# A PRECISE EXECUTION TIMING ROUTINE FOR ENGINEERING SOFTWARE EVALUATION

**Magazinović, G.**

**Abstract:** Program execution time is one of the major parameters used in the performance evaluation of the computationally intensive engineering software. Unfortunately, computer systems based on Microsoft Windows platforms under the Intel's x86 architecture lack sufficiently precise timing capabilities needed for performing measurements of some short executions. Therefore, in this paper a precise timing routine capable of successfully measuring executions at the processor clock cycle level is proposed and discussed. Furthermore, its complete source code is provided. Finally, the utility of this timing routine is demonstrated on an example of comparative efficiency evaluation of two well-known two-point function approximations.

***Keywords:*** *Computational experiments, Performance evaluation, Software, Timing*

## 1 INTRODUCTION

The central processing unit (CPU) time needed for program execution is traditionally used as one of the major parameters in computationally intensive engineering software efficiency evaluation and comparison [1]. Furthermore, some authors have concluded that "in spite of its obvious shortcomings, the direct measurement of the CPU time is still the most reliable way of comparing different minimization algorithms" [2].

Execution timing may be surprisingly complex [3], since at least two problems are usually imposed: the available standard library routines are often not very precise, and the other processes running concurrently may significantly deteriorate the accuracy and utility of the obtained results.

When the computer systems based on Microsoft Windows platforms under the Intel's x86 architecture are concerned, a resolution of the available timers is given in Table 1. System timer is too low-resolution. A slightly better performance may be achieved by using the multimedia timer. Finally, the high-resolution performance counter provides the greatest precision. However, it can be shown that on some systems, the accuracy of that timer deteriorates significantly for the measurement readings below some 50 μs, Fig. 1. For the sake of completeness, a resolution of the StopWatch routine proposed in this paper is also included in Table 1.

Table 1. Resolution of the available timers

| Timer | Resolution/μs |
|---|---|
| System timer | 10,000 |
| Multimedia timer | 1,000 |
| High-resolution performance counter | 0.279[a] |
| StopWatch routine (this paper)[b] | 0.001[a] |

[a] Processor dependent; Intel Pentium III/1003.3 MHz

[b] Should not be confused with the same named timing package by Mitchell [4]

The influence of other processes may be reduced to some extent by giving the highest system priority to the measured code. However, it should be clearly realized

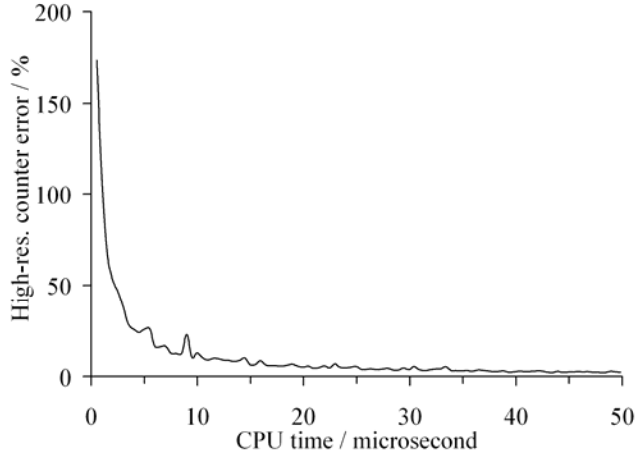


Fig. 1. Percent difference in readings between the high-resolution performance counter and a StopWatch routine (Intel Pentium III/1003.3 MHz, Microsoft Windows 2000 Professional SP4)

that some system processes are completely out of the user control [3]. Although this problem could play a prominent role, the full list of measurement issues is much larger [3, 5].

The software solution presented herein is developed to Microsoft Windows platforms and Intel's 32-bit and 64-bit x86 processor architectures. However, the key concepts presented are quite general and therefore, applicable to other platforms.

## 2 EXECUTION TIMING

Strictly speaking, there is a difference between the so-called wall time (real time) and the bare CPU time, where the wall time includes the CPU time and the system time. In what follows we would refer the measured time as a CPU time, although the majority of timing routines actually measure the wall time. That way, it will always cause overestimates of the true execution time [3]. Hopefully, if the system time is sufficiently bounded, the difference between the real (wall) time and the CPU time practically vanishes. Although the system time is a major source of timing overhead, it is not the only one. For example, the timing routine itself is a source of overhead too.

