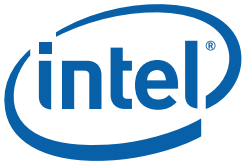


Intel® Summary Statistics Library

Application Notes

Revision: 1.1

World Wide Web: http://www.intel.com

Disclaimer and Legal Information

Revision History

Revision Number	Description	Revision Date
1.1	Initial release.	April 2009

Contents

1 About this Document

Intel® Summary Statistics Library is a solution for parallel statistical processing of multi-dimensional datasets. The Intel® Summary Statistics Library contains functions for initial statistical analysis of raw data, which enables you to investigate the structure of datasets and understand their basic characteristics, estimates, and internal dependencies.

These Application Notes are intended to show you how to use the Intel® Summary Statistics Library when you work on your applications. This document covers the usage model of the library, the most important features and performance aspects. Additional information about the algorithms, interfaces, and supported languages is available in the library User Manual. Building an application that uses the algorithms of the Intel® Summary Statistics Library and the linkage model are considered in the User Guide.

1.1 Conventions and Symbols

The following conventions are used in this document.

Table 1 Conventions and Symbols used in this Document

`This type style`	Indicates an element of syntax, parameter name, keyword, filename, computer output, or part of a program example. The text appears in lowercase unless uppercase is significant.
This type style	Indicates the exact characters you type as input. Also used to highlight the elements of a graphical user interface such as buttons and menu names.
This type style	Indicates a placeholder for an identifier, an expression, a string, a symbol, or a value. Substitute one of these items for the placeholder.
[*items*]	Indicates that the items enclosed in brackets are optional.
{ *item* | *item* }	Indicates to select only one of the items listed between braces. A vertical bar (|) separates the items.
… (ellipses)	Indicates that you can repeat the preceding item.

2 Several Estimates at One Stroke

Let us consider a problem of computing statistical estimates for a dataset. Suppose the observations are weighted and only several components of the random vector have to be analyzed. Use the Intel® Summary Statistics Library to solve such problems.

Any typical application that uses the library passes through four stages. All the stages are considered below in greater detail using a simple example for computation of the mean, variance, covariance, and the variation coefficient.

First, we create a new task and pass into the library the parameters of the problem, dimension *p*, the number of observations *n*, and a pointer to the memory where the dataset *X* is stored:

```
storage_format_x   = VSL_SS_MATRIX_COLUMNS_STORAGE;
errcode = vsldSSNewTask( &task, &p, &n,  &storage_format_x, X,
weights, indices );
```

The array `weights` contains weights assigned to each observation, and the array `indices` determines components of the random vector to be analyzed. For example, `indices` can be initialized as follows:

```
indices[p] = {0, 1, 1, 0, 1,..};
```

That means observations for the zero and third components of the random vector are excluded from the analysis. The dataset can be stored in columns or in rows; its storage format is passed into the library using the `storage_format_x` variable. By the way, you can pass the `NULL` pointers instead of `weights` and `indices` if you need to set all weights to one and to process all components of the random vector.

Further, we need to register arrays to hold computation results and other parameters in the library. The Intel® Summary Statistics Library provides a set of editors. We use some of them in the example below:

```
errcode = vsldSSEditTask( task, VSL_SS_ACCUMULATED_WEIGHT, W );
errcode = vsldSSEditTask( task, VSL_SS_VARIATION_ARRAY, Variation );
errcode = vsldSSEditMoments( task, Xmean, Raw2Mom,0,0,Central2Mom, 0, 0);
```

```
cov_storage = VSL_SS_MATRIX_FULL_STORAGE;
errcode = vsldSSEditCovCor( task, Xmean, Cov, &Cov_storage , 0, 0);
```

Estimates for the mean, 2nd algebraic moment, variance, and the variation coefficient are stored in the arrays `Xmean`, `Raw2Mom`, `Central2Mom`, `Variation`, respectively. The covariance estimate is placed in the array `Cov`. We also should set storage format for the covariance matrix as the library supports full and packed format and needs to "know" how to store the matrix. Registration of an array of means is required in most cases even if we do not need to know the estimate. This is necessary as many other statistical estimates use the mean value (see section 6.1 of the User Manual [1] for necessary details).

Now we can compute the estimates of our interest. It is enough to call the computing routine just once:

```
errcode = vsldSSCompute( task,VSL_SS_MEAN | VSL_SS_2CENTRAL_MOMENT |
VSL_SS_COVARIANCE_MATRIX | VSL_SS_VARIATION, VSL_SS_FAST_METHOD );
```

Note that the library expects just a pointer to the memory with the dataset. This allows placing another data to the same memory and calling the `Compute` routine without any need for editing the task descriptor once again.

Finally, task resources are de-allocated:

```
errcode = vslSSDeleteTask( &task );
```

Solution of the problem described above with p = 500 and n = 100,000 took 1.42 seconds on a two-way Quad Core Intel® Xeon® E5440 2.8GHz CPU (8 cores total) based system. Run of this application on the same machine in serial mode (using only one core) took 9.09 seconds.

3 How to Process Data in Chunks?

This chapter considers the problem of computation of statistical estimates for out-of-memory datasets using tools available in the Intel® Summary Statistics Library. For simplicity, we assume that we need to compute the same estimates as in Chapter 2 for a dataset that cannot fit into the memory of a computer. To process the data, we split it into chunks (block analysis of data is also possible for in-memory data arrays that are not available at once). This is a typical example of the analysis where the Intel® Summary Statistics Library can help. Besides, you can easily tune the application for out-of-memory data support.

