Using Massively Parallel Processing in the Testing of the Robustness of Statistical Tests with Monte Carlo Simulation

Tamás Ferenci[1] – Balázs Kotosz[2]

In this paper, we will examine the application of the Monte Carlo method in the testing of the robustness of statistical tests. The very computation-intensive Monte Carlo testing was implemented using the so-called GP-GPU method, which utilises the video cards' GPU (Graphical Processing Unit) to perform the necessary calculations.

The robustness of a statistical test is defined as its property to remain valid even if its assumptions are not met. (We call a test valid if its significance level is equal to its Type I Error Rate.) One way to investigate the robustness of a statistical test (especially useful if the test's complicated structure makes analytic handling infeasible or impossible) is to employ the so-called Monte Carlo method. Here, we generate many random samples (meeting or violating the assumptions, which we can arbitrarily set), perform the statistical test many times on them, and then check whether the empirically found Type I Error Rate converges to the specified significance level or not. This way, we give up exactness for the complete insensitivity to the complexity of the examined statistical test.

This method (by requiring enormous amount of random number generations and statistical testings for reliable results) is very computation-intensive; traditionally only supercomputers could be used effectively, limiting the availability of this method. However a new approach, called GP-GPU, makes it possible to harness the extremely high computing performance of – even ordinary, widely available – video cards found in every modern PC.

We implemented a framework that performs the abovementioned MC-testing of the robustness of statistical tests. We call our program "framework", because it can be easily expanded with virtually any statistical test (to which no GP-GPU knowledge is needed), and be tested with very high performance – even with widely available tools.

As an example, we performed the analysis of the well known Student's t-test; and – using this as a starting point – we demonstrated the main advantages of our framework.

Keywords: robustness of statistical tests, Monte Carlo method, GP-GPU, simulation

[1] Tamas Ferenci, MSc, Corvinus University of Budapest, Faculty of Economics, Department of Statistics (Budapest)
[2] Balazs Kotosz, PhD, assistant professor, Corvinus University of Budapest, Faculty of Economics (Budapest)

1. On the robustness of statistical tests and its empirical investigation with Monte Carlo simulation

1.1. Introduction

Statistical hypothesis testing, a method of inductive statistics, is of central importance in both theoretical and applied statistics (Hunyadi–Vita 2006). Since its introduction by Ronald Fisher, Jerzy Neyman and the two Pearsons in the early 20th century (Hald 1998) it became a cornerstone of modern statistics, widely used in nearly every area of nature-, life- and social sciences and technological practice. (Despite receiving challengers, mainly the Bayesian approach – for an introduction see (Lee 2009) – in the late 20th century.)

Statistical hypothesis testing is performed using so-called *statistical tests*, of which hundreds (Hunyadi 2001) have been developed throughout the decades for a variety of purposes. These tests are usually based on *assumptions*, typically regarding the populations from which the samples are coming on which the test operates. The test behaves the way it was specified only if these assumptions are met.

To illustrate the abovementioned, Table 1 shows some statistical tests for testing the equality of the means (expected values) of two populations, from which independent samples are available. Table 1 also gives the assumptions of these statistical tests.

Table 1. Selected statistical tests for testing the equality of population means (expected values)

Name of the test	Assumptions
Student's *t*-test	Populations normally distributed, having equal variances
Welch-test	Populations normally distributed
Mann-Whitney *U*-test	Populations arbitrarily distributed, but PDFs having the same shape
Brunner-Munzel test	None

Source: own creation

As it can be well seen, tests being lower in Table 1 have fewer assumptions. This is not a universal advantage however: these tests also have less statistical power. (So, when selecting a statistical test for a purpose[3], it has to be carefully considered, which are the assumptions that can be either a priori accepted or tested (preferably on an independent sample), and select a test that is based *only* on those – no less (loss of power), no more (wrong behaviour).)

[3] See (Ferenci 2009b) for a brief discussion of these questions in connection with a (biostatistical) application.

1.2. Validity and robustness of statistical tests

In the previous subsection, we said "The test behaves the way it was specified only if these assumptions are met". This can be made more precise as follows.

A test is guaranteed to be *valid* only when its assumptions are met. The validity of a statistical test is defined as its property of having Type I Error Rate equal to its significance level.

When a statistical test's assumptions are not met, it is no longer mathematically guaranteed that the test is valid. However, statistical tests can show quite different behaviours when being operated on samples violating the assumptions: some tests still remain valid to some extent, even if their assumptions are not met. This property of a statistical test is called *robustness*; we call a statistical test *robust*, if it has this property.

Validity can be obviously checked not only when the assumptions are met, but also when they are violated (logically it needs to be – necessarily – confirmed only in the former case). Hence the checking of robustness can be viewed as an extension of the checking of validity: we can conclude on robustness by performing many checkings of validity (with assumptions violated to different extent).

1.3. Using Monte Carlo (MC) method to examine the robustness of a statistical test

One way to check the robustness of a statistical test is to analyse its mathematical structure: suppose some assumption-violating property of the samples (which can be formalised mathematically), expose the test to such samples, and – still purely mathematically – derive how the test would operate on those samples. (E.g. what Type I Error Rate would it produce with a given significance level.)

The advantage of this method is that it results in *exact analysis*. However, there is one serious limitation: it becomes increasingly *infeasible* (if not impossible) when used on tests that have more and more complex algebraic structure – typical for today's statistical tests. This might even make exact, mathematical analysis impossible.

An alternative approach is to use the so-called *Monte Carlo* (MC) method for checking robustness: it gives up the exactness, but in exchange it provides a testing method which is completely insensitive to the complexity of the examined statistical test (Rubinstein–Krose 2008).

MC method (which is a general approach used in many other areas of scientific research) is based on the idea of *empirically exploring* a system: instead of describing the system with rigorous mathematical formalism, it is investigated stochastically, i.e. the system is faced with many randomly selected inputs, and its "operation" is reconstructed by the outputs it generates as an answer to the random inputs. Consequently, this is a stochastical approach, and it can be effective if a very high number of random inputs are given, so that the system can be mapped appropriately.

The MC method is employed in applications where deterministic examination is infeasible or impossible due to the complexity of the problem. (A classical example for the application of MC is high dimensional numerical integration: while it is almost impossible to perform a numerical integration in 100 dimensions with classical mathematical tools (10^{100} points would be needed just to have 10 integration point on each axis!), it is well possible with MC, in which points are randomly selected in the 100 dimensional space and then checked whether they are "above" or "below" of the function – this avoids the exponential increase in the number of required points.)