Different timing routines utilize different measurement units. Some routines deal with time units, usually milliseconds, while others apply some kind of counters. The value of the count is processor dependent. If the count is the cycle rate of the processor clock, the following timing equation apply

$$t_{\mathrm{CPU}} = \frac{c_{\mathrm{END}} - (c_{\mathrm{START}} + c_{\mathrm{OVERHEAD}})}{f_{\mathrm{CPU}}}, \tag{1}$$

where $t_{\mathrm{CPU}}$ is the CPU time measured, expressed in seconds, $c_{\mathrm{END}}$ and $c_{\mathrm{START}}$ are counter readings at the end and start of timing, respectively, expressed in processor cycles, while $c_{\mathrm{OVERHEAD}}$ is the timing routine overhead, expressed in processor cycles too. Finally, $f_{\mathrm{CPU}}$ is the processor clock frequency, expressed in cycles per second, Hz.

According to [6], the Intel time-stamp counter is a 64-bit model specific register (MSR) that is incremented every clock cycle. Its content is accessible by the `rdtsc` instruction, that loads the current count of the time-stamp counter into the EDX:EAX registers. On computer reset, the time-stamp counter is set to zero. Since the recent processor frequencies are usually in excess of 2 GHz, the extremely fine timing resolutions of less than a half of nanosecond are readily available.

As already mentioned in Introduction section, the concurrent processes running in the background could drastically deteriorate the accuracy of timing measurements. However, some other events might also impact heavily the utility of conducted measurements:

*(i) Cache effects*. If some data or instructions are not already present in the cache, the additional access to the slower memory is required. Such transactions significantly slow- down the overall performance. This unfavourable behaviour might be avoided to some level by a so-called *cache warming* [5] – the technique of filling memory into a cache before it is actually used.

*(ii) Out-of-order execution*. The out-of-order code execution becomes a standard feature of recent processors based on the Intel's architecture. That means that some program instructions could be executed earlier than it follows from its relative position in the program source code. A practical workaround is to execute some serializing instruction, i.e. the instruction that force completion of the entire preceding program code, e.g. `cpuid`, [5].

*(iii) Measurement overhead.* The timing routine itself produces some overhead, and this fact should be taken into account. Moreover, when short code segments are timed, the influence of the timing overhead could become predominant. In general, the mean timing overhead might be estimated by averaging the readings of multiple successive calls of the timing routine.

## 3 DESCRIPTION OF A STOPWATCH ROUTINE

This Section details the structure and operations performed by the StopWatch timing routine and the associated subroutines. The complete source code is provided in Fig. 2. Fig. 3 provides a sample main routine that calls the StopWatch(). Finally, in Fig. 4, a Fortran 90/95 wrapper of the StopWatch() routine is given.

The attached source code is thoroughly commented and hence self-explanatory. In order to better explain all related processes, in this Section some additional information are also provided. The code is successfully compiled, linked and tested with Microsoft Visual C++ Version 6.0 Professional compiler, enriched with the Processor Pack, readily available from the manufacturer's web site. If compiled by a different compiler, some minor changes may be required. Nevertheless, consultation of the respective compiler documentation could be helpful.

The proposed timing routine contains three C/C++ language functions: StopWatch(), GetCPUFrequency(), and GetRDTSC().