As described in Chapter 2, our application should follow the same four stages for the library usage. However, we need to do an additional initialization to support a huge dataset. First of all, we set the estimates of our interest to zero (or to any other meaningful value):

```
for( i = 0; i < p; i++ )
{
     Xmean[i] = 0.0;
     Raw2Mom[i] = 0.0;
     Central2Mom[i] = 0.0;
     for(j = 0; j < p; j++)
     {
         Cov[i][j] = 0.0;
     }
}
```

Then we initialize the array *W* of size 2 with zero values. This array will hold accumulated weights that are important for correct computation of the estimates:

```
W[0] = 0.0; W[1] = 0.0;
```

At the next step, we get the first portion of the dataset into the array *X* and weights that correspond to those observations into the array *weight*:

```
GetNextDataChunk( X, weights );
```

Further steps are similar to those described in Chapter 2: creation of the task, editing its parameters, computing necessary estimates, and de-allocating the task resources:

```
/* Creation of task */
storage_format_x   = VSL_SS_MATRIX_COLUMNS_STORAGE;
errcode = vsldSSNewTask( &task, &p, &n_portion,  &storage_format_x, X, weights,
indices );

/* Edition of task parameters */
errcode = vsldSSEditTask( task, VSL_SS_ACCUMULATED_WEIGHT, W );
errcode = vsldSSEditTask( task, VSL_SS_VARIATION_ARRAY, Variation );
errcode = vsldSSEditMoments( task, Xmean, Raw2Mom, 0, 0, Central2Mom, 0, 0 );

Cov_storage = VSL_SS_MATRIX_FULL_STORAGE;
errcode = vsldSSEditCovCor( task, Xmean, (double*)Cov, &Cov_storage, 0, 0 );

/* Computation of estimates for dataset split in chunks */
for( nchunk = 0; ; nchunk++ )
{
     errcode = vsldSSCompute( task,
   VSL_SS_MEAN | VSL_SS_2CENTRAL_MOMENT |
   VSL_SS_COVARIANCE_MATRIX |  VSL_SS_VARIATION,
                              VSL_SS_1PASS_METHOD );
     If ( nchunk >= N ) break;
     GetNextDataChunk( X, weights );
}

/* De-allocation of task resources */
errcode = vslSSDeleteTask( &task );
```

The library also provides an opportunity to read the next data block into a different array. The whole computation scheme remains the same. We just need to “communicate” the address of this data block to the library:

```
for( nchunk = 0; ; nchunk++ )
{
     errcode = vsldSSCompute( task, VSL_SS_MEAN | VSL_SS_2CENTRAL_MOMENT |
VSL_SS_COVARIANCE_MATRIX | VSL_SS_VARIATION,VSL_SS_1PASS_METHOD );

     If ( nchunk >= N ) break;
     GetNextDataChunk( NextXChunk, weights );
```

```
    errcode = vsldSSEditTask( task, VSL_SS_OBSERVATIONS, NextXChunk );
}
```

Nothing else is required.

Table 5 in Section 4 of the User Manual [1] lists estimators that support processing datasets in blocks.

4 How to Detect Outliers in Datasets?

Chapter 3 describes computations of various statistical estimates like the mean or a variance-covariance matrix using the Intel® Summary Statistics Library. In the examples considered, datasets did not contain “bad” observations (points that do not belong to the distribution that is the subject of the analysis), or outliers. However, in some cases such outliers contaminate datasets. Sometimes this happens because process of data collection is not very reliable, as in the case of using microarray technologies for measurement of gene expression levels. In other cases, presence of outliers in datasets is a result of intentional actions, like network intrusion. Anyway, outliers in datasets can result in biased estimates and wrong conclusions about the object. How to deal with such datasets? Use an outlier detection tool from the Intel® Summary Statistics Library.

To see how this tool works, we generate a dataset from a multivariate Gaussian distribution using a corresponding generator available in the Intel® Math Kernel Library. Some of the observations are then replaced with the points/”outliers” from the multivariate Gaussian distribution that has significantly bigger mathematical expectation. The number of outliers is ~20%. Let us see how the BACON outlier detection algorithm available in the Intel® Summary Statistics Library identifies the outliers.

We consider the most important elements for the outlier detection method. Before using the algorithm, we should initialize its parameters. First, we define the initialization scheme of the algorithm. The library gives two options: Median- and Mahalanobis distance-based schemes. Rejection level alpha and stopping criteria level beta (details on the parameters are provided in Section 6.6 of the User Manual [1]) are also defined. The parameters are initialized as shown in the code below:

```
init_method = VSL_SS_BACON_MEDIAN_INIT_METHOD;
alpha = 0.05;
beta  = 0.005;
BaconN = 3;
BaconParams[0] = init_method;
BaconParams[1] = alpha;
BaconParams[2] = beta;
```

The parameters are passed into the library using a suitable editor as shown below:

```
errcode=vsldSSEditOutliersDetection( task, &BaconN, BaconParams, BaconWeights );
```

What is the `BaconWeights` parameter? This is an array of weights that will hold output of the algorithm and point at suspicious observations. Size of the array equals to the number of points. Zero value in the *i*-th position of the array indicates that the *i*-th observation requires special attention, and one is the indicator that the observation is “good”.

