SoftwareX 23 (2023) 101428

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

JGMP: Java bindings and wrappers for the GMP library

Gianluca Amato, Francesca Scozzari *

Laboratory of Computational Logic and AI Department of Economic Studies, University of Chieti–Pescara, Pescara, Italy

ARTICLE INFO

Article history:
Received 6 April 2023
Received in revised form 24 May 2023
Accepted 31 May 2023

Keywords:
Arbitrary precision arithmetic
Integer numbers
Rational numbers
Floating point numbers
Java

ABSTRACT

The GNU Multiple Precision Arithmetic Library (GMP) is a widely used library for computing with arbitrary precision arithmetic. The library has functionally complete bindings for many programming languages, including .NET, C++, OCaml, Python, Ruby, and Rust, with the notable exception of Java. The JGMP library provides Java bindings and wrappers for using GMP from within any JVM-based language. The library has been thoroughly tested and benchmarked.

Code metadata

Current code version	1.0.1
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-23-00223
Code Ocean compute capsule	
Legal Code License	GNU LGPL v3.0
Code versioning system used	git
Software code languages, tools, and services used	Java, Maven
Compilation requirements, operating environments & dependencies	JDK (≥11), Maven (≥3.2.5), GMP C library (≥6.1.0)
If available Link to developer documentation/manual	https://javadoc.io/doc/it.unich.jgmp/jgmp/latest/index.html
Support email for questions	gianluca.amato@unich.it

1. Motivation and significance

The GMP (GNU Multiple Precision Arithmetic Library) [1] is a library for computing with arbitrary precision arithmetic (see for instance [2,3]). It is widely used in many application areas, such as security, cryptography, computational algebra systems (such as Mathematica [4] and Maple [5]), financial and approximation algorithms, static analysis (e.g., [6–11]), in the building of the GNU Compiler Collection (GCC). Parts of its algorithms have been proven correct using the Coq proof system [12]. There are currently bindings for using GMP from many languages besides C and C++, such as .NET, OCaml, Perl, PHP, Python, R, Ruby, and Rust. Surprisingly, there are no well established and functionally complete bindings for Java or other languages based on Java bytecode.

One of the reasons for this state of affair is that Java has a built-in implementation of arbitrary precision integer numbers in the `BigInteger` class, and that the cost of calling native code from Java is quite high. Therefore, interfacing with GMP has never been considered worth the effort. However, we beg to dissent from this argument, for several reasons:

1. The GMP library not only implements arbitrary precision integers, but also arbitrary precision rational and floating point numbers, for which there is no alternative in the standard Java library (`BigDecimal` implements fixed point numbers, not floating point ones).
2. Even if we want to restrict our interest to integers, the GMP library implements many operations and algorithms which have no counterpart in the standard Java library, such as many number-theoretic functions.
3. Although in Java the cost of calling native code is high, when the numbers involved are big the performance of GMP overcomes this drawback (we will show some benchmarks in Section 4.1);
4. Sometimes you want to use from Java a native library which depends on GMP. In this case, you need GMP wrappers to be able to interface with the native library. This

* Corresponding author.
E-mail addresses: gianluca.amato@unich.it (Gianluca Amato), francesca.scozzari@unich.it (Francesca Scozzari).

https://doi.org/10.1016/j.softx.2023.101428

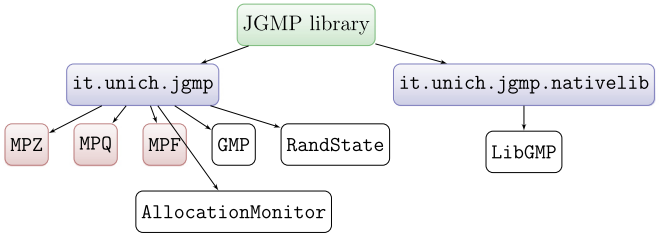

Fig. 1. The structure of the packages and classes in the JGMP library.

is the case, for example, of the PPL (Parma Polyhedra Library) [13], a well-known library for polyhedral computation. The authors of the PPL, in their Java bindings, also had to wrap some functionalities of the GMP. The fact that each library which needs GMP has to develop its own wrappers brings a fragmentation of incompatible and partial solutions.

