BIG INTEGER DATA TYPES — USAGE AND PERFORMANCE COMPARISON

M. Mikac, R. Logožar, M. Horvatić, E. Dumić

University North (CROATIA)

Abstract

Students in STEM fields often need to propose and develop programming solutions of different levels of complexity, for various scientific, technical, and engineering problems. In order to solve such tasks, they must have sufficient general programming skills and the knowledge of some standard programming language, such as C/C++, C#, Python, Java, Kotlin, JavaScript, Pascal. Alternatively, specialized programming languages and computing tools such as Matlab, Scilab, or GNU Octave, may be better suited for solving some of the problems and could sometimes be considered as a first choice.

In this paper, the authors discuss the use of the nonstandard integer data types, so-called big integers, being able to store and handle large integer values that can be only represented with more than 64 bits. They analyse the possibilities of using those data types in education and programming practice for the well-known general-purpose languages: C/C++, C#, Java, JavaScript, Kotlin, and Pascal, as well as for the specialized high-level programming tools like free Matlab alternatives - Scilab and GNU Octave. For the languages without direct support for the big integers, the authors examine the possibility of inclusion of the freely available libraries, which include such types. After that, they measure and analyse the performance of the calculations with the big integer types on the standard Windows based, computing platform by using the fast-growing benchmark functions, such as factorials. The obtained results are presented, discussed, and compared.

Keywords: big integer, programming, performance comparison, benchmark, complexity, JavaScript, C/C++, C#, Python, Kotlin, Java, Scilab, Pascal, Delphi, factorial, source code.

1 INTRODUCTION

Students in the STEM fields often need to develop programming solutions of different levels of complexity for various scientific, technical, and engineering problems. In order to solve such tasks, they must have sufficient general programming skills and the knowledge of some standard programming language, such as C/C++, C#, Python, Pascal, Kotlin, Java, JavaScript. Alternatively, the specialized programming languages and computing tools such as Matlab, Scilab, or GNU Octave, may be better suited for solving some of the problems.

While trying to provide a wider perspective of our programming classes and relate them to other technically oriented courses, such as digital electronics or computer architecture, the implementation of the basic numerical calculations is one of the main tasks and topics. On the other hand, when teaching the programming variables and data types, the latter are often covered only lightly, and without proper elaboration. Many programmers in the high-level programming languages are used to the ready-made solutions and are not very interested in the true mechanisms behind the coding and storing of the data types they use, or the in the ways the calculations are performed with them. However, when confronted with some special programming tasks or just the school extreme examples requiring either huge size of integer types or the extra demands for the precision of the floating-point types, one may start investigating the possibilities of the use of the nonstandard data types in the language(s) she or he uses.

The authors of this paper took that challenge, too and investigated the implementations of the big integer types in several programming languages. A critically inclined reader might ask what would be the purpose of such extreme calculations, and isn't it just a computational *l'art pour l'art*. However, besides the ever-growing inflations of all kinds, the examples from combinatorics or simply the need for the large and fully precise fixed-point calculations justify such numerical investigations.

Large integer numbers that cannot fit in standard 32 or 64-bit integers supported in all general-purpose programming languages are something that is required in certain type of calculations. Some of the simplest examples presented to our STEM undergraduate students, such as factorial calculation, were identified as ideal candidates for discussion about the usage of special data type enabling arithmetic

ISBN: 978-84-09-42484-9

for, often-called, big numbers or big integers. Essentially, it is about data types or structures that should allow arbitrary range, precision and bit-length to store extremely large numbers.

After this introduction, the paper starts with important section explaining the topic of the big-integer support in the collection of the today widely known and popular general-purpose programming languages, both the compiled and interpreted ones. All programming languages that are later used to implement performance test programs are listed, including the information about native or extension-based support for integer numbers of arbitrary range. Next section deals with the methodology of our investigations, setting the calculation of the factorials as our benchmark. It gives all source codes used for the benchmarks. After that, the results section presents and visualizes our measurements and comments our findings. Conclusions and some remarks about possible future work related with this subject are given in last section.

2 THE BIG INTEGER SUPPORT IN ANALYZED PROGRAMMING LANGUAGES

This paper is focused on analyzing the support for integer calculations with large values – as large values we consider integers not being able to store their values in standard 32-bit or 64-bit binary memory words. Usually, that kind of integer numbers are called big integers or arbitrary range integers. For that kind of "big numbers", it is expected that there is no practical limit to their range or precision except the limits implied by available memory. Of course, for integers we mainly discuss their range, while the precision is considered to be able to provide usage of any successive integers in given range.