In our context (checking the validity/robustness of statistical tests), this means the following. We generate a random sample and then apply the statistical test. Then, we record the result (accept or reject) and repeat this procedure ("test-cycle") many times. Thus, we obtain a Type I Error Rate, which we can compare to the significance level: as stated, the test is valid if these two are equal. An important side note is however, that "equality" has to be understood in its statistical sense: as Type I Error Rate was determined empirically (i.e. by testing samples), this rate will be exposed to statistical fluctuation. (Caused by the fact that the rate depends on which samples we actually test.) When comparing this with the fixed significance level, we can never say "surely" that they are equal; rather we can only draw a statistical conclusion. (Using another statistical test, as a matter of fact…) However, we can very well aim to draw a conclusion that they are "extremely highly likely" to be equal. This can be achieved by using very high number of random samples. Thus, we actually examine whether the empirically found Type I Error Rate *converges to* the significance level as the number of test-cycles increases.

This operation makes it understandable why it is also called MC-simulation.

A critical aspect of this approach is that we have full control on the properties of the samples. (As we generate them, we can arbitrarily parameterise the random number generator algorithm.) Hence for any statistical test, we can generate samples that meet the test's assumptions, and samples that do not. From this point, the completion of the MC method for checking validity/robustness is quite straightforward: if we generate samples that *meet* the investigated test's assumptions, we can test validity, if we generate samples that *do not meet* these assumptions, we can test robustness.

Moreover, if the extent, to which the assumptions are not met, can be quantified, we can also *iterate through* different levels of departure from the assumptions, hence "mapping" the robustness of the test. (A prime example for this is the violation of normality, which can be well quantified with skewness and kurtosis. We will return to this question later in this paper.)

1.4. Our demonstrational example: checking the robustness of the t-test for non-normality

We chose Student's two independent samples *t*-test as a *demonstrational example* for MC-simulational testing. It is solely a demonstration: the results obtained will not be of scientific value, as this test has been completely analysed long ago – the method, the MC-simulational testing itself is interesting now. And for demonstration, it seemed logical to choose a simple, well-known statistical test (Hunyadi–Vita 2006). This is in fact the most well-known statistical test, and perhaps also the most widely used ever developed. It can be used to check the equality of the means (expected values) of two populations, from which independently drawn samples are available.

The test assumes (among others) that the populations are *normally distributed.* (See also Table 1.) Our aim will be to check robustness in respect to this assumption. (I.e. we will let every other assumption be met, but violate this one.) Furthermore, we will quantify the violation of this normality assumption (by the skewness and kurtosis of the population's distributions), so we will also be able to "map" the robustness: to check how validity alters when the test's assumptions are more and more violated. Thus we will be able to answer the question: how valid is the *t*-test when the populations are non-normal, i.e. how robust it is to the violation of population normality?

1.5. Problem statement

As already stated, we will use MC-simulation to answer this question. In this context, it means that we will randomly generate many samples coming from distributions having specified skewness and kurtosis, and test them. By recording empirical Type I Error Rate, we can judge the validity of the test. And by iterating through different levels of skewness and kurtosis, we can test robustness.

The problem is that this is very *time-consuming*, even on modern personal computers. This can be traced back to the fact that even modern PCs have inadequate computing performance when using very high number of test cycles. (Which is needed however to achieve low statistical fluctuation in empirical Type I Error Rate, a question which we already discussed.) If multiple parameters govern the level of violation of the assumptions (two in our demonstrational example: skewness and kurtosis), and we want to check *every possible combination* of them, it will also exponentially raise the number of needed tests.

In this paper we use a novel approach to handle this situation. Traditionally, the only resolution would have been to employ a supercomputer, grid computing etc., available only to a few researchers. Now, we will show a solution which is *widely available* (both technically and financially), but still produces *impressive performance* (sometimes magnitudes higher computing performance than the CPU!), and can be used to MC-simulational testing of the robustness of statistical tests. We will discuss this in Section 2.

2. On GP-GPU computing and its usage in the MC-simulational examination of statistical tests

2.1. Introduction

Since the introduction of the category in the early 1980's, *video cards* (Cserny 1997) constitute a regular part of Personal Computers (PCs). These devices (typically built as a standalone expansion card, communicating with the host computer through some standardised bus) are responsible for generating visual output from the computer (typically displayed on monitors).

Since the mid-1990's, an increasing demand appeared for video cards that can perform image-generation related calculations themselves, instead of relying on the computer's central processing unit (CPU), thus relieving that. This was especially true when 3D image generation became a commonplace (due to computer games, above all). These required tremendous amount of calculations, but only from a very few types, so it seemed logical to design a hardware *specifically* for that purpose (instead of loading the general purpose CPU with this task), which is usually named *hardware acceleration*. Hence hardware for this purpose, so-called Graphical Processing Units (GPUs) began to widely spread – typically VLSI (Very Large Scale Integration) chips, designed specifically to accelerate image generation-, especially 3D image generation-related calculations.

The early-2000s saw an enormous development in the field of GPUs. They became more and more complex (as an illustration: the number of transistors (Hagerdoorn 2009) grew from 1 million (Voodoo, 1996) to 1.5 billion (NVidia GT200, 2009)!), they were able to perform more and more sophisticated tasks in the field of hardware acceleration. Because these were highly specialized designs (as we already mentioned, only a few types of calculations have to be done by a GPU, compared to the general purpose CPU), their *processing power* grew in a much faster rate than that of CPUs. (As only a few types of calculations have to be done… but these have to be done very fast.)

This, along with the fact, that the GPU's more and more complex structure also widened the range of possible calculations, rose up the idea of using the GPU for applications that are *non-graphical*, but require *high computing capacity*. This led to a technique called General Purpose GPU, or GP-GPU (Owens et al. 2007). Note however that the term "general" is a bit misleading: it is not "general" in the sense we named CPU so. While theoretically GP-GPU in fact aims to run any program on GPU, practically this is only useful when the program fits to the structure of the GPU.

The key point in that structure is the *very highly parallel* nature: a GPU consists of processing units which are very simple themselves, but a very high number of them are used. Thus logically those programs can be effectively recoded to run on GPU (with the GP-GPU approach) which can be well parallelized… just like Monte Carlo simulation! (In addition to that, a range of other tasks might be solved effectively with GP-GPU, from signal processing to cryptography.) For these tasks, GPUs can be used essentially as "supercomputers".

This approach can be related to the Massively Parallel Processing (MPP) concept in the theory of computer architectures.

2.2. *Usage of GP-GPU in the MC-simulational examination of statistical tests*

It has to be strongly emphasized, that even cheap, widely available video cards (more precisely: their GPUs) can provide *extremely high* computing performance for applications that can utilise it. As we just pointed it out, Monte Carlo simulation is a prime example for this, so we decided to undertake this (i.e. GP-GPU) approach to resolve the problem of obtaining high computing performance (which is needed in MC-simulation), but still creating a solution available to virtually any interested user.