In this paper we present the JGMP library, which provides complete bindings and wrappers for all the GMP data types (integer, rational and floating point numbers) for Java and other JVM-based languages. JGMP aims to become the reference implementation for accessing GMP within Java, by replacing and extending all the currently available solutions.

2. Software description

2.1. Software architecture

The software structure of the JGMP library is shown in Fig. 1. The library consists of two packages:

- `it.unich.jgmp`: contains all the classes wrapping multi-precision integer, rational and floating point numbers. This is the package which developers will generally use.
- `it.unich.jgmp.nativelib`: contains all the low-level classes of JGMP. In particular, all the code interfacing with the native C library is part of this package. Developers may generally ignore this package, although in the current version of JGMP there is a very limited set of low level functionalities which are only available here.

The main classes in the package `it.unich.jgmp` are:

- `MPZ`: for multi-precision integer numbers;
- `MPQ`: for multi-precision rational numbers;
- `MPF`: for multi-precision floating point numbers.

There are three additional classes: `GMP` which collects global variables and static methods, `AllocationMonitor` which keeps track of the amount of native memory allocated by GMP and calls the Java garbage collector, and `RandState` which encapsulates the current state of random number generators.

The package `it.unich.jgmp.nativelib` collects all the code directly interfacing with the native C library. The most important class is `LibGMP`, which contains the static native methods corresponding to GMP functions, the constants for the size of native GMP structures and some global variables. Some documented GMP functions are actually macros, and they have been reimplemented here. Other classes in this package are mostly Java proxies for the parameter and return types used by the native methods.

For interfacing with native code we use the Java Native Access (JNA) library. Although somewhat slower w.r.t. the standard approach based on Java Native Interface (JNI) [14,15], JNA allows accessing native code entirely within Java, without having to write a single line of code in C. This makes maintenance and distribution of binary code quite easier, since we do not have to deal with different binaries for different operating systems and CPUs. In order to reduce the overhead of calling C functions, we use direct mapping almost everywhere, and we treat GMP objects as black boxes which we keep in the native memory and never bring on the Java heap.

In the future, we plan to test alternatives to JNA such as the JNR-FFI library, and the Foreign Linker API of the newest Java versions.

2.2. Software functionalities

JGMP provides Java bindings and wrappers for accessing the GMP library from Java and other languages running on the JVM platform. The main classes `MPF`, `MPQ` and `MPZ` provide wrappers for multi-precision floating point, rational and integer numbers. Almost all GMP functions are exposed by wrappers, expect for a few low-level functions (such as those directly dealing with *limbs*). These functions may be used in C for implementing custom operations, but we deemed them to be scarcely useful in Java.

The names and signatures in the classes `MPF`, `MPQ` and `MPZ` have been carefully chosen to adhere to Java naming conventions, while keeping methods discoverable by the developers who already know the GMP library. In order to simplify the development of code without side effects, we have enriched the API provided by JGMP with side-effect-free methods, which builds new objects instead of modifying old ones. Finally, JGMP wrappers have been thoroughly tested by an extensive test suite.

2.2.1. The immutable API

The GMP library, differently from most of the Java arbitrary precision arithmetic libraries, implements a mutable API. This is reflected in JGMP. For example, the code fragment

```
MPZ x = new MPZ(0)
x.addAssign(1)
```

creates a multi-precision integer number `x` initialized with 0, and later changes its value to 1. However, JGMP also implements an immutable API, which may be preferable for Java programmers. For example:

```
MPZ x = new MPZ(0)
MPZ y = x.add(1)
```

creates a multi-precision integer number `x` initialized with 0, then adds 1 to `x` returning a different object, which is assigned to the variable `y`.

The immutable operations are generally slower than the mutable ones. Moreover, there is potentially a problem of uncontrolled memory allocations. The problem is that, when an `MPZ` object is created, a corresponding amount of native memory (outside the Java heap) is allocated by GMP. When the `MPZ` object is reclaimed by the garbage collector, the native memory is correctly deallocated by the cleaner thread implemented by JGMP. However, the size of the allocated native memory is not used by the JVM to decide when it is the moment to call the garbage collector. When we create many big JGMP objects, we can allocate a huge amount of native memory, although the size of the same objects in the Java heap is negligible. As a result, the Java garbage collector is not invoked, and GMP exhausts all the native memory, leading to the program being killed by the operating system.