In order to document the abilities of general-purpose languages and specialized programming tools, certain official documents had to be analyzed. Additionally, for languages not supporting big integers by default, natively, the way of using additional libraries, if any, had to be investigated and documented.

Being involved with high-education process in STEM fields, the authors have some experience with different kind of programming related courses but also with applying the programmer's skills when solving some real-world problems. During some previous studies, we noticed that we never analyzed the support for big integers in programming languages we teach our students. One simple mathematical definition resulting in fast-growing numeric function, calculating a factorial lead us to the problem of correctly calculate factorials – some of the detected issues and ways to solve it were described in [1]. In addition to calculation issues, this paper not only expands the list of programming languages checked and used, but it adds performance measurement in order to somehow value the quality of implemented big integers support in different languages.

2.1 General purpose programming languages

General-purpose languages covered in this paper include, alphabetically ordered: C/C++, C#, Java, JavaScript, Kotlin, Pascal, and Python. The languages were selected based on our educational experience (C/C++, JavaScript, Kotlin, and Python are used in standard education process on our institution), personal preference and history of use (C#, Pascal, Java), as well as the language popularity – popularity lists maintained by TIOBE [2] or PYPL [3] list most of those (except Pascal as first authors historical favourite choice, due his Delphi experience) at top of others. Certainly, other languages could be analysed and discussed, but we decided to limit on selected and not expand the scope of the paper.

2.2 Specialized high-level programming tools

Similar to general-purpose languages, in our education process we encourage students to use specialized tools and programming languages included with them – recently, we were evaluating free Matlab alternatives – Scilab and GNU Octave, mainly comparing their performance (as interpreted environments) [4] and ability to provide quite good calculation performance when using suggested vectorization approach [5]. So, it seemed natural for us to include big integer analysis for Scilab or GNU Octave in this paper. Unfortunately, as it shall be seen in following sections, only limited usage of big integers was successfully accomplished in Scilab, with poor performance results.

2.3 Integer numbers and their binary representation

The paper focuses on integer numbers – floating-point numbers are not considered. It is well-known fact that integer number binary representation is quite straightforward - the most important information related to their binary representation is the number of bits used to store the number.

Integers are binary represented as sets of bits of certain (predefined, by data type) length. Bit-length directly relates to the value range. For x bits being used, it is clear that there are 2^x possible combination of bits (varying from $0_{(10)}$ – all zeros - to $2^x - 1_{(10)}$ – all ones – for example, a standard 8-bit word can store all combinations from $00000000_{(2)} = 0_{(10)}$ to $11111111_{(2)} = 2^8 - 1 = 255_{(10)}$. The programming languages use different types of integers, but in general all languages support up to 64-bit integers.

In order to binary define certain positive integer number *n*, as described in [1], certain number of bits, *NoBD* (Number of Binary Digits) must be used. The mathematics behind it, show that number of binary digits can be calculated using the logarithmic function as:

$$\mathrm{NoBD}(n) = \lfloor \log_2 n \rfloor + 1$$

Similar, *NoDD* (Number of Decimal Digits) can be calculated as:

$$\mathrm{NoDD}(n) = \lfloor \log n \rfloor + 1$$

For all integers that require *NoBD* larger than 64 bits, the representation and arithmetic limitations may appear in certain languages not supporting integers of arbitrary bit-length.

2.4 Programming languages comparison

The programming languages selected for analysis in this paper differ significantly. We could compare or group them using different criteria – the way of code execution (compiled vs. interpreted, with additional remarks about languages that may use intermediate code, being translated or “interpreted” prior execution on target machine), the way of declaring variables (typed vs. untyped languages, or more precisely – strongly or static typed where the programmer has to explicitly define variable data type prior execution against dynamical typed where the data type is determined during runtime), etc.

Table 1. Native big-integer support in analysed programming languages

Language	***Native Support Keyword / class / type***	***External Library Keyword / class / type***
C/C++	✘	MPIR – Multiple precision integers [7] C data type – mpz_t
C#	✔ System.Numerics - BigInteger	
Delphi	✘	Delphi BigNumbers [8] uses BigIntegers + BigInteger type
Java	✔ import java.math.BigInteger	
JavaScript	✔ BigInt()	
Kotlin	✔ import java.math.BigInteger	
Kotlin/Native	✘	✘
Pascal	✘	TForge [9], [10] *FNX – Multiprecission numbers [11]* *GMP for FreePascal [12]*
Python	✔ import math + standard variable	
Scilab	✘	BigInteger toolbox [13] bigint type
GNU Octave	✘	✘

Out of all analysed programming languages, only C/C++ and Pascal can be considered compiled languages with no doubt. Both can be compiled and linked to executable code. Java uses intermediate code when compiled, and uses virtual machine (JVM) to execute the code – therefore it is somewhere “in the middle”, when discussing execution and compilation. Kotlin, as “simpler Java”, also relies on Java and JVM, but there are some additional “compilers” or translators including native compiler to executable code (so called, Kotlin Native [6]), allowing the execution without virtual machine. Another popular language, C#, compiles to intermediate byte-code and uses .NET environment at runtime.