The video card we selected was based on NVidia's GeForce 9600 GT GPU (from the GeForce 9 series), based on the G94a/b GPU core (NVidia 2009a). This is a low-middle class video card, sold at around 20 000 HUF (available at any online store at around 100 €). It

- has 505 million transistors,
- a core clock rate of 650 MHz,
- 64 stream processors, each running at 1 625 MHz,
- memory bandwidth of 57,6 GB/sec, and
- a theoretical *processing rate* of 312 GFlops!

The last feature is perhaps the most important for us at first glance: it tells that the GPU is (theoretically) able to perform more than 300 billion floating point operations per second!

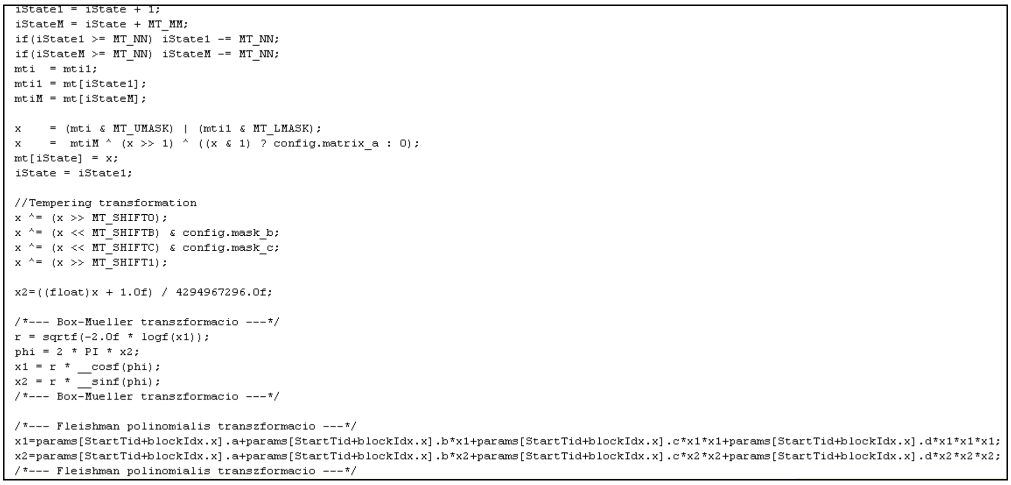
2.3. *The usage of CUDA in developing GP-GPU programs*

Even if the video card and the concept of the program to be run on the GPU are provided, it is still a question how we can in fact code a program to run on GPU.

In the first era of GP-GPU, programs had to be written to "mimic" the graphical operations, making development very complicated. (Programs have to be written "as if" they were performing graphical operations, as GPUs only understood such (i.e. graphics-related) commands.) However, GPU manufacturers soon realised the power of GP-GPU and began developing architectures which more naturally supported the development of non-graphical programs for GPUs. (This tendency became so pronounced, that lately even such "video" cards have been released that were specifically designed for GP-GPU, lacking even video output.)

NVidia's such architecture is called *Compute Unified Device Architecture* (NVidia 2009b), or CUDA for short. This is not only a programming language, but a complete environment for GP-GPU. Naturally it also includes a programming language, called "C for CUDA" which can be used to develop programs for GPU. This is the approach we employed for coding our MC-simulational program; an excerpt from the code is shown on Figure 1.

Figure 1. Excerpt from the CUDA program performing MC-simulation



```
iState1 = iState + 1;
iStateM = iState + MT_MM;
if(iState1 >= MT_NN) iState1 -= MT_NN;
if(iStateM >= MT_NN) iStateM -= MT_NN;
mti  = mti1;
mti1 = mt[iState1];
mtiM = mt[iStateM];

x    = (mti & MT_UMASK) | (mti1 & MT_LMASK);
x    =  mtiM ^ (x >> 1) ^ ((x & 1) ? config.matrix_a : 0);
mt[iState] = x;
iState = iState1;

//Tempering transformation
x ^= (x >> MT_SHIFT0);
x ^= (x << MT_SHIFTB) & config.mask_b;
x ^= (x << MT_SHIFTC) & config.mask_c;
x ^= (x >> MT_SHIFT1);

x2=((float)x + 1.0f) / 4294967296.0f;

/*--- Box-Mueller transzformacio ---*/
r = sqrtf(-2.0f * logf(x1));
phi = 2 * PI * x2;
x1 = r * __cosf(phi);
x2 = r * __sinf(phi);
/*--- Box-Mueller transzformacio ---*/

/*--- Fleishman polinomialis transzformacio ---*/
x1=params[StartTid+blockIdx.x].a+params[StartTid+blockIdx.x].b*x1+params[StartTid+blockIdx.x].c*x1*x1+params[StartTid+blockIdx.x].d*x1*x1*x1;
x2=params[StartTid+blockIdx.x].a+params[StartTid+blockIdx.x].b*x2+params[StartTid+blockIdx.x].c*x2*x2+params[StartTid+blockIdx.x].d*x2*x2*x2;
/*--- Fleishman polinomialis transzformacio ---*/
```

Source: own creation

2.4. *On our testing framework*

From this point on, we will refer to our program as "*framework*". This is to emphasize that the most important part of our work is not the coding of statistical tests, but the coding of the MC-simulation robustness-checking module. (This also foreshadows that we took huge effort to write a *modular program*, i.e. a program in which the MC-simulational part and the description of the statistical test itself is *separated.* We will discuss this question in detail in the next section.)

In its current version, our program includes two statistical tests (Student's two independent sample *t*-test, and the Mann-Whitney *U*-test, as we will later discuss), which may be used as demonstrational subjects to the MC-simulational module. (Results from these investigations will be shown in Section 5.)

Another advantage of the "framework" approach is that new statistical tests might be *added* by a description in standard C language – no CUDA or parallel computing knowledge is needed at all.

3. Technical details on the coding of our testing framework

As we explained in the previous section, our investigations were not purely theoretical: we also developed a "testing framework" as a program which can be used to check the robustness of statistical tests with the described Monte Carlo simulation method. In this section, we will review a few considerations that cropped up during the actual implementation of this program.

3.1. Development environment

We developed the program under Microsoft's Visual Studio 2008 integrated development environment (Microsoft 2009).

As we have already mentioned, NVidia's CUDA architecture uses the so-called "C for CUDA" programming language, which is a minimal extension to the standard C language to support the coding of the GPU (NVidia 2009b). This can be comfortably handled within the MS Visual Studio environment, where coding, debugging and compiling can be performed at the same interface.

3.2. Random number generation

The key idea of Monte Carlo methods is to use random inputs (in our case: random samples) to explore the behaviour of the system (in our case: the statistical test). Thus, it is necessary to generate random numbers (lots of random numbers actually, as this has to be repeated many times) in our program.

Random number generators (RNGs) have a library-wide literature – for a relevant review, see for example (Gentle 2004) – and random number generation in a parallel environment is an even more non-trivial question.