### 3.1 GetRDTSC()

This function, a heart of the proposed timing routine, is derived from the code segment written by the staff of Intel Corporation [5]. Actually, it is an inline assembly code that

```
#include <windows.h>                                        // Header file required

unsigned __int64  GetRDTSC()                                // READ TIME-STAMP COUNTER ROUTINE
{                                                           //   - derived from Pentium II Application Notes, [5]
    unsigned __int32 HighPart, LowPart;                     // Representation of 64-bit integer

    __asm                                                   // Inline assembly code
        {
        pushad                                              // Push all registers onto stack
        cpuid                                               // Force the in-order (serial) execution
        rdtsc                                               // Read the time-stamp counter
        mov   HighPart, edx                                 // Take content of the EDX register
        mov   LowPart,  eax                                 // Take content of the EAX register
        popad                                               // Pop all registers onto stack
        }
    return ((unsigned __int64)HighPart << 32) | LowPart;    // Return current state of the counter, in cycles
}

extern "C" double  GetCPUFrequency()                        // PROCESSOR FREQUENCY MEASUREMENT ROUTINE
{                                                           //   - derived from CPUInfo open source tool, [7]
    LARGE_INTEGER Counter, Frequency;
    double Counts, TStart, TEnd, CStart, CEnd, CPUFrequency;  // All variables are double due to simple arithmetics

    QueryPerformanceFrequency(&Frequency);                  // Read a high-performance timer frequency,
    CPUFrequency = (double)Frequency.QuadPart;              //   in counts per second
    Counts = CPUFrequency/1000;                             // Number of counts during a millisecond time frame
    QueryPerformanceCounter(&Counter);                      // Read a high-performance timer counter
    TStart = (double)Counter.QuadPart;                      // High-performance counter initial reading, in counts
    CStart = (double)GetRDTSC();                            // Time-stamp counter initial reading, in cycles
    do
        {
        QueryPerformanceCounter(&Counter);                  // Iterate
        TEnd = (double)Counter.QuadPart;                    //   until a predefined number of
        }                                                   //   counts expires
        while ((TEnd - TStart) < Counts);
    CEnd = (double)GetRDTSC();                              // Time-stamp counter final reading, in cycles
    CPUFrequency *= (CEnd - CStart)/(TEnd - TStart);        // Actual processor frequency
    return CPUFrequency;                                    // Return processor frequency, in Hz
}

extern "C" double  StopWatch(const char *Event)             // EXECUTION TIMING ROUTINE
{
    static double CPUFrequency, CStart, CEnd, COverhead;    // Retain values upon execution
    const double Calibrator = 125;                          // Should be set as appropriate, in cycles
    double  CPUTime;                                        // Final result

    if (!strcmp(Event, "START"))                            // "START" - initialize timer
        {
        SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_TIME_CRITICAL);  // Take thread priority
        CPUFrequency = GetCPUFrequency();                   // Current processor frequency
        CStart = (double)GetRDTSC();                        // Multiple calls to
        CEnd = (double)GetRDTSC();                          //   GetRDTSC -
        CEnd = (double)GetRDTSC();                          //   instruction serialization
        CEnd = (double)GetRDTSC();                          //   and cache warming
        COverhead = (CEnd - CStart)/3 + Calibrator;         // Total overhead, in cycles
        CStart = CEnd;                                      // Number of cycles at the start of measurement
        return 0;                                           // End of timer initialization
        }
    else
        {                                                   // "STOP" - measure elapsed time
        CEnd = (double)GetRDTSC();                          // Number of cycles at the end of measurement
        CPUTime = (CEnd - CStart - COverhead)/CPUFrequency;  // Elapsed time evaluation
        SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_NORMAL);  // Release thread priority
        CStart = CEnd;                                      // Enable multiple measurement (if required)
        return (CPUTime > 0.0) ? CPUTime : 0.0;             // Return elapsed time, in seconds
        }
}
```

Fig. 2. StopWatch routine source code listing

executes the `cpuid` and `rdtsc` processor instructions, where the `cpuid` sole intention in this case is to ensure that all preceding instructions in the code are completed before execution of the `rdtsc`. This way, the `cpuid` acts as a code serialization instruction.

GetRDTSC() passes no arguments. Its return value is a current status of the time-stamp counter, expressed as an unsigned 64-bit integer.

### 3.2 GetCPUFrequency()

This function performs processor clock speed measurement. It is derived from the code segment written by Chad Austin [7].

This function passes no arguments too. Its return value is a processor clock speed, expressed in Hertz. Due to straightforward arithmetic (the unsigned 64-bit integer arithmetic is not supported by many systems), all function variables are declared and treated as double floating-point values.

The processor clock speed is determined as a number of processor cycles executed during the specified time frame. In this case a millisecond time frame is adopted. This function utilizes a high-resolution performance counter functions QueryPerformanceCounter() and QueryPerformanceFrequency(). Since the reading of the QueryPerformanceFrequency() function is processor dependent, it is combined with the GetRDTSC() to ensure the processor clock speed in cycles per second (Hz).

### 3.3 StopWatch()

This is a user callable timing routine. It expects one argument (control string that directs further execution) and returns the elapsed time in seconds. As in the case of GetCPUFrequency() function, all variables are declared and treated as double floating-point values.

Program execution is determined by a control string named `Event`. If the `Event` is equal to `START`, the program initializes the stopwatch. Otherwise, the program determines the elapsed clock cycles and, by using the already determined processor clock speed, calculates the elapsed time in seconds.

StopWatch() initialization comprises of five operations:

*(i) Taking priority*. Initially, by calling the SetThreadPriority() function, program takes the maximum thread priority, to minimize the influence of other processes that run concurrently. However, it should be clearly understood that the real impact of this measure is actually limited - some computer processes could not be suppressed by any user action, by any means. For instance, the unexpected disk activity is one of the signs of such processes.