In order to overcome this problem, JGMP implements an experimental feature which, using appropriate hooks provided by GMP, keeps track of all the native memory allocated by the library. When the amount of allocated memory is too big, JGMP

manually invoke the Java garbage collector. Unfortunately, this means that any memory allocation in the GMP library invokes a Java method, which degrades performance. For this reason, this feature is normally disabled and may be enabled manually by calling `AllocationMonitor.enable()`.

For invoking the garbage collector, we call the `System.gc()` method. According to the Java API documentation "Calling the gc method suggests that the Java Virtual Machine expend effort toward recycling unused objects", so execution of the garbage collector is only suggested, not forced. Nonetheless, it seems to work consistently, and this is the method that the Java library itself uses when it wants to free native memory (see the source code of the `reserveMemory` method in the non-public `java.nio.Bits` class).

2.2.2. Thread safety

Thread safety of JGMP depends on thread safety of the native GMP library. In general, all operations are thread safe, provided there is no attempt to write to the same GMP object from multiple threads. There are some exceptions:

- methods `randomAssign`, `random`, `random2Assign`, `random2` of the `MPZ` class are not thread safe, and are marked as deprecated;
- the memory tracker in the `AllocationMonitor` class is enabled or disabled globally, for all the threads of the program;
- the default precision for floating point numbers (which may be set by using `MPF.setDefaultPrec` method) is global for all the threads.

3. Illustrative examples

We show a full, simple example of a program for computing the factorial of a number.

```
import it.unich.jgmp.*;

public class FactorialExample {

  public static void main(String[] args) {
    int x = 100000;
    MPZ factorial = factorialMPZ(x);
    System.out.println(factorial);
  }

  public static MPZ factorialMPZ(int x) {
    MPZ f = new MPZ(1);
    while (x >= 1) {
      f.mulAssign(x);
      x -= 1;
    }
    return f;
  }
}
```

The assignment `MPZ f = new MPZ(1)` creates a new MPZ object initialized to 1, while `f.mulAssign(f, x)` computes `f*x` and assigns the result to `f`.

The following method computes the factorial using the method `mul` from the immutable API, which returns `f*x` without side effects.

```
public static MPZ immutableFactorialMPZ(int x) {
 MPZ f = new MPZ(1);
 while (x >= 1) {
  f = f.mul(x);
  x -= 1;
 }
 return f;
}
```

In the GitHub project jandom-devel/JGMPBenchmarks it is possible to find the complete source code, with further examples and benchmarks.

4. Impact

While the number of projects using GMP is extremely large, the number of Java projects which exploit the GMP library is limited. There are several projects where the authors had to implement a part of the JGMP library (usually exposing only those GMP functions needed for the projects itself), but in most cases they do not come with tests and benchmarks, and it is very difficult to reuse the implementation in a different project.

Having arbitrary precision operations is important not only for algebraic and cryptographic applications, which are the main target of these libraries, but also in financial and approximation algorithms, included static analyzers of software. For instance the Parma Polyhedra Library, a C++ library widely used in static analysis, provides a Java interface and thus some Java bindings for handling GMP objects from Java.

4.1. Benchmarks

We show in this section that, although calling native code from Java is costly, if the calculation is computationally expensive, then JGMP is nonetheless faster than the alternatives. We have compared JGMP with the following libraries:

