "␣-␣Choose␣one␣by␣the␣integer␣from␣0:␣no␣threshold,␣1-" ≪ n_error_level − 1 ≪
        "␣in␣above␣table" ≪ endl;
    cin ≫ select_level;
    while (select_level < 0 ∨ select_level ≥ n_error_level) {
      cout ≪ "␣-␣*Warning*␣Rechoose␣one␣by␣the␣integer␣from␣0:␣no␣threshold,␣1-" ≪
          n_error_level − 1 ≪ "␣in␣above␣table" ≪ endl;
      cin ≫ select_level;
    }
  }
  else select_level = A;
  cout ≪ "␣-----␣(␣Selection␣of␣epsion␣and␣marginal␣dimension␣)␣with␣\
      user-specified␣A=␣" ≪ A ≪ "␣----␣" ≪ endl;
  if (narrow ≡ Nothing ∨ narrow ≡ Filtering)
    cout ≪ "1)␣Error␣rate␣=␣" ≪ error_level[select_level] ≪ "␣%␣" ≪ endl;
  else cout ≪ "1)␣Error␣rate␣+1␣0.1%%␣error=␣" ≪ error_level[select_level] + 0.1 ≪ "␣%␣" ≪ endl;
      /* Setting of optimal q (l_opt) */
```

```
if (qsdim ≡ 0) q = opt_q[select_level] + 1;
else q = qsdim;
cout ≪ "2)␣Marginal␣dimension␣=␣" ≪ q ≪ endl;
pp = int(n_sampling * error_level[select_level] * 0.01);
for (int i = 0; i ≤ q; i++) {
  if (select_level ≡ 0) {
    thresh_l[i] = 1. · 10^33;
    q = 0;
    cout ≪ "**␣q␣is␣reset␣to␣0␣" ≪ endl;
  }
  else thresh_l[i] = threshold[select_level][i];
}
if (A > 0) thresh_1st = threshold[1][0];    /* Plus 0.1% error */
else thresh_1st = 1. · 10^33;
cout ≪ "\t␣thresh_1st␣was␣set␣to␣" ≪ thresh_1st ≪ "␣by␣threshold[1][0]" ≪ endl;
double delta;
if (q > 0) delta = gain[select_level][q − 1];
else delta = 0.0;
cout ≪ "3)␣Threshold␣=␣" ≪ threshold[select_level][q − 1] ≪ "␣with␣" ≪ pp ≪ "␣tail␣samples" ≪
    endl;
cout ≪ "4)␣Estimated␣probability␣of␣full␣distance␣=␣" ≪ (1.0 − delta) * 100.0 ≪ "␣%" ≪ endl;
cout ≪ "5)␣Estimated␣time␣reduction␣rate␣=␣" ≪ estimated_reduction(n, m, q + 1, delta) ≪ endl;
cout ≪ "␣---------------------------------------------␣" ≪ endl;
}
```

This code is used in section 11.

69.

⟨ Input of distribution parameters 69 ⟩ ≡