The compilation and execution status of other general-purpose languages is not so clear – generally, JavaScript and Python were always considered interpreted languages, but lately there were some concept changes and optimized approach allow just-in-time (JIT) compilation or other similar techniques that may significantly improve execution performance.

When considering variable declaration as a criterion – C/C++, C#, Pascal, Java, Kotlin are considered statically typed languages, while JavaScript and Python do not require variable type to be defined in code – they are considered dynamically typed languages.

The most important information for this article was whether certain language natively supports working with arbitrary range integers. The Table 1 shows native big-integer support in analysed languages. For languages not supporting big-integers natively, reference to available libraries is given.

For Pascal, only *TForge* library was used in our tests, other two solutions are just documented. As seen in the table, there are two languages with no native support for big-integer type for which we were not able to find any external libraries or other solutions that may allow big integer calculations – Kotlin/Native and GNU Octave. We would appreciate our readers having related information to get in touch with us. For all other languages, we managed to execute performance tests.

3 METHODOLOGY

To make things as simple as possible, we decided to create simple test programs for each of analyzed programming languages and measure the time required to execute the program with certain parameters. Our intention was to determine and compare performances by measuring average time (duration) of process of generating a list of factorials. The factorial calculation function was intentionally selected as fast-grow function - as explained in [1], factorial for number as low as 21 results with integer value not being able to fit 64-bit memory words. In other words, mathematically, for calculating factorial for numbers n > 20, arbitrary range big integers must be applied in order to get valid results.

Table 2. Exact factorial calculation results for test numbers n above 20, with number of binary digits (NoBD) and number of decimal digits (NoDD)

n	*Factorial n!*	*NoBD*	*NoDD*
20	2.432.902.008.176.640.000	62	19
21	51.090.942.171.709.440.000	66	20
22	1.124.000.727.777.607.680.000	70	22
23	25.852.016.738.884.976.640.000	75	23
24	620.448.401.733.239.439.360.000	80	24
25	15.511.210.043.330.985.984.000.000	84	26
26	403.291.461.126.605.635.584.000.000	89	27
30	265.252.859.812.191.058.636.308.480.000.000	108	33
33	8.683.317.618.811.886.495.518.194.401.280.000.000	123	37
34	295.232.799.039.604.140.847.618.609.643.520.000.000	128	39
35	10.333.147.966.386.144.929.666.651.337.523.200.000.000	133	41
40	815.915.283.247.897.734.345.611.269.596.115.894.272.000.000.000	160	48
45	119.622.220.865.480.194.561.963.161.495.657.715.064.383.733.760.000.000.000	187	57
50	30.414.093.201.713.378.043.612.608.166.064.768.844.377.641.568.960.512.000.000.000.000	215	65

Exact factorial calculation results for certain relatively low numbers n is given in table 2. In addition to factorial value, two informative columns are included – one showing *NoBD* (number of binary digits), as described in 2.3, and other showing number of decimal digits (*NoDD*). It can be seen that 128-bit length is reached for n = 34. For all test programs, the exact list was used to manually check validity of obtained results. All the test programs, for all languages managed to get correct values!

The test programs were implemented so that the execution depends on three parameters – N_{from} and N_{to} were used as first and last number for which the factorial was calculated, and N_{rep} as parameter defining number of repeated procedures. For instance, with N_{from} = 1, N_{to} = 1000 and N_{rep} = 50, the

program repeated generation of all factorials for numbers in interval from 1 to 1000 for 50 times. In order to measure the performance, total time was measured and finally divided with N_{rep} to get average time required to generate list of factorials. For each number *N*, complete factorial calculation is executed (complete loop multiplying all numbers from 1 to N) – therefore, our performance tests dominantly benchmark big integer multiplication.

Since the scenario of calculating all the factorials for such a large and continuous range/interval cannot be considered realistic, we intentionally skipped possible performance improvement – when calculating 1000!, we could determine all other factorials from 1 to 999! using single pass through for-loop, but instead we decided for brute-force test repeating the calculation for each number in [N_{from}, N_{to}] range.