We used the RNG employed in the CUDA's own Software Development Kit (SDK), the so-called Mersenne Twister algorithm (Podlozhnyuk 2007). It is one of the most widely used RNGs nowadays, mathematically guaranteed to be equidistributed up to 623 dimensions (Matsumoto–Nishimura 1998), passing all modern randomness test, including the diehard (Matsumoto–Nishimura 1998) and the even more stringent TestU01 (McCullough 2006). This – together with its speed – makes Mersenne Twister especially fit for Monte Carlo simulations.

However, in itself, it is not suitable for use in parallel environment. This was resolved by the so-called DCMT algorithm, which is also employed (Matsumoto–Nishimura 2000).

The Mersenne Twister with DCMT produces uniformly distributed numbers. As we will see in the next subsection, we need random numbers from standard normal distribution for our purpose, so we applied the well known Box-Mueller transform (Ketskeméty 2005).

3.3. Fleishman's polynomial transformation method

As we have already pointed it out, the departure from the assumptions of the test will be quantified by the level of non-normality in our demonstrational example (logically, as the assumption was normality in that example). To quantify non-normality, we will apply the most widely used indicators: *skewness* and *kurtosis* (i.e. the third and fourth standardized central moments, but see Subsection 3.6 as well).

Note that distributions can be viewed as points on the skewness/kurtosis space (or plane, to be more specific).

If we recall the procedure of MC-testing the robustness (Section 2.3), it is immediately obvious that we will need to generate samples from distributions having arbitrary kurtosis and skewness. (In other words we have to iterate through the points (distributions) of the skewness/kurtosis space, and generate samples at every point.) This is a nontrivial problem, as the widely used distributions either have given kurtosis and/or skewness (like normal or exponential distribution) or only one parameter[4] governs kurtosis and skewness (like gamma or lognormal distribution).

There are many solutions for this problem. We might well go back to Karl Pearson when reviewing the history of this question, as the system of Pearson distributions (developed in the early 20th century) can be viewed as an answer to this problem. Since then, many other solutions have been developed, more or less fit for our purpose. (As an example: the Pearson distribution is largely unfit, as it involves distributions that have diverse algebraic structure, so the generating algorithm would have to change between essentially different distributions when iterating through the skewness/kurtosis space.)

The solution we employ now was published in 1978 by Allen I. Fleishman (Fleishman 1978). As it might be unknown to the reader, we will discuss it in detail.

The key idea behind Fleishman's solution is the following. When generating samples from a standard normal distribution we can control neither the mean, nor standard deviation, nor the skewness, nor the kurtosis of the distribution. However, if we add a constant to every generated sample, we can arbitrarily set the mean of the distribution. Similarly, by multiplying the samples we can arbitrarily set the standard deviation of the distribution. (With this, we essentially switched from standard normal distribution to general normal distribution.) But the multiplication and the adding of a constant is simply a linear transformation, a subclass of polynomial transformation. The crucial point is to note that when the transforming polynomial has an order of zero (i.e. adding of a constant), we can only set the mean (i.e. the first moment), when it has an order of one (i.e. multiplication and adding of a constant), we can set the mean and the standard deviation (i.e. the first and second moments). One would expect that by transforming the sample with a polynomial having

[4] Obviously, in this case the two indicators cannot be set independently, to the contrary: setting one of them automatically defines the other.

an order of two (i.e. quadratic term), we would be able to arbitrarily set the first three moments, that is, mean, standard deviation and skewness.

To investigate this, Fleishman analytically derived the first four standardized central moment of the distribution generated with the $Y = a + bX + cX^2 + dX^3$ transformation, where $X \sim N(0;1)$. (Obviously, he used a polynomial having an order of three, as the aim was to set the first four moments arbitrarily.) He obtained the following equations for the mean, variance, and μ_3 and μ_4 indicators of skewness and kurtosis:

$$\mu = a + c$$

$$\sigma^2 = b^2 + 6bd + 2c^2 + 15d^2$$

$$\mu_3 = 2c\left(b^2 + 24bd\right) + 105d^2 + 2$$

$$\mu_4 = 24\left(bd + c^2\left[1 + b^2 + 28bd\right] + d^2\left[12 + 48bd + 141c^2 + 255d^2\right]\right)$$

Now we can answer the previous question: this power transformation method can be used to generate a distribution with arbitrary μ_3 skewness and μ_4 kurtosis if and only if the above system of equations can be solved for the μ_3/μ_4 skewness/kurtosis in question. (Mean might be chosen to 0 and variance to 1, without loss of generality. Our figures will follow this convention, i.e. we will only plot skewness and kurtosis on them; mean will be assumed to be 0, variance to be 1, everywhere.)

It can be demonstrated that as a result we obtain a skewness/kurtosis plane that may be generated (i.e. where the above system of equations may be solved) being quite close to the theoretically possible[5] one.

Another favourable aspect of Fleishman's method is that it only requires random numbers from standard normal distribution – exactly what we obtain from our random number generator.

Clearly, the only complicated part is the solution of the above system of equation. Although it is a complex, non-linear system of equations, and can be solved only numerically (not analytically), it can luckily be solved off-line: the necessary coefficients can be calculated before the MC-simulation. This is important, as the solution of the equation is time-consuming. To sum up, the correct approach is to solve these equations for every possible skewness/kurtosis combination we will need, store the results and then give them to the simulation routine as a constant.

We used the GNU Scientific Library (GSL) to solve the non-linear system of equations. It is a free C library designed to support mathematical/scientific computa-

[5] There are skewness/kurtosis combinations which do not represent a distribution at all. In other words, correct probability distributions cannot have arbitrary skewness and kurtosis, namely $\mu_4 \geq \mu_3^2 + 1$ stands for every probability distribution. See (Ferenci 2009a) for more detail.

tions, developed under the GNU project, and licensed under GNU GPL (GNU 2009). We employed version 1.13.