The next step is obvious – to call the *`Compute`* function:

```
errcode = vsldSSCompute( task, VSL_SS_OUTLIERS_DETECTION,
          VSL_SS_BACON_METHOD );
```

Upon completion of the computations, the array `BaconWeights` contains weights of the observations that have to be analyzed. This array can be used in further data processing; it is enough to register it as an array of observation weights and to use in the usual manner. Expectedly, after removal of the outliers, the statistical estimates for the contaminated dataset are not biased.

5 Why Not Use Multi-Core Advantages?

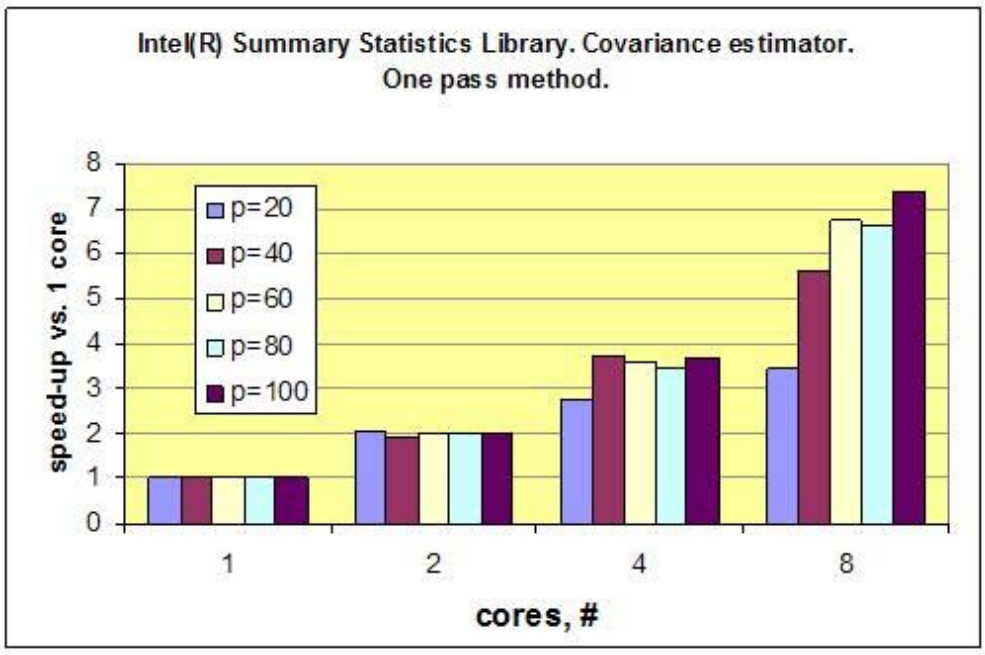
Chapter 2 describes some features and the usage model of the Intel® Summary Statistics Library. However, numerous statistical packages provide good similar functionality. Does the Intel® Summary Statistics Library deliver difference, bring something new and specific? The answer is 'Yes'.

The new era raises new problems of big dimensions. For example, human genome has at least 3 billion DNA base pairs, 20,000-25,000 protein coding genes. This is a really huge amount of information. Fortunately, multi-core processors come to help and make processing of such data arrays easier.

One of important estimators in the library is the algorithm for computation of a variance-covariance matrix. How fast is the algorithm in the Intel® Summary Statistics Library versus the same feature available in the other popular libraries? For comparison, we choose a C-written covariance estimation algorithm underlying the R* project, a GNU suite of functions for data processing, version 2.8.0. Performance measurements are done on a two-way quad-core Intel® Xeon® processor E5440 Series running at 2.83 GHz with 8 GB RAM, 2x6MB L2 Cache. The total number of available cores is 8; we use the function *omp_set_num_threads()* to set the maximal number of cores to be exploited in the measurements. Dimension of the task is 100 and the number of observations is 1,000,000. The dataset is generated from a multivariate Gaussian distribution, the one-pass method is used for computation of the variance-covariance matrix. The measurements show that if the number of available threads is 8, the algorithm in the Intel® Summary Statistics Library is ~16.7 x times faster than the algorithm in R*.

Computation of Variance-Covariance Matrix, seconds				
Cores	1	2	4	8
Intel® Summary Statistics Library	20.55	10.41	5.59	2.79
R* Project	46.56	46.56	46.56	46.56
Speed-up vs. R*	2.27	4.47	8.33	16.69

The chart below gives an additional idea of how the covariance estimator in the Intel® Summary Statistics Library is scaled over the number of additional cores/threads. In the performance measurements, the number of observations remains the same that is, 1,000,000 for all task dimensions p=20, 40, 60, 80, and 100. The experiment shows that the more cores are used the faster the results are obtained.