In order to measure the time required to perform complete calculation process, proper time measurement functions had to be used. Since all the languages used for our performance comparison offer different functions to check current time (with different precision), we had to check documentation and adapt to each any every language.

3.1 Development tools

Modern development environments are often available for free (such as Community editions of professional tools like Microsoft Visual Studio or Intellij IDEA, or completely free such as Microsoft Visual Studio Code, Lazarus) and offer support for multiple languages. Using more programming languages lead us to using more development tools, probably with some redundancy and overhead. When working on this paper, following development environments were used – Visual Studio 2022 Community edition to work with C/C++ and C#; Visual Studio Code to work with JavaScript, Python and Java, Intellij IDEA to work with Java and Kotlin, Delphi (commercial) for Delphi and Lazarus for Free Pascal. In order to simplify installation of certain extensions and tools, package manager for Visual Studio, *vcpkg* [14] was used. Visual Studio Code was extended with several standard add-ons that allowed easier setup for using VS.Code for Python and Java.

3.2 Time measurement

All languages offer certain functions to get current time. Some of the languages offer more specialized functions. The simplest approach, supported in most languages, was to get time (in number of milliseconds) past from certain fixed timestamp – often, Januray 1st, 1970, as historically used in most languages - prior starting calculations and repeat the same thing after the calculations finish. Difference between two obtained values represent number of milliseconds (or other timer accuracy unit, depending on precision of integrated functions) required to finish the calculations. Of course, there may be internal differences in implementation of the functions, but that is something we did not analyzed. List of functions used, for each language, is given in table 3. Exact examples of usage are completely given in source codes in next subsection.

Table 3. Time measurement functions used in performance test programs

Language	***Functions***
C/C++	`std::chrono::high_resolution_clock::now() + std::chrono:duration`
C#	`System.Diagnostics + StopWatch()`
Delphi	`getTickCount()`
Java	`System.currentTimeMillis()`
JavaScript	`performance.now()`
Kotlin	`System.currentTimeMillis()`
Pascal	`getTickCount()`
Python	`import time + time.time()`
Scilab	`tic() toc()`

3.3 Source code

This subsection includes source-code of our test programs in all analyzed languages. By that, anyone can reimplement the same and check the performance on its own computer.

The source code is given without any additional explanation.

```cpp
1  #include <iostream>
2  #include <chrono>
3  #include "mpir.h"
4  using namespace std;
5  using namespace std::chrono;
6
7  void main()
8  {
9    int Nrep = 50, Nfrom = 1, Nto = 1000;
10   mpz_t tmpFct; mpz_init(tmpFct);
11   mpz_t tmpN;   mpz_init(tmpN);
12
13   auto t1 = high_resolution_clock::now();
14
15   for (int t = 1; t <= Nrep; t++)
16   {  for (int N = Nfrom; N <= Nto; N++)
17      {   mpz_set_d(tmpFct, 1);
18          for (int i = 1; i <= N; i++)
19          {  mpz_set_d(tmpN, i);
20             mpz_mul(tmpFct, tmpFct, tmpN);
21          }
22      }
23   }
24   auto t2 = high_resolution_clock::now();
25
26   auto delta = t2 - t1;
27   duration<long long, nano> duration(delta);
28   double deltaMS = delta.count() / 1000000;
29   double avgT = deltaMS / Nrep;
30   cout << "Avg. = " << avgT << "ms" << endl;
31 }
```

C/C++ with MPIR library

```csharp
using System.Numerics;
using System.Diagnostics;





int Nrep = 50, Nfrom = 1, Nto = 1000;
BigInteger tmpFct = new BigInteger(1);

Stopwatch duration = new Stopwatch();

duration.Start();

for (int t = 1; t <= Nrep; t++)
{  for (int N = Nfrom; N <= Nto; N++)
   {  tmpFct = (BigInteger)1;
      for (int i = 1; i <= N; i++)
      {
         tmpFct = tmpFct * i;
      }
   }
}
duration.Stop();



long deltaMS = duration.ElapsedMilliseconds;
float avgT = (float)deltaMS / Nrep;
Console.WriteLine("Avg.duration =" + avgT + "ms");
```

C#

Figure 1. C/C++ and C# source code

```javascript
1
2
3  Nrep = 50;
4  Nfrom = 1; Nto = 1000;
5
6  t1 = performance.now();
7
8  for (t = 1; t <= Nrep; t++)
9  {  for (N = Nfrom; N <= Nto; N++)
10    {  tmpFct = BigInt(1);
11       for (i = 1; i <= N; i++)
12       {  tmpFct = tmpFct * BigInt(i);
13       }
14    }
15 }
16 t2 = performance.now();
17 avgT = (t2-t1) / Nrep;
18 console.log("Avg.duration = " + avgT);
```