```
{
  int dummy;
  static char *s = new char[512];
  char iver[256];
  fscanf(fp, "#␣Version:␣%s\n", &iver);
  if (strncmp(ver, iver, 5) ≠ 0) {
    fprintf(stdout,
        "(Error)␣Version␣does␣not␣mactch:␣expected␣version␣=␣%s␣for␣model␣version␣%s\n",
        ver, iver);
    exit(0);
  }
  fscanf(fp, "#␣Data␣parameter␣n,␣m\n");
  fscanf(fp, "%d␣%d\n", &in, &im);
  fscanf(fp, "#␣parameter␣q\n");
  fscanf(fp, "%d\n", &iq);
  fscanf(fp, "#␣flag␣parameters:␣allpca_flag,␣pflag,␣tflag,␣rflag\n");
  fscanf(fp, "%1d␣%1d␣%1d␣%1d\n", &iallpca_flag, &ipflag, &itflag, &irflag);
  fscanf(fp, "#␣thresholds,␣error_level,␣gain_level\n");
  if (n ≠ in ∨ m ≠ im) {
    fprintf(stdout, "(Error)␣Input␣file␣does␣not␣match␣:␣n=%d␣m=%d␣vs.␣in=%d␣im=%d\n", n, m,
        in, im);
    exit(0);
  }
  if (iq > q) {
    for (int i = 0; i < n_error_level; i++) {
      gain[i] = new REAL[iq + 1];
      threshold[i] = new REAL[iq + 1];
      gain[0][i] = 0.0;
    }
    for (int i = 0; i < n; i++) {
      ptrx[i] = new REAL[iq];
    }
    ev = new REAL *[iq];
    for (int i = 0; i < iq; i++) ev[i] = new REAL[m];
  }
  int pp;
  REALa, b;
  for (int i = 0; i < n_error_level; i++) {
    if (sizeof (REAL) ≡ sizeof(double)) fscanf(fp, "%lf␣", &a);
    else fscanf(fp, "%f␣", &a);
    for (int j = 0; j ≤ iq; j++) {
      if (sizeof (REAL) ≡ sizeof(double)) fscanf(fp, "%lf␣%lf␣", &threshold[i][j], &gain[i][j]);
      else fscanf(fp, "%f␣%f␣", &threshold[i][j], &gain[i][j]);
    }
    fscanf(fp, "\n");
  }
  if (allpca_flag) {
    fscanf(fp, "#␣Number␣of␣eigen␣vectors\n");
    fscanf(fp, "%d\n", &m);
```

```
    fscanf(fp, "#␣Eigen␣vectors\n");
    for (int i = 0; i < m; i++) {
      for (int j = 0; j < m; j++)
        if (sizeof (REAL) ≡ sizeof(double)) fscanf(fp, "%lf␣", &ev[i][j]);
        else fscanf(fp, "%f␣", &ev[i][j]);
      fscanf(fp, "\n");
    }
    fscanf(fp, "#␣Transformed␣data\n");
    int i;
    for (int ii = 0; ii < n; ii++) {
      i = iri[ii];
      for (int j = 0; j < m; j++)
        if (sizeof (REAL) ≡ sizeof(double)) fscanf(fp, "%lf␣", &ptrx[i][j]);
        else fscanf(fp, "%f␣", &ptrx[i][j]);
      fscanf(fp, "\n");
    }
  }
  else {
    fscanf(fp, "#␣Number␣of␣eigen␣vectors\n");
    fscanf(fp, "%d\n", &q);
    fscanf(fp, "#␣Eigen␣vectors\n");
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++)
        if (sizeof (REAL) ≡ sizeof(double)) fscanf(fp, "%lf␣", &ev[i][j]);
        else fscanf(fp, "%f␣", &ev[i][j]);
      fscanf(fp, "\n");
    }
    fscanf(fp, "#␣Transformed␣data\n");
    int i;
    for (int ii = 0; ii < n; ii++) {
      i = iri[ii];
      for (int j = 0; j < iq; j++)
        if (sizeof (REAL) ≡ sizeof(double)) fscanf(fp, "%lf␣", &ptrx[i][j]);
        else fscanf(fp, "%f␣", &ptrx[i][j]);
      fscanf(fp, "\n");
    }
  }
  if (tflag) {
    fscanf(fp, "#␣Termination␣radius\n");
    int i;
    for (int ii = 0; ii < n; ii++) {
      i = iri[ii];
      if (sizeof (REAL) ≡ sizeof(double)) fscanf(fp, "%lf\n", &TermR2[i]);
      else fscanf(fp, "%f\n", &TermR2[i]);
    }
  }
  if (sizeof (REAL) ≡ sizeof(double))
    fscanf(fp, "#␣Estimated␣execution␣costs:␣c1&c2=␣%lf␣%lf\n", &c1, &c2);
  else fscanf(fp, "#␣Estimated␣execution␣costs:␣c1&c2=␣%f␣%f\n", &c1, &c2);
  printf("#␣Estimated␣execution␣costs:␣c1&c2=␣%f␣%f\n", c1, c2);
}
```

This code is used in section 24.

70.