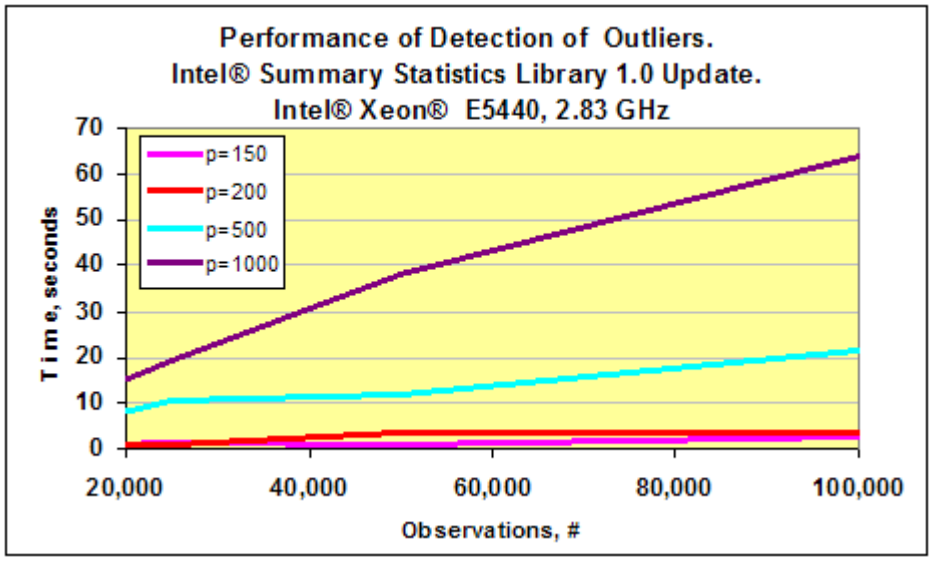


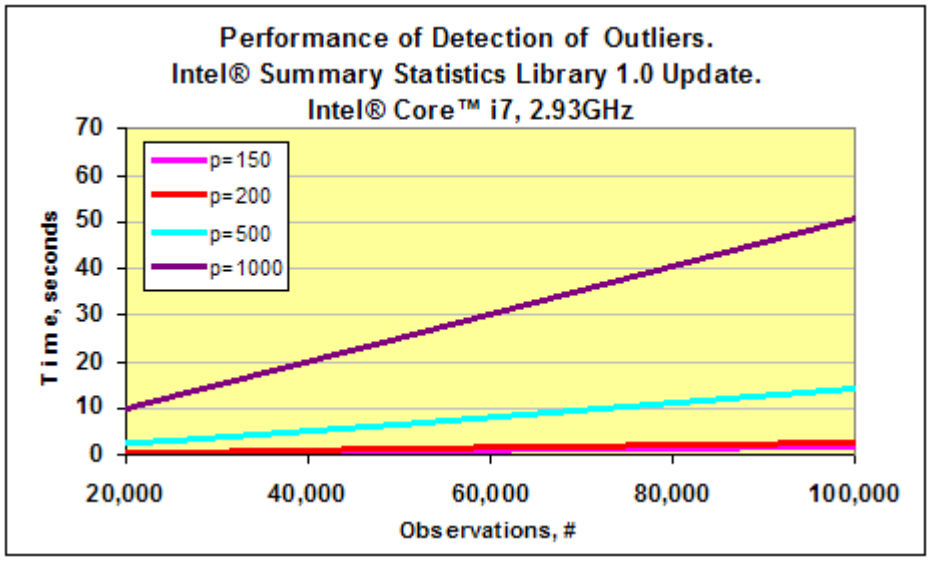
If

6 How Fast is the Algorithm for Detection of Outliers?

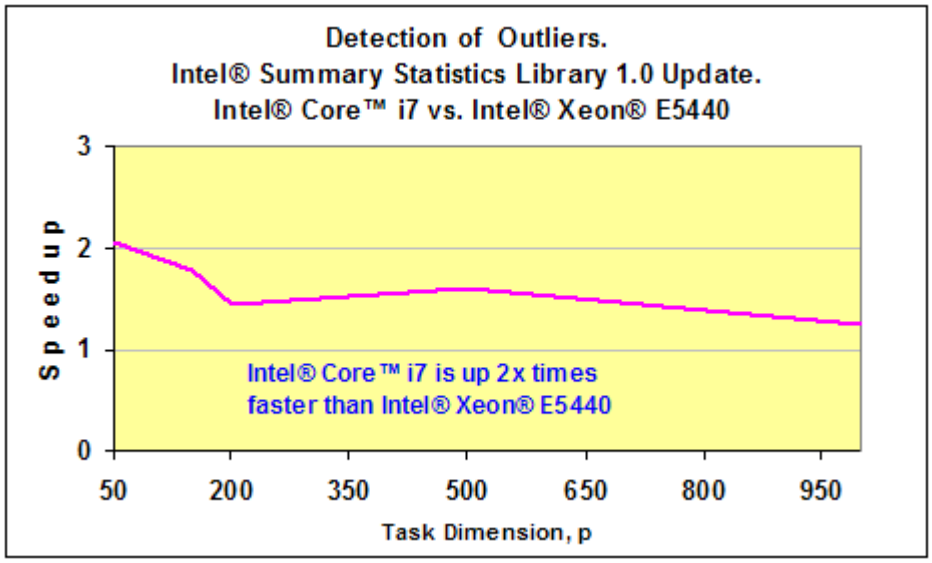
Chapter 4 describes the usage model of the algorithm for detection of outliers available in the Intel® Summary Statistics Library. To have an idea about the speed of the algorithm, we measure its performance on Intel® Xeon® E5440, 2.83 GHz and Intel® Core™ i7, 2.93GHz based machines. To do this experiment, we generate a dataset from a multivariate Gaussian distribution. Dimension of the Gaussian vector *p* is varied from 50 to 1,000, and the number of observations *n* – from 20,000 to 100,000. Generation of outliers is similar to that described in Chapter 4. Two graphs below demonstrate performance of the outlier detection scheme in the Intel® Summary Statistics Library. For *p*=50, performance of the algorithm is less than 0.5 second and is not shown in the graphs.