- Apfloat 1.10.1 [16], an arbitrary precision arithmetic library implementing integers (`Apint`), rationals (`Aprational`), floating point numbers (`Apfloat`). The library also implements complex numbers (`Apcomplex`), that we do not benchmark here.
- JScience 4.3.1 [17], a comprehensive Java library for the scientific community. The `org.jscience.mathematics.number` package implements arbitrary precision integers (`LargeInteger`), rationals (`Rational`) and floating point numbers (`FloatingPoint`). It also implements arbitrary precision modular integers (`ModuloInteger`) and floating point numbers with known uncertainty (`Real`), that we do not benchmark here.
- Apache Common Numbers 1.1 [18], part of the well-known Apache Commons library, implements arbitrary precision rationals (`BigFraction`).
- MPFR Java Bindings [19], a set of JNI bindings and wrappers for the GNU MPFR library. The GNU MPFR library extends the floating point type of GMP with well-defined rounding properties. Note that these bindings are only guaranteed to work with GNU MPFR version 3 which is obsolete.
- GMP4J [20], an old JNI-based wrapper for GMP that implements arbitrary precision integers. Among the prior work discussed in Section 4.2, we have chosen to benchmark GMP4J since it uses JNI (and therefore offers a way to evaluate the performance benefits of this technology w.r.t. JNA) and is not tailored to a specific application.

In the case of JGMP we benchmark both the mutable and immutable APIs, while for all the other libraries only the immutable API is available. We consider the following four benchmarks:

- `Factorial`: computes the factorial of the numbers $n = 10$, 100, 1 000, 10 000, 100 000.
- `Prime`: computes the next 100 pseudo-primes starting from $n = 10$, 100 and 1 000. We only benchmark the `MPZ`, Java `BigInteger` and GMP4J `BigInteger` classes since `Apint` and `LargeInteger` have no specific support for computing the next pseudo-prime number.

Table 1
Benchmark results. We show the mean execution time in ms of each procedure, together with its 99.9% confidence interval (computed assuming a normal distribution). Benchmark have been executed on an Intel Core i5-2500K clocked at 1.6 GHz with 16 GB of RAM, running Ubuntu Linux 22.04, under OpenJDK 11 64-Bit Server VM. The maximum Java heap size is set to 2 GB.

Factorial

n	10	100	1,000	10,000	100,000
JGMP built-in (MPZ)	0.010 ± 0.002	0.010 ± 0.003	**0.023 ± 0.002**	**0.718 ± 0.001**	**23.416 ± 0.168**
JGMP mutable (MPZ)	0.013 ± 0.002	0.078 ± 0.001	0.888 ± 0.009	26.503 ± 0.072	2485.123 ± 4.904
JGMP immutable (MPZ)	0.107 ± 0.026	1.075 ± 0.213	10.499 ± 2.739	killed	killed
GMP4J (BigInteger)	0.037 ± 0.003	0.370 ± 0.025	4.932 ± 0.203	killed	killed
Java (BigInteger)	$\approx$**10**$^{-3}$	**0.009 ± 0.001**	0.607 ± 0.004	68.328 ± 0.220	8574.014 ± 21.485
Apfloat (Apint)	0.007 ± 0.001	0.080 ± 0.001	1.841 ± 0.028	163.048 ± 5.076	19362.493 ± 442.535
JScience (LargeInteger)	$\approx$**10**$^{-3}$	0.010 ± 0.001	0.953 ± 0.025	121.901 ± 0.567	16497.285 ± 49.511

Prime

n	10	100	1,000
JGMP mutable (MPZ)	**0.417 ± 0.010**	**8.572 ± 0.040**	**6937.242 ± 80.031**
JGMP immutable (MPZ)	1.306 ± 0.406	9.065 ± 0.145	**6993.390 ± 34.124**
GMP4J (BigInteger)	0.614 ± 0.048	8.705 ± 0.052	**6967.942 ± 39.874**
Java (BigInteger)	8.562 ± 0.229	90.019 ± 0.503	8753.752 ± 16.732

PiFraction

steps	1	10	100	1,000
JGMP mutable (MPQ)	0.043 ± 0.009	0.070 ± 0.008	**0.428 ± 0.005**	**5.032 ± 0.045**
JGMP immutable (MPQ)	0.085 ± 0.017	0.439 ± 0.092	4.469 ± 0.875	50.356 ± 11.657
Apfloat (Aprational)	0.022 ± 0.001	1.762 ± 0.041	876.600 ± 9.320	113077.876 ± 1006.002
Apache Commons (BigFraction)	$\approx$**10**$^{-3}$	**0.007 ± 0.001**	2.577 ± 0.030	951.747 ± 1.104
JScience (Rational)	0.002 ± 0.001	0.029 ± 0.001	3.705 ± 0.021	1145.914 ± 33.544