```
⟨ Parameters for Initializing of total environment 70 ⟩ ≡
  int nkind = 5;
  enum kind {
    FullDistance, MarginalDistance, PartialDistance, RadiusTermination, RFullDistance
  };
  const char *kindname[] = {"Full Distance", "Marginal Distance", "Partial Distance",
      "Radius Termination", "Recovery FullDist"};
  long *kcounter = new long[nkind];
  unsigned long *examined_dim = new unsigned long[m];
  for (int i = 0; i < nkind; i++) kcounter[i] = 0;
  for (int i = 0; i < m; i++) examined_dim[i] = 0;
  int term_flag;   /* termination flag */
  REAL r2;   /* sqaured distance */
  int checked_sample = 0;
  int edim;   /* examined dimension */
  REAL current_d2;
  int isample;
  REAL theta_q;
  int kNN;
  ncorrect = 0;
  nwrong = 0;
```

This code is used in section 15.

71. Utility functions.

⟨ Estimation of time coefficients 71 ⟩ ≡

```
{
  time_t t1, t2, tt;
  time_t st, ed;
  int N_time = 1000;
  cout << "Now measuring actual time of distance calc and inner produc\
      t ..it will take about 40 seconds" << endl;
    /* Get random vectors; */
  ColumnVector u(m);
  ColumnVector v(m);
  RandomMatrix U(m, 1);
  u << U;
  RandomMatrix V(m, 1);
  v << V;     /* Time for distance calculation in full dimensions */
  int R_time = 0;
  t1 = 0;
  st = (unsigned) time(Λ);
  while (t1 < 20) {
    for (int i = 0; i < N_time; i++) DIS2(u, v);
    ed = (unsigned) time(Λ);
    t1 = ed − st;
    R_time++;
  }     /* Time for dot-product calculation in full dimensions */
  st = (unsigned) time(Λ);
  for (int r = 0; r < R_time; r++)
    for (int i = 0; i < N_time; i++) innerp(u, v);
  ed = (unsigned) time(Λ);
  t2 = ed − st;
  c1 = (REAL)t1;
  c2 = (REAL)t2;
  printf("In estimation of time coefficients: c1=%f (t1=%d) for dist calc. and c2=%f (t2=\
      %d) for inner product with N_time %d x R_time %d\n", c1, t1, c2, t2, N_time, R_time);
  printf("** Be careful, time %d was consumed in this estimation\n", t2 + t2);
}
```

This code is used in section 66.

72.

⟨ Function Declaration 33 ⟩ +≡

```
REAL estimated_reduction(int n, int m, int l, REAL delta)
{
  REAL r;
  r = 1.0 − delta + REAL(l)/m + c2 * REAL(l)/c1/REAL(n);
  return r;
}
```

73.

⟨ Counter of marginal distance strategy 73 ⟩ ≡

```
#ifdef COUNTER
  kcounter[MarginalDistance]++;
  examined_dim[edim]++;
#endif
```

This code is used in section 12.

74.

⟨ Counter of 1-dim marginal distance strategy 74 ⟩ ≡

```
#ifdef COUNTER
  kcounter[MarginalDistance] += (n − (int_ed − int_st + 1));
  examined_dim[0] += (n − (int_ed − int_st + 1));
#endif
```

This code is used in section 12.

75.

⟨ Counter of partial distance strategy 75 ⟩ ≡

```
#ifdef COUNTER
  kcounter[PartialDistance]++;
  examined_dim[edim]++;
#endif
```

This code is used in section 12.

76.

⟨ Counter of full distance strategy 76 ⟩ ≡

```
#ifdef COUNTER
  kcounter[FullDistance]++;
  examined_dim[m − 1]++;
#endif
```

This code is used in sections 12 and 77.

77.

⟨ Recovery process with full distance search 77 ⟩ ≡

```
  {
    ⟨ Do full search for every data 48 ⟩
    r2 = stock_k_nn(K, 0.0, 0, knn_id, knn_dis, 1);
    ⟨ Counter of full distance strategy 76 ⟩;
  }
```

This code is used in section 12.

78. Memory release.

⟨ Memory release of no more necessary variavles 78 ⟩ ≡