*(ii) Processor speed determination.* The program calls the GetCPUFrequency() to obtain the processor clock current speed.

*(iii) Instruction serialization and cache warming*. The program performs four consecutive calls to GetRDTSC(), to minimize the effects of instruction and memory cache. The auxiliary purpose of this operation is to provide sufficient information for the GetRDTSC() function overhead determination.

*(iv) Timing overhead.* In this operation the program determines the total overhead incurred by the timing routine. In general, this value could be determined by summing up the individual overheads of the involved routines. In the proposed procedure the total overhead is composed of two parts. The first one is overhead caused by the GetRDTSC() function alone. The second part is the overhead caused by other processes. The second overhead is taken into account by the `Calibrator` parameter. This constant, expressed by the number of clock cycles, is processor specific and should be determined by experiment.

Table 2. Overhead measurement results ($N$ = 10,000 StopWatch runs)

| Computer system | A | B |
|---|---|---|
| Processor architecture | 32-bit | 64-bit, dual core |
| Processor clock speed, MHz | 2593.7 | 2133.4 |
| Total overhead, cycles | 737 | 418 |
| Mean calibrator, cycles | 220 | 125 |
| Calibrator standard deviation, cycles | 125 | 27 |

Table 2 provides some overhead measurement results obtained on two workstations. Each measurement contained 10,000 consecutive StopWatch runs. Furthermore, each measurement has been repeated five times. The presented data are those that expressed least standard deviation of the measurement results.

*(v) Timer restart.* Finally, the program restarts the `CStart` variable by assigning the current time-step counter value. After that, the program execution is returned to the caller routine. The corresponding return value is arbitrary set to zero.

Since all function variables are declared as static, their last values will be retained upon the next call of the StopWatch() routine.

Similar to the StopWatch() initialization, its measurement phase comprises of four operations:

*(i) Timing*. First, the program reads the current state of the time-stamp counter and calculates the elapsed time since the timer's last initialization. During calculation, the timing routine overhead is taken into account.

*(ii) Releasing priority*. Since the measurement is completed, the StopWatch() routine releases the time-critical thread priority. This is achieved by the second call of the SetThreadPriority() function and setting normal thread priority.

*(iii) Timer restart*. In this step, the program restarts the `CStart` variable again by assigning the current time-step counter value. This operation may be valuable in the cases when more consecutive measurements are required. However, it should be taken into account that the timing process already lost its thread priority and the greater result fluctuations may be expected.

*(iv) Suppressing misleading results.* Sometimes, when the extremely short code sections are timed, it is possible that, due to the predefined value of the `Calibrator` parameter, the calculated elapsed time becomes negative. In such occurrences, the StopWatch() routine return value is arbitrary set to zero.

### 3.4 Sample main routine

In Fig. 3, a typical application of the StopWatch() routine is presented. Furthermore, the provided source code contains all the necessary function prototypes and data declarations.

If a series of consecutive measurements is required, the main routine has to be modified. Two approaches are possible. The first one is by multiple calls to StopWatch() and multiple use of the `STOP` argument, except the first time, when the `START` argument is applied, and the second one, by multiple calls to StopWatch() and interchangeable use of the `START` and `STOP` arguments. The latter approach, although slightly slower, has a better behaviour regarding the possible influence of other processes running concurrently.

```
#include <stdio.h>                                          // Program header and function prototypes
extern "C" double  StopWatch(const char *);
extern "C" double  GetCPUFrequency(void);

void main()                                                 // SAMPLE MAIN ROUTINE
{
    double  CPUTime;

    printf ("CPU Frequency = %7.1f MHz\n", GetCPUFrequency()/1.0e6);
    StopWatch("START");
    // ...
    // ...                                                  // Evaluated code comes here
    // ...
    CPUTime = StopWatch("STOP");
    printf ("CPU Time =%15.9f s\n", CPUTime);
}
```

Fig. 3. Sample main routine source code listing

### 3.5 Fortran 90/95 wrapper

For those who prefer a Fortran programming language in their software development, a simple Fortran to C/C++ wrapper is provided in Fig. 4. The code is successfully compiled, linked and tested with the Compaq Visual Fortran Professional Version 6.6C compiler. For other Fortran implementations some minor code changes may be required. In any case, consultation of the respective compiler documentation could be helpful.

```
REAL(8) FUNCTION  FStopWatch(Event)                         !  StopWatch ROUTINE IN FORTRAN 90/95

IMPLICIT NONE
INTERFACE                                                   !  Fortran call to C function interface
   REAL(8) FUNCTION  StopWatch(Event)
      !DEC$ ATTRIBUTES ALIAS:'_StopWatch' :: StopWatch
      CHARACTER(6) :: Event
   END FUNCTION
END INTERFACE
CHARACTER(6) :: Event

      FStopWatch = StopWatch(Event)                         !  Actually, FStopWatch calls the StopWatch

END FUNCTION  FStopWatch
```