JavaScript

```python
import math
import time
Nrep = 50
Nfrom = 1
Nto = 1000
t1 = time.time()

for t in range(1,Nrep):
   for N in range(Nfrom,Nto+1):
      tmpFct=1
      for i in range(1,N+1):
         tmpFct=tmpFct*i



t2 = time.time()
avgT = (t2-t1)*1000/Nrep
print("Avg.duration = " + str(avgT))
```

Python

Figure 2. JavaScript and Python source code

Please note that the factorial calculation is done inside for-loop with counter variable *i* – given source does not include printouts of the factorial results (if needed, it should be done inside *N*-for-loop, after *i*-for-loop). Also, note that the code is somehow “compressed”, meaning it is not completely visually well-formed as we use it when coding – some empty rows were skipped, some add, curly brackets and blocks brought closer, some code rows were wrapped etc. Also, because figures include two sources each next to another, we intentionally tried to sync the lines by functionality. But, overall, the code is correct and can be reused – in languages with no native big integers support, the environment has to be setup properly in order to successfully execute the programs.

For anyone with experience in programming with general-purpose languages, all the sources should be easily analyzed and understood. The same variable names were used in all languages (not using any particularly naming scheme). Shortly – there were three loops in the code – the first loop with counter *t* is used to repeat calculation *Nrep* times, the second with counter *N* is used to pass through all numbers for which factorial is calculated (from *Nfrom* to *Nto*), and the third loop is used to implement the simplest possible (but unoptimized) calculation of the factorial, by multiplying all the values from 1 to *N*, using counter *i*. The variable *tmpFct* is used to calculate factorials inside the third loop (results can be printed for each required *N* at the end of the second loop).

Java

```
 1 import java.math.BigInteger;
 2 public class FctJava2022 {
 3  public static void main(String[] args) {
 4    int Nrep = 50, Nfrom = 1, Nto = 1000;
 5
 6    BigInteger tmpFct;
 7    BigInteger tmpI;
 8    long t1 = System.currentTimeMillis();
 9
10    for (int t = 1; t <= Nrep; t++)
11    {  for (int N = Nfrom; N <= Nto; N++)
12       {  tmpFct = BigInteger.valueOf(1);
13          for (int i = 1; i <= N; i++)
14          {   tmpI = BigInteger.valueof(i));
15              tmpFct = tmpFct.multiply(tmpI);
16          }
17        }
18    }
19    long t2 = System.currentTimeMillis();
20    float avgT = (t2-t1) / (float) Nrep;
21    System.out.println("Avg. = " +
22                Float.toString(avgT));
23  }
24 }
```

Kotlin

```
import java.math.BigInteger

fun main(args: Array<String>) {
  val Nrep = 50
  val Nfrom = 1; val Nto = 1000
  var tmpFct: BigInteger
  var tmpI: BigInteger
  val t1 = System.currentTimeMillis()

  for (t in 1..Nrep)
  {  for (N in Nfrom..Nto)
     {  tmpFct = BigInteger.valueOf(1)
        for (i in 1..N)
        {  tmpI = BigInteger.valueOf(i.toLong())
           tmpFct = tmpFct.multiply(tmpI)
        }
     }
  }
  val t2 = System.currentTimeMillis()
  val avgT = (t2 – t1).toFloat() / Nrep
  println(("Avg.duration = " + avgT.toString())


}
```

Figure 3. Java and Kotlin source code

Delphi + Velthuis.BigIntegers

```
 1 program fctDelphi2022;
 2 {$APPTYPE CONSOLE}
 3 uses System.SysUtils, Windows,
 4      Velthuis.BigIntegers;
 5
 6 var Nrep, Nfrom, Nto, t, N, i:integer;
 7     t1, t2: DWORD;
 8     tmpFct: BigInteger;
 9 begin
10   Nrep := 50; Nfrom := 1; Nto := 1000;
11   t1 := getTickCount();
12
13   for t:=1 to Nrep do
14   begin
15     for N:=Nfrom to Nto do
16     begin
17       tmpFct := BigInteger(1);
18       for i:= 1 to N do
19       begin
20         tmpFct := tmpFct * i;
21       end;
22     end;
23   end;
24   t2 := getTickCount();
25   avgT := (t2-t1)/Nrep;
26   writeln('Avg.dur.='+floattostr(avgT)+'ms');
27 end.
```