```
X.Release();
SX.Release();
M.Release();
Cov.Release();
Eval.Release();
PV.Release();
PX.Release();
```

This code is used in section 11.

79. Sorting of stored data in the first principal axis.

⟨ Sorting of data in the first principal axis 79 ⟩ ≡

```
{   /* sort ptrx in the increasing order of the 1st axis and store the order into an array 1stord. */
    for (int i = 0; i < n; i++) {
      pair[i].id = i;
      pair[i].val = ptrx[i][0];
    }
    qsort(pair, n, sizeof(CELL), comp_score);
    for (int i = 0; i < n; i++) pair[pair[i].id].rid = i;
#ifdef DEBUG
    printf("Sorting result  ... of %d data in 1st principal axis\n", n);
    for (int i = 0; i < n; i++) {
      printf("(i=%d, id=%d, rid=%d, %lf)\n", i, pair[i].id, pair[i].rid, pair[i].val);
      if (i % 10 == 9) printf("\n");
    }
#endif
}
```

This code is used in sections 11 and 84.

80. Determine the interval on the first principal axis.

⟨ Determine the interval on the first principal axis 80 ⟩ ≡

```
{
  REAL th1st = sqrt(thresh_1st);
  REAL q1st = px[0];
  REAL qst = q1st − th1st;
  REAL qed = q1st + th1st;
  int_st = 0;
  int_ed = n − 1;    /* Find inst_st close to qst by a binary search */
  int pl = 0;
  int pr = n − 1;
  int p = (pl + pr)/2;
  REAL v;
  while (pr > (pl + 1)) {
    v = pair[p].val;
    if (v < qst) pl = p;
    if (v ≥ qst) pr = p;
    p = (pl + pr)/2;
  }
  int_st = (pl > 0) ? pl − 1 : 0;    /* Find inst_st close to qst by a binary search */
  pl = int_st;
  pr = n − 1;
  p = (pl + pr)/2;
  while (pr > (pl + 1)) {
    v = pair[p].val;
    if (v < qed) pl = p;
    if (v ≥ qed) pr = p;
    p = (pl + pr)/2;
  }
  int_ed = (pr < n) ? pr : n − 1;
}
```

This code is used in section 12.

81. Data sorting on the basis of 1st principal dimension.

⟨ Data sorting on the basis of 1st principal dimension 81 ⟩ ≡

```
  {
    REAL **trxd = new REAL *[n];
    for (int i = 0; i < n; i++) trxd[i] = new REAL[m];
    REAL **ptrxd;
    ptrxd = new REAL *[n];
    if (allpca_flag ≡ 0)
      for (int i = 0; i < n; i++) ptrxd[i] = new REAL[q];
    else
      for (int i = 0; i < n; i++) ptrxd[i] = new REAL[m];
    REAL *pp, *pq;
    int target;
#ifdef DEBUG
    for (int i = 0; i < 100; i++) {
      printf("(%d,␣%lf,␣%lf)␣", i, ptrx[i][0], trx[i][0]);
      if (i % 10 ≡ 9) printf("\n");
    }
#endif
    printf("---␣Sort␣of␣given␣data␣in␣the␣order␣of␣1st␣principal␣axis\n");
    for (int i = 0; i < n; i++) {
      target = pair[i].id;
      if (i % 100 ≡ 99) printf("Data␣copy␣from␣%d␣->␣%d\n", target, i);
      pp = trxd[i];
      pq = trx[target];
      for (int j = 0; j < m; j++) *pp++ = *pq++;
      pp = ptrxd[i];
      pq = ptrx[target];
      if (allpca_flag ≡ 0)
        for (int j = 0; j < q; j++) *pp++ = *pq++;
      else
        for (int j = 0; j < m; j++) *pp++ = *pq++;
    }
    for (int i = 0; i < n; i++) {
      delete[] trx[i];
      if (¬allpca_flag) delete[] ptrx[i];
    }
    delete[] trx;
    if (¬allpca_flag) delete[] ptrx;
    trx = trxd;
    ptrx = ptrxd;
#ifdef DEBUG
    printf("---␣Sorted␣data␣(top␣100)\n");
    for (int i = 0; i < 100; i++) {
      printf("(%d,␣%lf,␣%lf)␣", i, ptrx[i][0], trx[i][0]);
      if (i % 10 ≡ 9) printf("\n");
    }
#endif
  }
```

This code is used in section 11.

82. Establish a data structure for a serach of $O(1)$. In the following, we divide the interval given by the mimimum value *sample_min* and maximal value *samle_max* of data into *sample_max* − *sample_min*/(2*w*) small cells in which the starting data number is kept. The width *w* is taken the distance of search.

⟨ Establish a data structure for a serach of $O(1)$ 82 ⟩ ≡