Fig. 4. StopWatch Fortran 90 wrapper source code listing

## 4 EXAMPLE: A COMPARISON OF TWO-POINT FUNCTION APPROXIMATIONS

For example application, a comparison of direct function evaluation and two mid-range function approximations is selected.

The purpose of this example is twofold. First, it should verify the hypothesis that the appropriate function approximation scheme might provide some gain when the computationally expensive functions are evaluated. Second, it should provide some insight regarding the relative computational efficiency of these two approximations.

For timing purposes the cost function of the cam design problem (Problem No. 332 of [8]) is utilized.

### 4.1 Problem statement

The cam design problem has two design variables, two general inequality constraints and four design variable bounds. A feasible starting point is (0.75, 0.75) and the minimum cost function value is 114.95 at the point (0.911, 0.029).

The cost function is given with

$$f(\mathbf{x}) = \frac{\pi}{3.6}\sum_{i=1}^{100}\Big\{\big[\log(t_i) + x_2\sin(t_i) + x_1\cos(t_i)\big]^2 + \\ + \big[\log(t_i) + x_2\cos(t_i) - x_1\sin(t_i)\big]^2\Big\}, \tag{2}$$

where

$$t_i = \pi\left(\frac{1}{3} + \frac{i-1}{180}\right), \qquad i = 1,...,100. \tag{3}$$

Due to a 100 member series sum, a significant computation costs are incurred by the frequent trigonometric and logarithmic functions evaluation. By using appropriate function approximation it is reasonable to expect that some reduction in the computational costs might be obtained. In this case, a Generalized Convex Approximation (GCA) of Chickermane and Gea [9] and Two-Point Adaptive Nonlinearity Approximation (TANA-3) of Xu and Grandhi [10] has been used.

### 4.2 Timing results

Timing measurements were performed during the actual optimization process by the RQPOpt v2.0 design optimization package [11]. The test runs were performed on a Hewlett-Packard xw4400 workstation (2.133 GHz, dual core, 64-bit processor) in double precision arithmetic using the Compaq Visual Fortran Professional Version 6.6C compiler and Microsoft Windows XP Professional SP2 operating system.

The measurement comprised of three test runs. In the first one all function values were obtained by the direct calculation. During the second and third test runs, all function values were approximated, whenever possible, by the GCA or TANA-3 approximations, respectively.

After 24 iterations and 34 line search cost function evaluations, all three test runs converged to the same point $\mathbf{x}^* = (0.909, 0.033)$ with $f(\mathbf{x}^*) = 115.028$ and the maximum constraint violation $V(\mathbf{x}^*) = 8.28\times10^{-5}$.

Timing results are summarized in Fig. 5. Since both approximation schemes need some initially unavailable information, at least the first function evaluation is

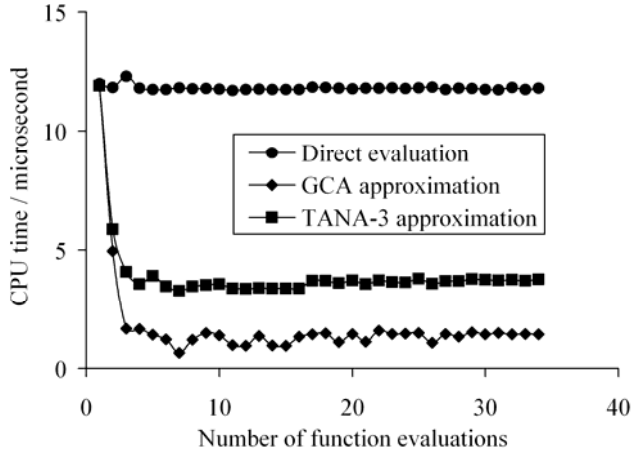


Fig. 5. Timing results

performed by the direct calculation, irrespective of the function approximation mode.

If the first five cost function evaluations or approximations are omitted in order to exclude some start-up effects, the mean execution times are as follows: 11.79 μs for direct function evaluation and 1.30 and 3.59 μs for GCA and TANA-3 approximations, respectively. In other words, the GCA approximation requires 36% calculation time with respect to the TANA-3 approximation.