FreePascal / Lazarus + TForge extension

```
program fctPascal2022x;
uses tfNumerics, Windows, Sysutils;



var Nrep, Nfrom, Nto, t, N, i:integer;
    t1, t2: DWORD;
    tmpFct: BigInteger;
begin
  Nrep := 50; Nfrom := 1; Nto := 1000;
  t1 := getTickCount();

  for t:=1 to Nrep do
  begin
    for N:=Nfrom to Nto do
    begin
      tmpFct := BigInteger(1);
      for i:= 1 to N do
      begin
        tmpFct := tmpFct * i;
      end;
    end;
  end;
  t2 := getTickCount();
  avgT := (t2-t1)/Nrep;
  writeln('Avg.duration='+floattostr(avgT)+'ms');
end.
```

Figure 4. Delphi and FreePascal/Lazarus source code

3.4 The fastest program – C/C++ with MPIR and optimized factorial function

The fastest calculation and overall test results were obtained using the specific, highly optimized factorial calculation function of MPIR library [7] used with C/C++ compiler. The same library was used for standard unoptimized, loop-multiplication based calculation and also performed quite well (details on performance test results are given in next section). The code using MPIR is depicted in Fig. 1, left – special big integer type is using keyword `mpz_t` and, according to official library documentation, specific functions (`mpz_init`, `mpz_set`, `mpz_mul`) have to be used to work with that data type.

Among others, the function `mpz_fac_ui`, implemented as part of the library, can be used to calculate the factorial of given number. Based on obtained results and available documentation, we can only conclude that it is highly optimized and gives the best results in our tests. The way to use it is quite simple – in order to get the fastest results, the code in Fig. 1 left should be changed – lines from 18 to 21 should be commented or deleted and replaced with simple call to the specialized function – `mpz_fac_ui(tmpFct, N);` - the first parameter is resulting variable and the second parameter is the number for which the factorial has to be calculated!

4 RESULTS

All the measurements were made on standard Windows based PC (Windows 10 Pro, 64-bit) with Intel i3-6100U CPU, 16GB RAM, NVM.e SSD disk drive. The results shown in table 4 were obtained by executing the programs given in source code subsection in 3.3. Results from the table are visualized on diagrams on Fig. 5 and Fig. 6.

Table 4. Performance test results – average duration of execution in milliseconds – N_{rep} = 50

Language / environment	***Nf_{rom} = 1, N_{to} = 1000***	***N_{from} = 1, N_{to} = 250***
C/C++ 32bit + MPIR	237.46	5.02
C/C++ 64bit + MPIR	96.18	2.53
C/C++ 32bit + MPIR fact	33.86	1.00
C/C++ 64bit + MPIR fact	9.28	0.53
C#	96.64	2.52
Delphi 64 bit	144.20	7.37
Delphi 32 bit	213.38	8.81
Java *(VS.Code)*	87.92	5.51
Java *(IntelliJ IDEA)*	98.73	7.81
JavaScript *(Edge)*	149.80	7.84
JavaScript *(Chrome)*	142.02	6.69
JavaScript *(Firefox)*	735.06	14.68
JavaScript *(Opera)*	145.00	7.64
Kotlin	97.37	7.25
Pascal + TForge	409.66	13.12
Python	288.38	11.08

For certain native development platforms (Visual C++, Delphi) both 32- and 64-bit executables were built and tested. It can be seen that both C/C++ and Delphi perform significantly better when using 64-bit compiler.

JavaScript was tested in four web browsers – the results look quite strange for Mozilla Firefox browser that seems to have much lower performance, when compared with others, especially in test with *Nto* = 1000, where it was almost 5 times slower than other browsers (nicely depicted in Fig. 5)!?

One, for authors little unexpected, conclusion may be that interpreted and dynamically typed languages such as JavaScript and Python offer the performances comparable to other, even natively compiled environments. The similar could be said for C#, Java and Kotlin – being intermediately compiled, they offer great performance – it must be because well designed and optimized JVM and .NET runtime. For us, standard old-school programmers, it was expected that native compiled C/C++ od Delphi (but,

unfortunately, without native big integer support) may provide the fastest calculations, but we were slightly disappointed seeing other, more modern, languages have similar or even better results. Luckily, MPIR library used with C/C++ includes optimized factorial calculation function that beats all other solutions.