If dimension of the task *p* is equal to 1,000 and the number of observations is 100,000, the whole procedure takes less then one minute on Intel® Core™ i7 CPU-based machine and a little bit longer – on Intel® Xeon® E5440 processor-based machine.





In other words, Intel® Core™ i7 CPU is up to 2x times faster than Intel® Xeon® E5440 processor in this specific application. The graph below, which compares the two platforms, specifies the CPU speed. As Intel® Core™ i7 CPU has a higher frequency the speed-up of the algorithm for detection of outliers on this platform is even higher.



7 How to Use Robust Methods?

The Intel® Summary Statistics Library provides several opportunities for processing datasets "contaminated" with outliers. Chapter 4 specifies different aspects of the algorithm for detection of "suspicious" observations in the dataset. Chapter 6 provides some ideas about its performance. Another approach to treatment of outliers is to use robust methods available in the library. How to use the algorithms for robust estimation of mean and variance-covariance matrix as they deserve special attention?

Robust methods in the Intel® Summary Statistics Library form a solution that is presented with two algorithms, Maronna, [2] and TBS, [3]. The first algorithm is used to compute start "point" (covariance and mean) for the second one. The TBS algorithm allows iterating until necessary accuracy is achieved or the maximal number of iterations is completed. In addition to these parameters, you can specify and pass the maximal breakdown point (the number of outliers the algorithm can hold) and an asymptotic rejection probability (ARP) [2] in the library. To avoid iterations of the TBS algorithm and compute robust estimate of mean and covariance using only the Maronna algorithm, just set the number of iterations to zero. Additional details about the robust methods are available in Section 6.5 of the Intel® Summary Statistics Library User Manual [1].

To use the Intel® Summary Statistics Library, we pass the same four stages:

1. creation of the task
2. edition of task parameters
3. computation of stat estimates
4. de-allocation of the resources.

All these steps are described in Chapter 2. Let us describe how to use the editor for the robust methods and *Compute* routine. Parameters of the algorithms, breakdown point, ARP, accuracy and the maximal number of TBS iterations are passed as an array:

```
breakdown_point = 0.2;
arp             = 0.001;
method_accuracy = 0.001;
iter_num        = 5;

robust_method_params[0] = breakdown_point;
robust_method_params[1] = arp;
robust_method_params[2] = method_accuracy;
robust_method_params[3] = iter_num;
```

We also need memory *t_est* and *cov_est*, where the robust estimates will be stored. In the example below, the covariance matrix is stored in the full format specified in variable *robust_cov_storage*.

```
errcode =  vsldSSEditRobustCovariance( task, &robust_cov_storage,
&robust_params_n, robust_method_params, t_est, cov_est );
```

Computation of the estimates is done by means of the *Compute* routine:

```
errcode=vsldSSCompute( task, VSL_SS_ROBUST_COVARIANCE,
                       VSL_SS_ROBUST_TBS_METHOD );
```

To see how The Intel® Summary Statistics Library deals with outliers, we create a task with dimension p = 10 and the number of observations n = 10,000. The dataset is generated from a multivariate Gaussian distribution with zero mean and a covariance matrix that holds 1 on the main diagonal and 0.05 in the other entries of the matrix. We then contaminate the dataset with shift outliers that have a multivariate Gaussian distribution with the same covariance matrix and a vector of means with all entries equal to 5.

Use of the non-robust algorithm for covariance and mean estimation for this dataset results in biased estimates. Expectedly, we get zero p-values for these estimates.

```
Means:
0.2566,0.2583,0.2576,0.2633,0.2439,0.2556,0.2530,0.2716,0.2535,0.2519

Covariance:
2.2540
1.2715 2.1819
1.2852 1.2462 2.2046
1.2885 1.2684 1.2553 2.2310
1.2850 1.2581 1.2571 1.2526 2.2112
1.2650 1.2284 1.2419 1.2820 1.2430 2.1929
1.2789 1.2435 1.2550 1.2555 1.2574 1.2478 2.2113
1.2773 1.2692 1.2676 1.2751 1.2725 1.2733 1.2739 2.2448
1.2813 1.2579 1.2688 1.2723 1.2670 1.2713 1.2839 1.3061 2.2246
1.2696 1.2631 1.2515 1.2701 1.2597 1.2686 1.2554 1.2638 1.2780 2.1893
```