During the tests all executions were also measured with a CPU_TIME, a Fortran 95 standard intrinsic subroutine provided by Compaq Visual Fortran Professional Version 6.6C [12]. However, all measurement results were equal to zero.

In order to obtain a meaningful time measurements with the CPU_TIME, it was necessary, for each timing, to execute a multiple function evaluations in a loop. All other measurement circumstances were equal to the previous single-evaluation tests. The mean execution times during 100,000 consecutive evaluations are summarized in Table 3. For evaluation purposes the corresponding StopWatch times are also included.

Table 3. Mean execution times during 100,000 evaluations in a loop, μs

| Timing routine | CPU_TIME[a] | StopWatch |
|---|---|---|
| Direct evaluation | 11.56 | 11.54 |
| GCA approximation | 0.99 | 0.98 |
| TANA-3 approximation | 3.20 | 3.19 |

[a]Compaq Visual Fortran Professional Version 6.6C

The generally shorter execution times recorded could be attributed to the absence of cache operations. Namely, during the cost function multiple evaluations all instructions and data remain into the cache unchanged. These results lead to an important conclusion that it is improper to directly compare a single-evaluation with multiple-evaluation measurement results. Otherwise, the times provided by the StopWatch and CPU_TIME routines are quite comparable.

## 5 CONCLUSIONS

In the paper, a precise execution timing routine is proposed and discussed. This routine could be a valuable tool in evaluation of engineering algorithms and software.

Execution timing is prone to various side effects that may deteriorate the accuracy and utility of performed measurements. Therefore, as in all other measurements, some common sense measures are always recommended: perform multiple measurements to obtain results repeatability and to justify its accuracy.

In general, dual core processors provide more stable timing. Such behaviour could be mainly attributed to a better management of the processes running concurrently.

After the code compilation and initial testing it is reasonable to collect the provided timing routines into the self-contained object library, say StopWatch.lib, to make them easily accessible and callable whenever required.

**Acknowledgements**
Portions of the source code presented herein are based on code segments written and published by Chad Austin and the staff of Intel Corporation. The author acknowledges their invaluable contribution.

In addition, special thanks to Professor Klaus Schittkowski for his valuable comments.

**References:**

[1] Crowder, H., Dembo, R.S., Mulvey, J.M. (1979). *On reporting computational experiments with mathematical software*, ACM Transactions on Mathematical Software 2, pp. 193-203.

[2] Miele, A., Gonzales, S. (1981). *Methods for evaluating nonlinear programming software*, in: Mangasarian, O.L., et al. (Eds.), Nonlinear programming 3, Academic Press, New York.

[3] Bryant, R.E., O'Hallaron, D.R. (2003). *Computer systems: A programmer's perspective*, Prentice Hall, Englewood Cliffs.

[4] Mitchell, W.F. (1997). *StopWatch User's Guide Version 1.0*, NISTIR 5971, available from: http://math.nist.gov/mcsd/Software.html (accessed 10 February 2008).

[5] Intel (1998). *Using the RDTSC instruction for performance monitoring*, Pentium II application notes, Intel Corporation, Santa Clara.

[6] Intel (2006). *Intel 64 and IA-32 architectures software developer's manual*, Vol. 3A: System programming guide, Part 1, Intel Corporation, Santa Clara.

[7] Austin, C. (2007). *CPUInfo open source library, Ver. 1.0.0*. Available online at: www.aegisknight.org/cpuinfo (accessed 17 May 2008)

[8] Schittkowski, K. (1987). *More test examples for nonlinear programming codes,* Lecture notes in economics and mathematical systems, No. 282, Springer, Berlin.

[9] Chickermane, H., Gea, H.C. (1996). *Structural optimization using a new local approximation method*, International Journal for Numerical Methods in Engineering 39, pp. 829-846.

[10] Xu, S., Grandhi, R.V. (1998). *Effective two-point function approximation for design optimization*, AIAA Journal 36, pp. 2269-2275.

[11] Magazinović, G. (2005). *Two-point mid-range approximation enhanced recursive quadratic programming method*, Structural and Multidisciplinary Optimization 29, pp. 398-405.

[12] Compaq (1999). *Compaq Fortran: Language reference manual*, Compaq Computer Corporation, Houston.

**Author:** Magazinović, Gojko, Associate Professor, Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture, University of Split, Split, Croatia, www.cadam.riteh.hr/whoiswho/CVs/Magazinovic_Gojko.htm