Please note that Scilab was not listed in results table. That is due its poor performances – as seen, the slowest results were obtained using JavaScript in Firefox. But, even those results, for example, 735ms for generating all the factorials for range [1, 1000] may be considered great result! Not to mention little less than 15ms for calculating all the factorials in range [1, 250]! But, in Scilab the overall procedure lasted so long that we decided to cancel the execution. Just to give some valuable information – Scilab uses additional *BigInteger* toolbox [13] and can work with big integer values. But, for instance, calculating only one small list of factorials (simplest possible multiplication implementation, as in other languages shown in source examples), for numbers in range [1, 50] it required more than 280 seconds! Or, calculating single factorial, e.g. 100! lasts 180 seconds.

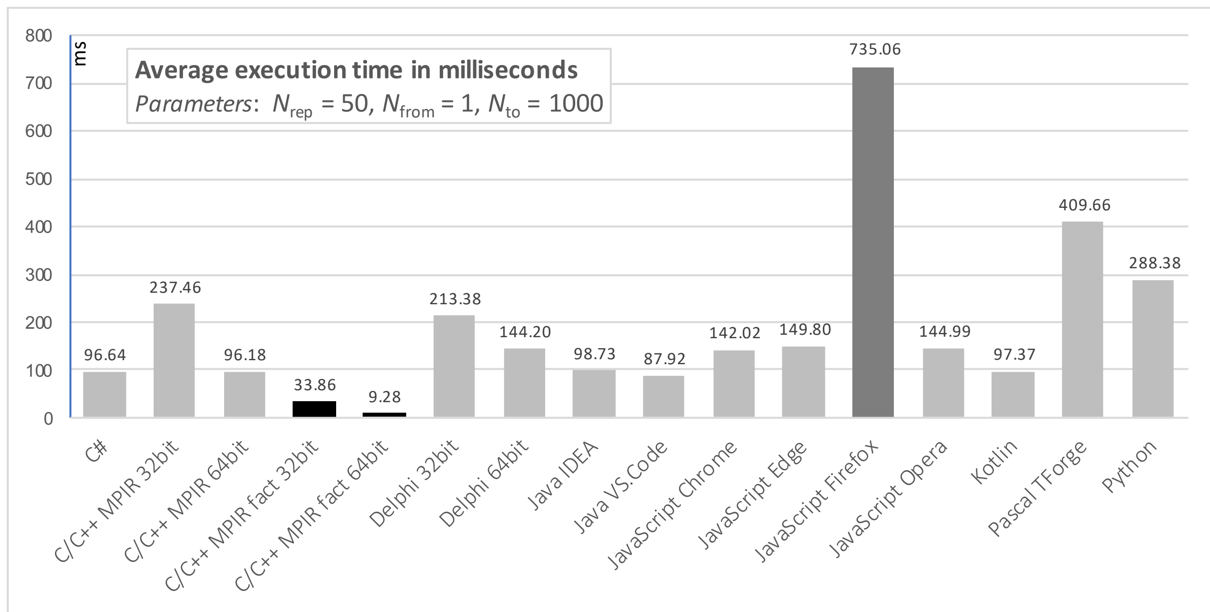

Figure 5. Chart visually presenting results for test with N_{from} = 1, N_{to} = 1000, time in milliseconds