Use of the Maronna algorithm (that is, *iter_num* = 0) results in the following estimates:

```
Means:
-0.0022,0.0081,-0.0075,0.0049,-0.0054,0.0012,-0.0087,0.0194,-0.0073,0.0022

p-values for means:
0.1792 0.6077 0.5640 0.3869 0.4281 0.1014 0.6375 0.9570 0.5602 0.1846
```

```
Covariance:
0.9164
0.0605 0.8945
0.0617 0.0374 0.9269
0.0602 0.0570 0.0472 0.9294
0.0584 0.0469 0.0599 0.0443 0.9183
0.0552 0.0394 0.0395 0.0655 0.0484 0.9049
0.0487 0.0449 0.0471 0.0451 0.0564 0.0461 0.9186
0.0293 0.0555 0.0539 0.0456 0.0450 0.0574 0.0501 0.9149
0.0507 0.0339 0.0433 0.0504 0.0429 0.0603 0.0597 0.0696 0.8962
0.0375 0.0573 0.0470 0.0472 0.0502 0.0607 0.0420 0.0381 0.0484 0.8848

p-values for covariance:
0.0000
0.2989 0.0000
0.2966 0.5842 0.0000
0.3471 0.4395 0.9592 0.0000
0.3994 0.9148 0.3590 0.8993 0.0000
0.5128 0.7023 0.6708 0.1869 0.8510 0.0000
0.8508 0.9752 0.9515 0.9411 0.4812 0.9714 0.0000
0.2669 0.4841 0.6001 0.9729 0.9530 0.4207 0.7751 0.0000
0.7151 0.4529 0.8765 0.7468 0.8689 0.2968 0.3317 0.0984 0.0000
0.6082 0.3734 0.9088 0.8997 0.7250 0.2720 0.8321 0.6358 0.7895 0.0000
```

These estimates are much better; however the main diagonal of the matrix still results in the zero p-value. To improve the estimate, we do five iterations of the TBS algorithm. Quick experiments show that it does not make sense to iterate longer as the estimates do not change significantly after five iterations:

```
Means:
-0.0018,0.0034,0.0026,0.0067,-0.0108,0.0012,-0.0024,0.0122,-0.0057,-0.0044

p-values for means:
0.1412 0.2612 0.2025 0.4860 0.7098 0.0943 0.1882 0.7693 0.4263 0.3381

Covariance:
1.0524
0.0583 1.0172
0.0757 0.0426 1.0403
0.0653 0.0630 0.0490 1.0538
0.0672 0.0604 0.0559 0.0462 1.0367
0.0493 0.0295 0.0434 0.0784 0.0442 1.0261
```

```
0.0620 0.0429 0.0509 0.0453 0.0491 0.0488 1.0397
0.0410 0.0503 0.0476 0.0507 0.0497 0.0514 0.0497 1.0367
0.0450 0.0370 0.0486 0.0464 0.0430 0.0526 0.0622 0.0719 1.0179
0.0477 0.0587 0.0461 0.0562 0.0514 0.0645 0.0443 0.0346 0.0485 1.0070

p-values for covariance:
0.0002
0.6951 0.2249
0.1676 0.5972 0.0044
0.4613 0.5057 0.8450 0.0001
0.3761 0.5862 0.8152 0.7231 0.0095
0.8726 0.1942 0.6233 0.1170 0.6604 0.0646
0.5690 0.6118 0.9464 0.6795 0.8671 0.8653 0.0050
0.5092 0.9507 0.7992 0.9266 0.9002 0.9932 0.8944 0.0094
0.6867 0.4013 0.8656 0.7504 0.6147 0.9305 0.5185 0.2177 0.2065
0.8205 0.6243 0.7594 0.7800 0.9869 0.4071 0.6776 0.3207 0.8961 0.6185
```

8 How to Deal with Missing Observations?

Real life datasets can have missing values. Sociological surveys and measurement of complex biological systems are two examples where missing observations cannot be further ignored. Outliers in datasets can also be treated as lost samples. Intel® Summary Statistics Library contains functionality to detect outliers or get robust estimates in presence of “suspicious” observations. Chapter 4 and Chapter 7 show how to use these algorithms. Let us consider another the Intel® Summary Statistics Library algorithm that deals with missing values.

The method for accurate processing of datasets with missing points is based on the approach described in [4]. The Expectation-Maximization (EM) and Data Augmentation (DA) algorithms are integral components of this solution (let us call it EMDA solution). Output of the algorithm is *m* sets of simulated missing points that can be imputed into the dataset, thus, producing *m* complete data copies. For each dataset, you can compute a specific stat estimate; the final estimate is a combination of such *m* estimates.

The usage model for the EMDA method is similar to that for other algorithms of the library (see Chapter 2): creation of the task, editing its parameters, computing, and de-allocation of the resources. Let us show how to pass parameters of the EMDA method into the library and to call the *Compute* routine.

The EM algorithm does *em_iter_num* iterations to compute an initial estimate for mean and covariance, which are then used to start the DA algorithm. The EM algorithm can terminate earlier if the given accuracy *em_accuracy* is achieved. In its turn, the DA algorithm iterates *da_iter_num* times. It also uses Gaussian random numbers underneath. For this reason, the Intel® Summary Statistics Library uses the Vector Statistical Library (VSL) component of the Intel® Math Kernel Library (Intel® MKL). *BRNG*, *SEED*, and *METHOD* are the parameters used to initialize VSL generators. Values of the parameters should follow VSL requirements. The EMDA algorithm also requires a number of missing values *missing_value_num*. Before calling the *Compute* routine, pre-process the dataset and mark all missing values using the *VSL_SS_DNAN* macro defined in the library. For a single precision dataset, the *VSL_SS_SNAN* macro is appropriate. Parameters of the algorithm are passed into the library as the array:

```
em_iter_num     = 10;
da_iter_num     = 5;
em_accuracy     = 0.001;
copy_num        = m;
miss_value_num  = miss_num;
brng            = BRNG;
seed            = SEED;
```

```
method          = METHOD;

mi_method_params[0] = em_iter_num;
mi_method_params[1] = da_iter_num;
mi_method_params[2] = em_accuracy;
mi_method_params[3] = copy_num;
mi_method_params[4] = missing_value_num;
mi_method_params[5] = brng;
mi_method_params[6] = seed;
mi_method_params[7] = method;
```