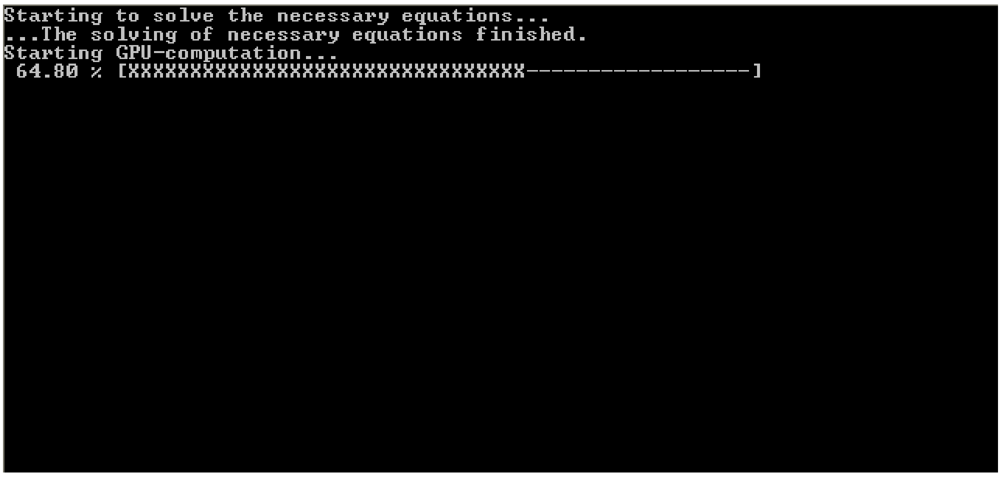
We used `gsl_multiroot_fsolver` to solve the system of equations[6] with initial values set to (0.0,1.0,0.5,0.5). (These were chosen to ensure that in normal case, the solution would be in fact the trivial transformation.) We used a simple solver algorithms (not using the derivatives, obviously), the so-called discrete Newton algorithm (`gsl_multiroot_fsolver_dnewton`). The solution was naturally iterative; we defined to stopping criteria: the number of iteration steps exceeds 1 000 (this was a backup, normally never active), and the residual is less then 10^{-5}.

3.4. User interface

As our program is computation-oriented with minimal user interaction, we designed a rather simple character-based user interface. (This also made coding easier, as we could write the program as a simple console application.)

The user interface has a *dual purpose*. First, it continuously informs the user on the progress of the computations. (This is especially important when calculations take more than minutes.) The user receives information on the current activity, and – in case of the most time-consuming GPU-computation – is also informed on the progress of that activity (with a "percentage complete" type feedback). The screenshot on Figure 2 shows this state of the user interface.

Figure 2. Screenshot of our program during the calculation phase



```
Starting to solve the necessary equations...
...The solving of necessary equations finished.
Starting GPU-computation...
 64.80 % [XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX------------------]
```

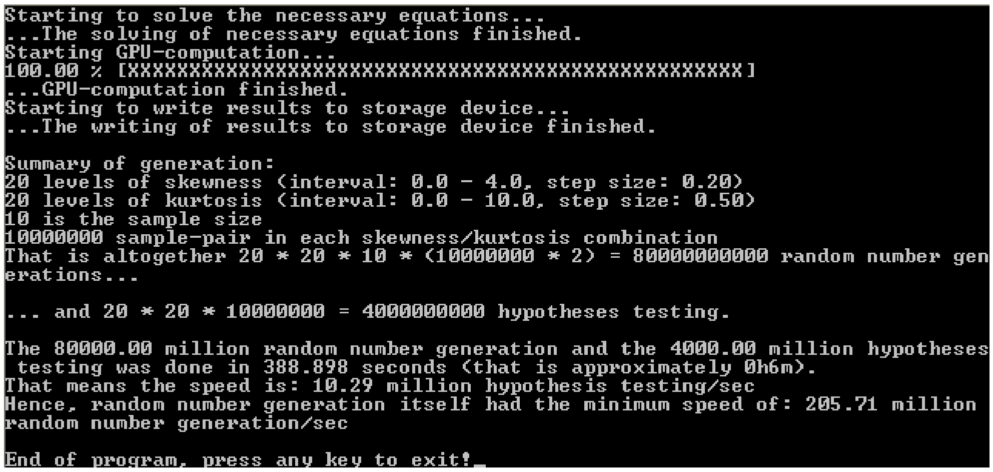
Source: own creation

[6] Multiroot stands for multidimensional root-finding.

The other aim of the user interface is to provide summary statistics when the calculation is finished. This is shown on Figure 3.

These statistics detail the parameters of MC-simulation (checked skewness and kurtosis levels, sample size etc.), the number of random number generations based on this and the time needed to complete the simulation. Based on the two latter information, the program displays the estimated speed of the computation in hypotheses testing/sec. It also provides a minimum estimate[7] for the speed of random number generation (in random numbers generated/sec).

Figure 3. Summary statistics provided by our program at the end of calculations



```
Starting to solve the necessary equations...
...The solving of necessary equations finished.
Starting GPU-computation...
100.00 % [XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX]
...GPU-computation finished.
Starting to write results to storage device...
...The writing of results to storage device finished.

Summary of generation:
20 levels of skewness (interval: 0.0 - 4.0, step size: 0.20)
20 levels of kurtosis (interval: 0.0 - 10.0, step size: 0.50)
10 is the sample size
10000000 sample-pair in each skewness/kurtosis combination
That is altogether 20 * 20 * 10 * (10000000 * 2) = 80000000000 random number gen
erations...

... and 20 * 20 * 10000000 = 4000000000 hypotheses testing.

The 80000.00 million random number generation and the 4000.00 million hypotheses
 testing was done in 388.898 seconds (that is approximately 0h6m).
That means the speed is: 10.29 million hypothesis testing/sec
Hence, random number generation itself had the minimum speed of: 205.71 million
random number generation/sec

End of program, press any key to exit!_
```

Source: own creation

No result is displayed on the console: the program saves every numerical result to file on the hard disk.

3.5. Visualisation

As we previously mentioned, the program saves the results as plain-text files. Although these can be viewed with any text editor (and their content interpreted), it makes analysis much simpler if we visualise their content. In other word, further (post)processing is needed, which we fulfilled with R scripts (R 2009) that we have written specifically for the purpose. We used R version 2.9.1; every figure showing results in this paper was generated this way.

[7] While the speed of hypothesis testing is a real, trustworthy indicator (as the random number generations form a part of hypothesis testing in this context), the speed of random number generation is well underestimated this way (as the time also includes calculations not related to random number generation, e.g. the performing of the statistical test).

3.6. On the measuring of kurtosis

Kurtosis – as a concept – is a complex abstraction, and its measuring is not at all unambiguous. (See (Hunyadi 2009) on this question.)

Classically the fourth standardized central moment (i.e. $\frac{E\left[(X-E[X])^k\right]}{\sigma^k}$ for $k=4$) is used to measure the kurtosis of a distribution. This takes the value of 3 for normal distribution, so sometimes the so-called excess kurtosis is also introduced:

$$\frac{E\left[(X-E[X])^4\right]}{\sigma^4}-3$$

This way, positive (excess) kurtosis indicates a leptokurtic, negative indicates a platykurtic distribution.

For our goals, these definitions are not fortunate. As we will generate distributions with substantial skewness, we cannot forget about the theoretical minimum of skewness. (See Footnote 5) When skewness equals just 5, the minimum possible kurtosis is 26 (compare to the minimum value of 1, in case of distributions that are not skewed at all). Should we use these kurtosis-measures, graphical representation would be overly complicated, as kurtosis would span over – literally – magnitudes. Interpretation would be also rendered more difficult, as the results for the higher values of skewness would be "shifted up". (Quite drastically, due to the quadratic nature of the theoretical minimum.)