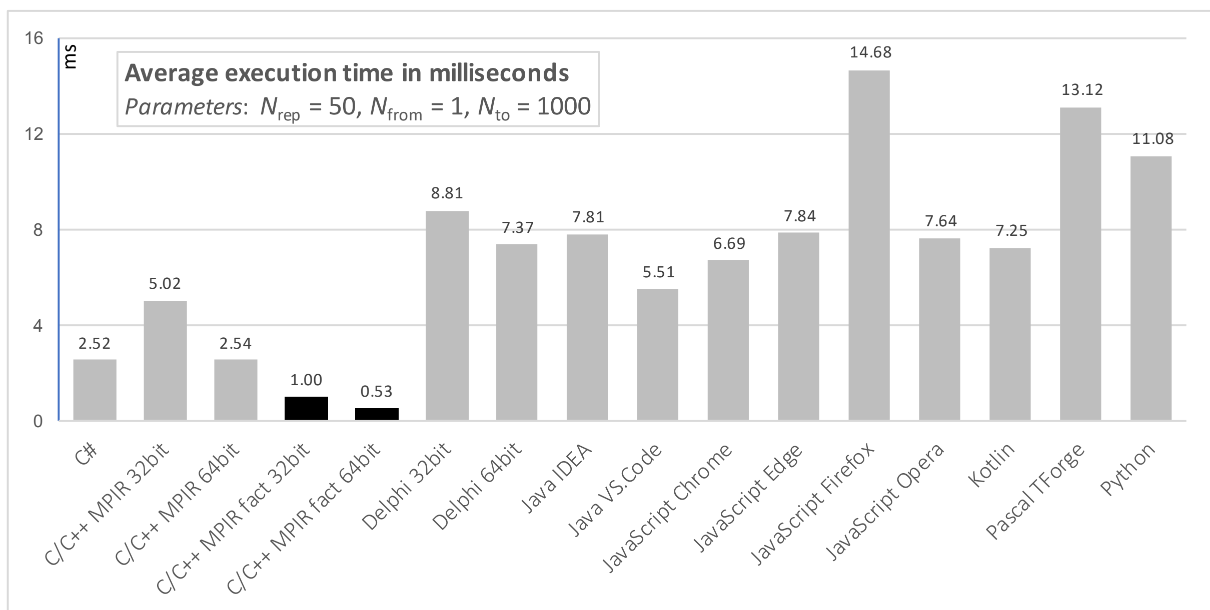

Figure 6. Chart visually presenting results for test with N_{from} = 1, N_{to} = 250, time in milliseconds

After conducting presented performance analysis, we shall express our regret on space limitations for this paper, since some additional comments, explanations or even source code modifications may be discussed. However, that could be marked as possible future work or a kind of extension of this paper.

5 CONCLUSIONS

The main objective of the paper was to analyze the abilities of the arbitrary range integers, so called big integers, in the collection of the today widely known and popular general-purpose programming languages, both the compiled and interpreted ones. We successfully managed to measure the big integer calculation performances of different languages by implementing benchmark function based on the fast-growing factorial function. Based on the obtained results, some objective findings were presented. It was concluded that most modern and lately popular languages such as JavaScript, Python, C# or Java natively support the arbitrary large integer arithmetic and that they all perform quite well, even above our conservative expectations. On the other hand, more classic languages, such as C/C++ and Pascal do not support big integers natively, so that they relay on external libraries. Fortunately, that kind of libraries exists, they are usually freely available and perform very well.

In general, all of the most popular languages today provide well performing big integer calculation abilities. However, there was a slightly disappointment related to the big integer support in specialized mathematical tools such as Scilab and GNU Octave – even though the type support may be available, the performances we obtained were so poor that we were not even willing to compare it with the results of general-purpose languages.

REFERENCES

[1] M. Mikac, M. Horvatić, “On explaining variable range in standard programming languages (to STEM students)”, in *ICERI2019 Proceedings*, IATED Academy, pp. 1551-1561, 2019.

[2] TIOBE Index – April 2022. Accessed 06. May 2022. Retrieved from https://www.tiobe.com/tiobe-index

[3] PLPY Popularity of Programming Language. Accessed 06. May 2022. Retrieved from https://pypl.github.io/PYPL.html

[4] M. Mikac, M. Horvatić, “Using open-source numerical computation software in education – basic performance comparison and lab examples”, in *EDULEARN20 Proceedings*, IATED Academy, pp. 2319-2327, 2020.

[5] M. Mikac, M. Horvatić, V. Mikac, “Using vectorized calculations in Scilab to improve performances of interpreted environment”, in *INTED2020 Proceedings*, IATED Academy, pp. 2127-2136, 2020.

[6] Kotlin Native. Accessed 08. May 2022. Retrieved from https://kotlinlang.org/docs/native-overview.html

[7] B. Gladman, W. Hart, J. Moxham, et al. *MPIR: Multiple Precision Integers and Rationals*, 2015. version 2.7.0, A fork of the GNU MP package (T. Granlund et al.). Accessed 10. May 2022. Retrieved from https://mpir.org

[8] R. Velthuis. Delphi Big Numbers. Accessed 10. May 2022. Retrieved from https://github.com/rvelthuis/DelphiBigNumbers

[9] TForge for Lazarus. Accessed 10. May 2022. Retrieved from https://sergworks.wordpress.com/2016/04/10/tforge-0-75/

[10] TForge download. Accessed 10. May 2022. Retrieved from https://torry.net/files/vcl/security/strong/tforge073.zip

[11] M. Martin. FNX – Multiprecission numbers library for FreePascal on Linux. Accessed 10. May 2022. Retrieved from http://www.ellipsa.eu/public/fnx/fnx.html

[12] GMP for FreePascal. Accessed 10. May 2022. Retrieved from https://wiki.freepascal.org/gmp

[13] D. Clement, P. Roux. BigInteger toolbox for Scilab. Accessed 10. May 2022. Retrieved from https://forge.scilab.org/index.php/p/bigint/

[14] VCPKG – package manager. Accessed 10. May 2022. Retrieved from https://vcpkg.io