```
{
  th1st = sqrt(thresh_1st);
  sample_min = pair[0].val;
  sample_max = pair[n − 1].val;
  W = sample_max − sample_min;
  int BB = B_default ∗ int(W/th1st);
  if (BB > B) B = BB;
  w = W/REAL(B);
  printf("Establish␣of␣data␣structure:␣n=%d␣W=%f␣B=%d␣w=%f␣th=%f␣in␣[%f␣-␣%f]\n", n, W, B,
      w, th1st, sample_min, sample_max);
  iw = int(th1st/w) + 1;
  REAL stv = pair[0].val + w;
  start_data_no = new int[B + 2];
  start_data_no[0] = 0;
  int next = 1;
  for (int i = 1; i < n; i++)
    while (pair[i].val > stv) {
      start_data_no[next] = i;
      next++;
      stv += w;
    }
  start_data_no[next] = n − 1;
}
```

This code is used in section 11.

83. Determine a small data subset (index subset) including the query data.

⟨ Determine a small subset including the query point 83 ⟩ ≡

```
{
  REAL q1st = px[0];
  int ind = int((q1st − sample_min)/w);
  int_st = ((ind − iw) ≥ 0) ? ind − iw : 0;
  int_ed = ((ind + iw + 1) < B) ? ind + iw + 1 : B;
  int_st = start_data_no[int_st];
  int_ed = start_data_no[int_ed];
#ifdef DEBUN
  printf("Found␣subset␣starts␣for␣q=%f␣(ind=%d␣B=%d,␣w=%f,␣iw=%d)␣at␣%d␣and␣ends␣at␣%d␣in\
      ␣(%f␣-␣%f)␣for␣th=%f\n", q1st, ind, B, w, iw, int_st, int_ed, pair[int_st].val, pair[int_ed].val,
      th1st);
#endif
}
```

This code is used in section 12.

84.

⟨ Output of sorted data to a file 84 ⟩ ≡

```
{
  FILE *fp;
  char ofile[256];
  sprintf(ofile, "%s_sorted_index", out_file);
  fp = fopen(ofile, "w");
  ⟨ Sorting of data in the first principal axis 79 ⟩
  fprintf(fp, "n= %d m= %d\n", n, m);
  for (int i = 0; i < n; i++) fprintf(fp, "%d %d %f\n", pair[i].rid, pair[i].id, pair[i].val);
  fclose(fp);
}
```

This code is used in section 11.

85.

⟨ Input of sorted data from a file 85 ⟩ ≡

```
{
  FILE *fp;
  char ifile[256];
  sprintf(ifile, "%s_sorted_index", in_file);
  if ((fp = fopen(ifile, "r")) ≡ Λ) {
    fprintf(stderr, " * file <%s> open error * \n", ifile);
    exit(0);
  }
  fscanf(fp, "n= %d m= %d\n", &n, &m);
  printf("From Indexfile <%s>: n=%d m=%d\n", ifile, n, m);
  fflush(stdout);
  pair = new CELL[n];
  for (int i = 0; i < n; i++) fscanf(fp, "%d %d %lf\n", &pair[i].rid, &pair[i].id, &pair[i].val);
  fclose(fp);
}
```

This code is used in section 11.

The kNN_MDS program

(Version 1.3)