A natural solution would be to subtract the theoretical minimum (determined utilizing the skewness) from every kurtosis value, in other words to switch from "kurtosis" to "kurtosis above the theoretical minimum". Formally this suggests a new kurtosis-definition that has the form $\mu_4-\left(\mu_3^2+1\right)$. (Graphically, this means to subtract the μ_3^2+1 parabola from every μ_4 coordinate.)

However, instead of that we chose to use the

$$\mu_4-\left(\mu_3^2+3\right)$$

new kurtosis definition. We will name this "FK-kurtosis".

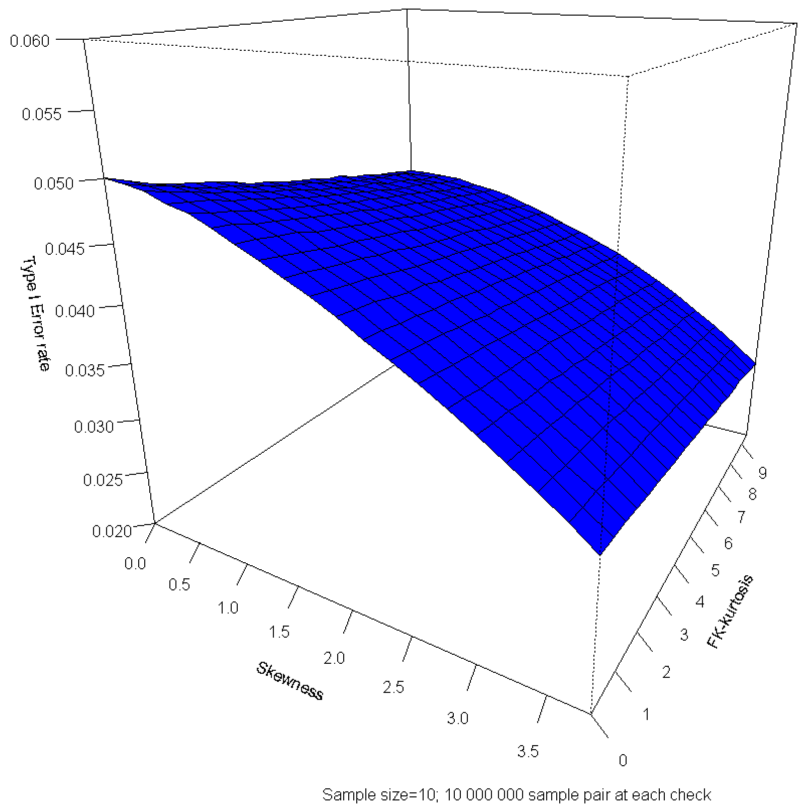
This has two advantages over the previous one. First, those distributions that have a positive FK-kurtosis can be surely generated by Fleishman's method, as it has problems[8] only with those distributions that are close to the theoretical minimum. Second, we ensure this way that the "zero skewness, zero kurtosis" point belongs to the normal distribution (just as with excess kurtosis).

[8] I.e. the necessary system of equations cannot be solved, see Subsection 4.2.

4. Results

Figure 4 shows the central result for our demonstrational example: the robustness of the Student's *t*-test for the violation of the normality assumption.

Figure 4. Robustness of the *t*-test for non-normality



Source: own creation

On the *z*-axis, we indicated the empirically found Type I Error Rate, while the *x*-*y* plane is a representation of the skewness/kurtosis space. (Using FK-kurtosis, as described in Subsection 3.6.)

One can clearly see that in case of normal distribution (zero skewness, zero excess kurtosis) the empirically found Type I Error Rate is almost exactly 0.05, the significance level. This confirms that the test is valid if its assumptions are met.

It is interesting to see how the empirically found Type I Error Rate deviates from the specified 0.05 as the samples become more and more non-normal. (To be more precise: the populations become more and more non-normal from which the samples are obtained.) This suggests that the test is not robust for the violations of the normality assumption: it loses its validity in case of samples where that assumption is not met. But we can be even more specific: it can be clearly seen that the test is far more sensitive for skewness than for kurtosis.

These results are all consistent with the literature data (Vargha 1996).

Note how *smooth* the figure is! This illustrates that the extremely high number of sample-pairs generated at each skewness/kurtosis point diminished the statistical fluctuation (due to the sampling error[9]) to almost zero. This is why very high computing performance was needed, what could be reached only with supercomputers, grid computing etc. traditionally, but what became widely available with our method based on GP-GPU.

As we always emphasized, the above results are uninteresting themselves. (These have been known virtually since the *t*-test has been introduced; long before not only MC-simulation, but also computers.) What is important: the *performance* we could reach when generating the results. (Because these are not scientific results nowadays, but our aim was to develop a framework for testing – that's the reason why we always called it only a demonstrational example. What we have to analyze is the framework itself, not the obtained results.) So now we will have a closer look on performance data.

4.1. Performance of the MC-testing

In the demonstrational example, we performed tests with populations having skewness from 0.0 to 4.0 (with 0.2 step size) and excess kurtosis from 0.0 to 10.0 (with 0.5 step size). It is important to note that we tested every possible combination of them (in other words, we completely "mapped" the skewness/kurtosis space, we iterated through every possible combination); hence, we had to perform $20 \cdot 20 = 400$ checking of validity.

For each checking we generated 10 million sample pairs. This meant $400 \cdot 10 = 4000$ million, or 4 billion hypothesis testing. Taking into account that the sample size was 10 (and sample-pairs were generated), it meant $4 \cdot 10 \cdot 2 = 80$ billion random number generation.

And this all was done in 389 seconds – again: on our 100€, middle-class, available-from-every-shop video card. In terms of speed, this means in excess to *10 million hypotheses testing per second*! (Purely the RNG itself has to produce a speed greater than 200 million random numbers generated a second.)

According to our measurements (performed by executing the very same simulation on both GPU and CPU), the GP-GPU computation is about *38 times faster*

[9] As such, this could be quantified, but the standard deviation due to sampling would be likely much smaller than even a single pixel.

than traditional CPU computation – even though the CPU in the comparison was a modern, dual-core processor.

This performance of GP-GPU is well enough for every common task, even if multiple parameters describe the departure from the assumptions and we have to iterate on them in a combinatorial way (i.e. using every possible combination; just as in our example).

4.2. On the flexibility of our framework

We already emphasised that *flexibility* was one of the primary priorities when designing our framework; now we will show two demonstrations for this principle.