The editor for the EMDA algorithm accepts a set of parameters:

```
errcode = vsldSSEditMissingValues(task, &mi_method_params_n, mi_method_params,
&initial_estimates_n, initial_estimates, &prior_n, prior,
&simulated_missing_values_n, simulated_missing_values, &estimates_n, estimates);
```

The array *mi_method_params* is described above. The array of initial estimates *initial_estimates* is used to start the EM algorithm; its first *p* positions are occupied with the vector of means and the rest *p*(p+1)/2* entries hold the upper-triangular part of the variance-covariance matrix (*p* is dimension of the task). The array *prior* is intended to hold prior parameters for the EMDA algorithm; the current version of the library does not support user-defined priors, and default values are used.

Sets of simulated missing points are returned in the array *simulated_missing_values*, in total *m* x *missing_value_num* values. Missing values are packed one by one, first all missing points for the 1st variable of the random vector, then missing values for the 2nd variable, and so forth. To estimate convergence of the DA algorithm, pass the array *estimates* that will hold mean/covariance for all iterations and all sets of simulated missing points, in total *m* x *da_iter_num* x $(p + 0.5 \times (p^2 + p))$. Each set of estimates is packed as above: first *p* positions are intended for mean, and the rest $0.5 \times (p^2 + p)$ entries hold upper-triangular part of the covariance matrix.

Finally, the EMDA algorithm is started using the *Compute* routine of the library:

```
errcode = vsldSSCompute( task, VSL_SS_MISSING_VALUES,
                         SL_SS_MULTIPLE_IMPUTATION );
```

In our experiment, the task was created with dimension *p = 10* and the number of observations *n = 10,000*. The dataset is generated from a multivariate Gaussian distribution with zero mean and covariance matrix that holds 1 on the main diagonal and 0.05 in the rest

entries of the matrix. Ratio of missing values in the dataset is 10%; each observation may have one missing point in any position. Our goal is to generate *m=100* sets of the lost observations. Start “point” for the EM algorithm is the vector of zero means and the identity covariance matrix; the pointer to the array *prior* is set to 0 (size of this array prior_n is also 0).

After a trial run of the algorithm with *da_iter_num* = 10 we analyze estimates in the array *estimates*. It turns out that five iterations are sufficient for the DA algorithm. We then simulate 100 sets of missing values, impute them into the dataset, and thus get 100 complete data arrays. For each complete dataset, we compute means and variance using the algorithms of the Intel® Summary Statistics Library, in total 100 sets of mean and covariance estimates:

```
Set:           Mean:
    1        0.013687  0.005529  0.004011 ...  0.008066
    2        0.012054  0.003741  0.006907 ...  0.003721
    3        0.013236  0.008314  0.008033 ...  0.011987

...

   99        0.013350  0.012816  0.012942 ...  0.004076
  100        0.014677  0.011909  0.005399 ...  0.006457
_____________________________________________________

Average      0.012353  0.005676  0.007586 ...  0.006004

Set:        Variance:
    1        0.989609  0.993073  1.007031 ... 1.000655
    2        0.994033  0.986132  0.997705 ... 1.003134
    3        1.003835  0.991947  0.997933 ... 0.997069

...

    99       0.991922  0.988661  1.012045 ... 1.005406
   100       0.987327  0.989517  1.009951 ... 0.998941
_____________________________________________________

Average      0.99241   0.992136  1.007225 ... 1.000804

Between-imputation variance:
0.000007 0.000008 0.000008 ... 0.000007

Within-imputation variance:
0.000099 0.000099 0.000101 ... 0.000100

Total variance:
```

```
0.000106 0.000107 0.000108 ... 0.000108
```

We also compute 95% confidence intervals for the vector of means:

```
95% confidence interval:
Left boundary of interval:  -0.008234 -0.015020 -0.013233 ...  -0.014736
Right boundary of interval: +0.032939 +0.026372 +0.028406 ... +0.026744
```

To test the output of the algorithm, we repeat the whole experiment 20 times. All twenty 95% confidence intervals contain the true value of mean.

See more details on support of missing values in Section 6.7 of the User Manual [1].

9 How to Compute Quantiles for Streaming Data?

Algorithms of the Intel® Summary Statistics Library provide support for huge datasets that cannot fit into memory of a computer. Chapter 3 describes analysis of the datasets available in blocks. The Intel® Summary Statistics Library contains another algorithm that supports out-of-memory data, a scheme for computations of quantiles. Let us take a closer look at the opportunities provided by this method.

The theory and properties of this algorithm are described in [5]. In a nutshell, this method allows getting an ε-approximate quantile (that is, the element in the dataset whose rank is within $[r-\varepsilon n, r+\varepsilon n]$ interval for a user-provided error ε, size of dataset n and any rank $r=1,\ldots,n$) for the price of one pass over the dataset. Another important aspect of the algorithm is that you don't need to know the total size of the dataset in advance.