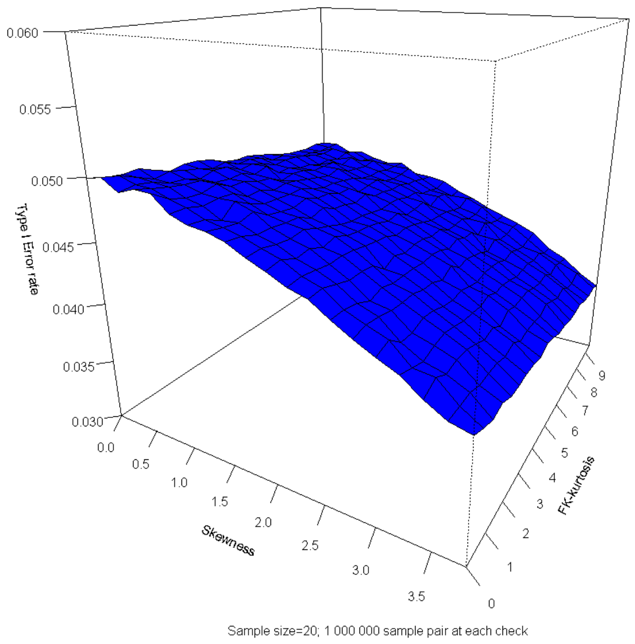
Perhaps the most important manifestation of this effort is the modularity of our program: we tried to isolate the description of the statistical test and the MC-testing parts as much as possible. This has two advantageous aspects: first, one can change the parameters of the testing by simply altering a few constants in the program; second, it is possible to add other statistical tests to be tested with leaving every other part of the program unchanged.

This means that our program is *highly scalable*: virtually any other test can be added (and, as we will see in 4.2.2 in more detail, there is no CUDA knowledge needed for this). Thus, we can use the framework to test any statistical test, and obtain results more interesting than our demonstrational example.

4.2.1. The effect of Central Limit Theorem

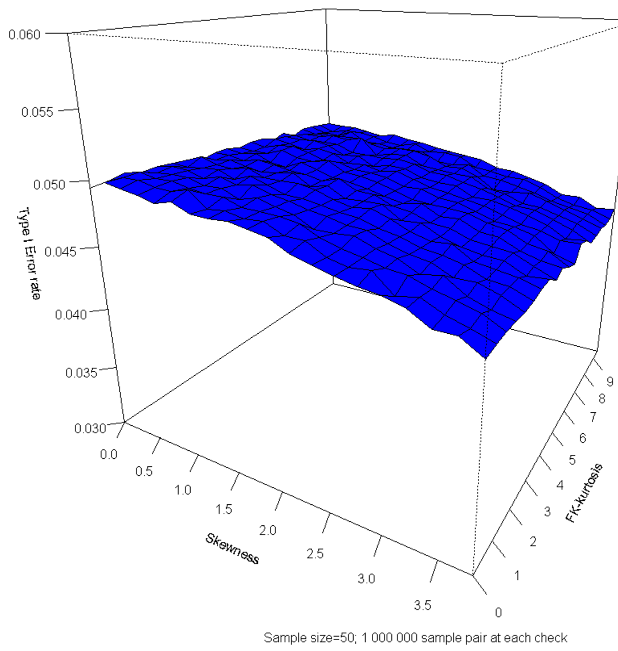
To demonstrate how simple it is to change the parameters of an MC-simulational testing, we re-run the test on the same example, but with sample sizes raised to 20, 50 and 100. Results are shown on Figures 5, 6 and 7.

Figure 5. The robustness of the *t*-test for non-normality with sample size 20



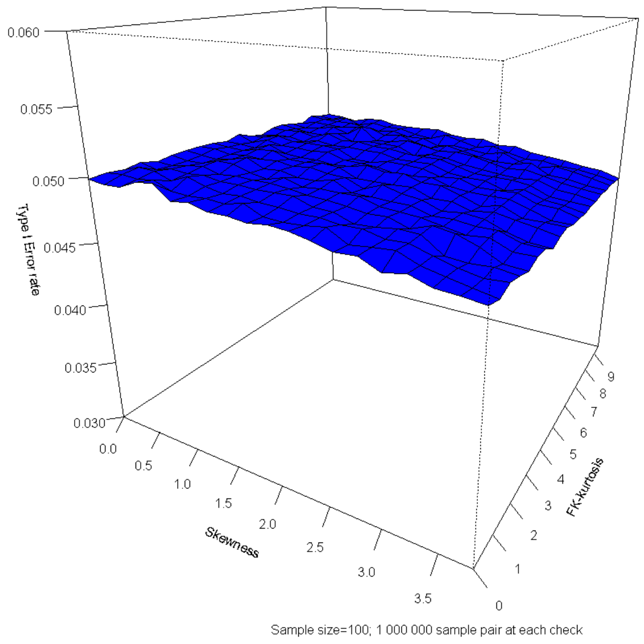
Source: own creation

Figure 6. The robustness of the *t*-test for non-normality with sample size 50



Source: own creation

Figure 7. The robustness of the *t*-test for non-normality with sample size 100



Source: own creation

One can clearly see the effect of the Central Limit Theorem: as it is well known from probability theory, with increasing sample sizes the sample mean follows more and more normal distribution, even if the population distribution was non-normal. As the test statistic of the *t*-test is based on the sample mean, this result in a higher robustness (increasing with sample size). At sample size=*100*, the test was virtually completely valid at every explored non-normality.

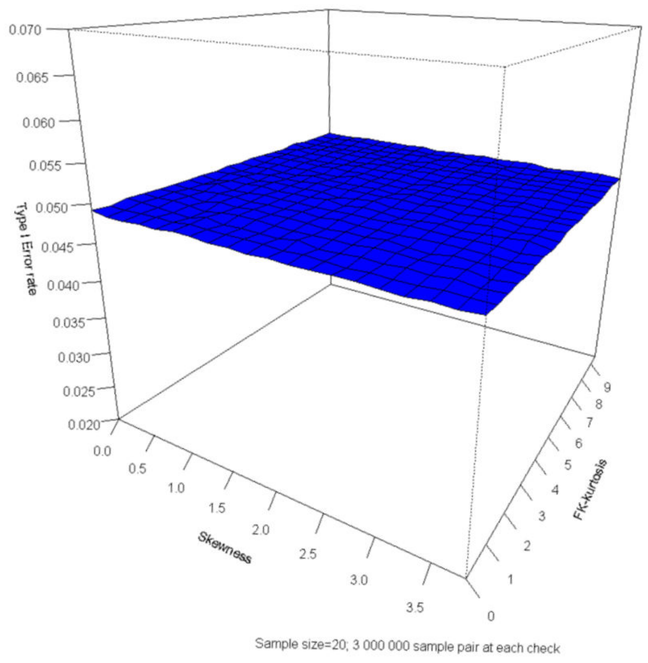
Again, not this "result" is interesting itself (it has also been known since the introduction of the *t*-test), but the way we obtained it: only a single number had to be changed in the program code for the above tests!

4.2.2. Expansion with other statistical tests

Perhaps even more important is the feature that our program can be very simply extended with other statistical tests. The adding of a statistical test to the framework only requires the specification of the test (as this part is completely separated from the code responsible for the MC-simulational testing).

To demonstrate this, we added the Mann-Whitney *U*-test (or Mann-Whitney-Wilcoxon test) to our framework, and then run exactly the same MC-simulational testing. Results are shown on Figure 8.

Figure 8. The robustness of the Mann-Whitney *U*-test for non-normality



Source: own creation

It is also important that this specification has to be done in standard C language – *no* CUDA or parallel computing knowledge is needed. (Parallelity is coded in other parts of the program.) The result is that other statistical tests can be added to our program with ease.

It can be clearly seen the Mann-Whitney *U*-test remains valid regardless of non-normality: it is robust from this point of view. This again empirically confirms the theoretical knowledge: if the PDFs of the populations have the same shape (they are just shifted) the non-parametric Mann-Whitney *U*-test is essentially distribution-free.

5. Conclusions and possibilities for further development

5.1. Summary

In this paper we discussed the question of testing the robustness of statistical tests, and especially the using of the so-called Monte Carlo (MC) method for that end. We have shown that this is an extremely computation-intensive method, which traditionally required supercomputers, grids etc., which made it available only to a few researchers.

However, a new approach, called GP-GPU can harness the extreme performance of modern – even low and middle class – video cards which can be magnitudes higher than that of CPU, for appropriate programs – just like MC-simulation. Environments for developing programs for GP-GPU are available (e.g. NVidia's CUDA).

Based on this idea, we developed a CUDA program for MC-testing the robustness of statistical tests. This can be run even with cheap, widely available video cards in ordinary PCs, despite that, it provides very high performance.

The true power of our work lies in the fact that the program we developed acts as a framework: virtually any statistical test may be included and tested – for which, no CUDA knowledge is needed at all.

Finally – as a demonstration – we performed the analysis of the well known Student's *t*-test; and – using this as a starting point – we also exhibited the main advantages of our framework.

5.2. Possibilities for further development

There are many ways to continue and improve this work. In this subsection, we will show those that seem to be the most promising.

There are many parameters we can experiment with – even in the case of the overly simple *t*-test, there is sample size and variance; from which, we have only investigated the effects of sample size. In case of more complex statistical tests, there might be far more parameters that might be changed.

The fact, that we can almost freely set the parameters of sample-generation might be used in other, essentially new ways, in addition to changing sample size and other simple descriptors. We might, for example, generate samples that deliberately do not meet the investigated test's null hypothesis – hence we can examine the power of the test. By generating samples representing different alternative hypotheses, we can also plot the approximate power function of the tests.

Fleishman's method (although having many advantageous properties) is not without problems. There are many alternatives (like Tukey's Generalized λ Distribution) which might be tried for the same end.

Finally, our framework might be in fact supplemented with statistical tests, i.e. a "test-bank" might be constructed by adding new and interesting tests to the already coded two. Such test-bank would be useful not only for research but also for education.

References

Cserny, L. 1997: *Mikroszámítógépek*. LSI Oktatóközpont, Budapest.

Ferenci, T. 2009a: *Kismintás biostatisztikai vizsgálatok néhány módszertani kérdése*. BCE TDK dolgozat, Budapest.

Ferenci, T. 2009b: *Kiskorú magyar populáció obesitassal összefüggő paramétereinek biostatisztikai elemzése*. BME VIK Diplomaterv, Budapest.

Fleishman, A. I. 1978: A Method for Generating Non-normal Distributions. *Psychometrika*, Vol. 43, Issue 4, 521-532. p.

Gentle, J. E. 2004: *Random Number Generation and Monte Carlo Methods*. Springer, New York.

GNU 2009: *GSL-GNU Scientific Library*. http://www.gnu.org/software/gsl/, 2009-12-31.

Hald, A. 1998: *A History of Mathematical Statistics from 1750 to 1930*. Wiley-Interscience, New York.

Hagerdoorn, H. 2009: *The history of Guru3D.com Part II.* http://www.guru3d.com/article/the-history-of-guru3dcom-part-ii, 2009-12-31.

Hunyadi, L. 2001: *Statisztikai következtetéselmélet közgazdászoknak*. Központi Statisztikai Hivatal, Budapest.

Hunyadi, L. – Vita, L. 2006: *Statisztika közgazdászoknak*. AULA Kiadó, Budapest.

Hunyadi, L. 2009: A negyedik mutatóról. *Statisztikai Szemle,* Vol. 87, Issue 3, 262-286. p.

Ketskeméty, L. 2005: *Valószínűségszámítás*. Műegyetemi Kiadó, Budapest.

Lee, P. M. 2009: *Bayesian Statistics: An Introduction*. Wiley, New York.

Matsumoto, M. – Nishimura, T. 1998: Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation*, Vol. 8, Issue 1, 3-30. p.

Matsumoto, M. – Nishimura, T. 2000: Dynamic Creation of Pseudorandom Number Generators. In Niederreiter H. – Spanier J. (eds.): *Monte Carlo and Quasi-Monte Carlo Methods 1998*. Springer Verlag, Berlin, 56-69. p.

McCullough, B. D. 2006: A Review of TESTU01. *Journal of Applied Econometrics*, Vol. 21, Issue 5, 677-682. p.

Microsoft 2009: *Microsoft Visual Studio*. http://msdn.microsoft.com/hu-hu/vstudio/default(en-us).aspx, 2009-12-31.

NVidia 2009a: *GeForce 9600 GT Specifications*. http://www.nvidia.com/object/product_geforce_9600gt_us.html, 2009-12-31.

NVidia 2009b: *NVidia CUDA Programming Guide Version 2.3.1.* http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf, 2009-12-31.

Owens, J. D. – Luebke, D. – Govindaraju, N. – Harris, M. – Krüger, J. – Lefohn, A. E. – Purcell, T. J. 2007: A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, Vol. 26, Issue 1, 80-113. p.

Podlozhnyuk, V. 2007: *Parallel Mersenne Twister.* http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/MersenneTwister/doc/MersenneTwister.pdf, 2009-12-31.

R 2009: *The R Project for Statistical Computing*. http://www.r-project.org/, 2009-12-31.

Rubinstein, R. Y. – Krose, D. P. 2008: *Simulation and the Monte Carlo Method.* Wiley, Hoboken.

Vargha, A. 1996: Az egymintás t-próba érvényessége és javíthatósága. *Magyar Pszichológiai Szemle*, Vol. 36, Issue 4-6, 317-345. p.