Let us consider a simple application that uses this algorithm. As usual, it passes through four basic important stages: creation of the task, edition of necessary task parameters, getting stat estimates, and de-allocation of the resources.

We set dimension of the task p to 1 and choose the total number of observations n=10,000,000. The dataset "arrives" in blocks of 10,000 elements each. Our goal is to compute deciles with a pre-defined error ε = 0.00001, that is, array elements whose ranks deviate from rank of the "accurate" deciles by no more than 100 positions.

The library contains a special editor for this algorithm:

```
status =  vsldSSEditStreamQuantiles ( task, &quant_order_n, quant_order,
                                      quantiles, &params_n, &params );
```

The array *quant_order* is initialized with quantile orders: 0.1, 0.2, …, 0.9; the total number of quantiles is *quant_order_n*=9. Results of the computations will be placed into the array *quantiles*. Finally, the algorithm for computation of streaming quantiles accepts an array of parameters. It contains one element, the user-defined error *eps* which is `0.00001` in this example. To initialize size of an array that contains parameters of the algorithm, we can use the macro defined in the library: *params_n = VSL_SS_STREAM_QUANTILES_ZW_PARAMS_N*.

The loop for computation of the deciles is as follows:

```
for ( block_index = 0; block_index < max_block_num; block_index++ )
{
// Get the next data block of size block_n
…
status = vsldSSCompute( task, VSL_SS_STREAM_QUANTILES,
VSL_SS_STREAM_QUANTILES_ZW_METHOD );
// Process computation results
…
}
```

We obtain intermediate estimates of deciles immediately after the processing of the next block. In this case (the dataset contains Gaussian numbers with mean equal to 0 and variance 1), the sequence of the estimates is as follows:

```
Block   Streaming deciles:
index        D1      D2      D3      D4      D5     D6      D7     D8    D9
1         -1.2671 -0.8442 -0.5257 -0.2667 -0.0115 0.2524 0.5391 0.8496 1.2695
2         -1.2880 -0.8478 -0.5374 -0.2766 -0.0192 0.2400 0.5131 0.8327 1.2690
3         -1.2848 -0.8386 -0.5261 -0.2656 -0.0110 0.2428 0.5163 0.8366 1.2704
…
…
998       -1.2815 -0.8414 -0.5241 -0.2531  0.0009 0.2536 0.5248 0.8412 1.2814
999       -1.2815 -0.8414 -0.5241 -0.2531  0.0009 0.2536 0.5248 0.8413 1.2814
1000      -1.2815 -0.8414 -0.5241 -0.2531  0.0008 0.2536 0.5248 0.8412 1.2813
```

If we do not need to know an intermediate quantile estimate and our task is computation of the estimate for the whole dataset, the library provides opportunity to use the so-called “Fast Mode” that allows you just to update the internal data structure in the library without actual computation of the intermediate estimates.

```
for ( block_index = 0; block_index < max_block_num; block_index++ )
{
// Get the next data block of size block_n
…
status = vsldSSCompute( task, VSL_SS_STREAM_QUANTILES_FMODE,
VSL_SS_STREAM_QUANTILES_ZW_METHOD );
}
```

To obtain the estimate, we set the *block_n* variable to zero and call the *Compute* function. Prior to the call, make sure the variable is registered in the library:

```
block_n = 0;
status = vsldSSCompute( task, VSL_SS_STREAM_QUANTILES,
VSL_SS_STREAM_QUANTILES_ZW_METHOD );
```

We note that the output of this application is identical to the last line of the previous table:

```
Streaming deciles:
   D1        D2        D3        D4       D5       D6       D7       D8      D9
-1.28154 -0.84141 -0.52418 -0.25312 0.00088 0.25367 0.52483 0.84129 1.28139
```

To check the estimates, we calculate “accurate” deciles for the same dataset using in-memory versions of the quantile algorithm available in the library. Its output is as follows:

```
“Accurate” deciles:
   D1        D2        D3        D4       D5       D6       D7       D8      D9
 -1.28155 -0.84142 -0.52417 -0.25311 0.00089 0.25368 0.52484 0.84130 1.28140
```

The maximal difference between ranks of “in-memory” and “out-of-memory” deciles does not exceed 100, which is in line with the theory:

```
Rank difference
   D1        D2        D3        D4       D5       D6       D7       D8      D9
   4         5         3         1        3        7        8        7       4
```

Section 6.2 of the User Manual [1] contains additional details about the algorithm for computations of quantiles for streaming data.

10 References

1. Intel® Summary Statistics Library User Manual.
2. R.A. Maronna and R.H. Zamar, Robust Multivariate Estimates for High-Dimensional Datasets. Technometrics, 44, 307–317, 2002.
3. David M. Rocke. Robustness properties of S-estimators of multivariate location and shape in high dimension. The Annals of Statistics, 24(3), 1327-1345, 1996.
4. J.L. Schafer. Analysis of Incomplete Multivariate Data, Chapman & Hall/CRC, 1997.
5. Zhang and W. Wang. A Fast Algorithm for Approximate Quantiles in High Speed Data Streams. SSDBM, p.29, 19th International Conference on Scientific and Statistical Database Management (SSDBM 2007), 2007.