PiFloat

steps	1	10	100	1,000
JGMP mutable (MPF)	0.050 ± 0.018	**0.076 ± 0.016**	**0.480 ± 0.005**	**4.614 ± 0.038**
JGMP immutable (MPF)	0.071 ± 0.013	0.366 ± 0.077	3.338 ± 0.800	31.651 ± 5.556
Java (BigDecimal)	0.005 ± 0.001	0.153 ± 0.001	1.590 ± 0.007	15.259 ± 0.109
Apfloat (Apfloat radix 10)	**0.003 ± 0.001**	0.564 ± 0.006	6.234 ± 0.086	63.531 ± 0.891
Apfloat (Apfloat radix 2)	**0.003 ± 0.001**	0.525 ± 0.009	5.919 ± 0.092	59.682 ± 0.612
JScience (FloatingPoint)	0.007 ± 0.001	0.301 ± 0.004	3.219 ± 0.192	31.595 ± 0.288
MPFR Java Bindings (BigFloat)	0.032 ± 0.003	0.200 ± 0.024	1.853 ± 0.271	18.239 ± 2.864

- `PiFraction`: computes an approximation of π as a rational number using the generalized continued fraction

$$\pi = 3 + \cfrac{1^2}{6 + \cfrac{3^2}{6 + \cfrac{5^2}{\cdots}}}$$

  Different expansions were tried, with number of approximation steps equal to 1, 10, 100 and 1000.
- `PiFloat`: implements the same computation as described above but using floating point numbers with 1024 bits of precision in the mantissa instead of rationals (actually, `BigDecimal` uses fixed point numbers, but we decided to benchmark it together with the others). Note that `MPF` and `MPFR` use a binary representation, `BigDecimal` and `FloatingPoint` use a decimal representation and `Apfloat` may use both. For classes with a decimal representation, precision has been set to $\lfloor 1024 \cdot \log_{10} 2 \rfloor = 308$ digits.

Benchmarking programs running on the JVM is not an easy task, since a lot of factors may impact the execution speed, such as just-in-time compilation and garbage collection. We have used the JMH (Java Microbenchmark Harness) to perform the benchmarks, using 5 forks, each fork composed of 5 iterations for warming up the JVM and 5 iterations for collecting the results. On top of this, we have tried to reduce the effect of automatic CPU performance scaling by disabling Turbo Boost and setting a fixed clock for the CPU, low enough to avoid overheating the processor. In particular, the results have been obtained on an Intel Core-i5 2500K clocked at 1.6 GHz with 16 GB of RAM. The source code for all the benchmarks may be found in the jandom-devel/JGMPBenchmarks GitHub repository.

The results of the benchmarks are shown in Table 1 and depicted in Fig. 2. They show that, expect a few easy cases, JGMP with the mutable API is faster than all the other libraries. This is particularly true when the GMP library provides a specific built-in function, like for instance the factorial. When using the immutable API, instead, the performance may reach and surpass that of the other libraries only for very costly computations. Nonetheless, the immutable API may be useful when performance is not a problem, since it is easier to use and more common among Java programmers. Note that GMP4J is consistently faster than the immutable API of JGMP due to the use of JNI instead of JNA. The gap is larger for small computations, where the overhead of calling native code is higher.

Note that, for the *Factorial* benchmark, the immutable API shows the phenomenon discussed in Section 2.2.1, which causes the exhaustion of native memory. For this benchmark, we show in Table 2 the results when the allocation monitor is enabled. In this case, all benchmarks may be performed without crashing. The downside is that enabling the allocation monitor causes a drop in performance, not only in the immutable case, but also for the mutable and built-in cases, where it brings limited benefit due to the small amount of native memory consumed.

The problem of native memory exhaustion may also be faced by manually placing calls to `System.gc()` within the user code. A careful positioning of these calls may achieve better performance than what it is possible with an automatic mechanism like our allocation monitor, at the expense of more work for the user of the library. As an example, in Tables 2 it is possible to find the results of the factorial benchmarks when calling `System.gc()` before each computation of the factorial. With this particular choice, the results are worse than what the allocation monitor achieves for smaller values of n (when the overhead of calling

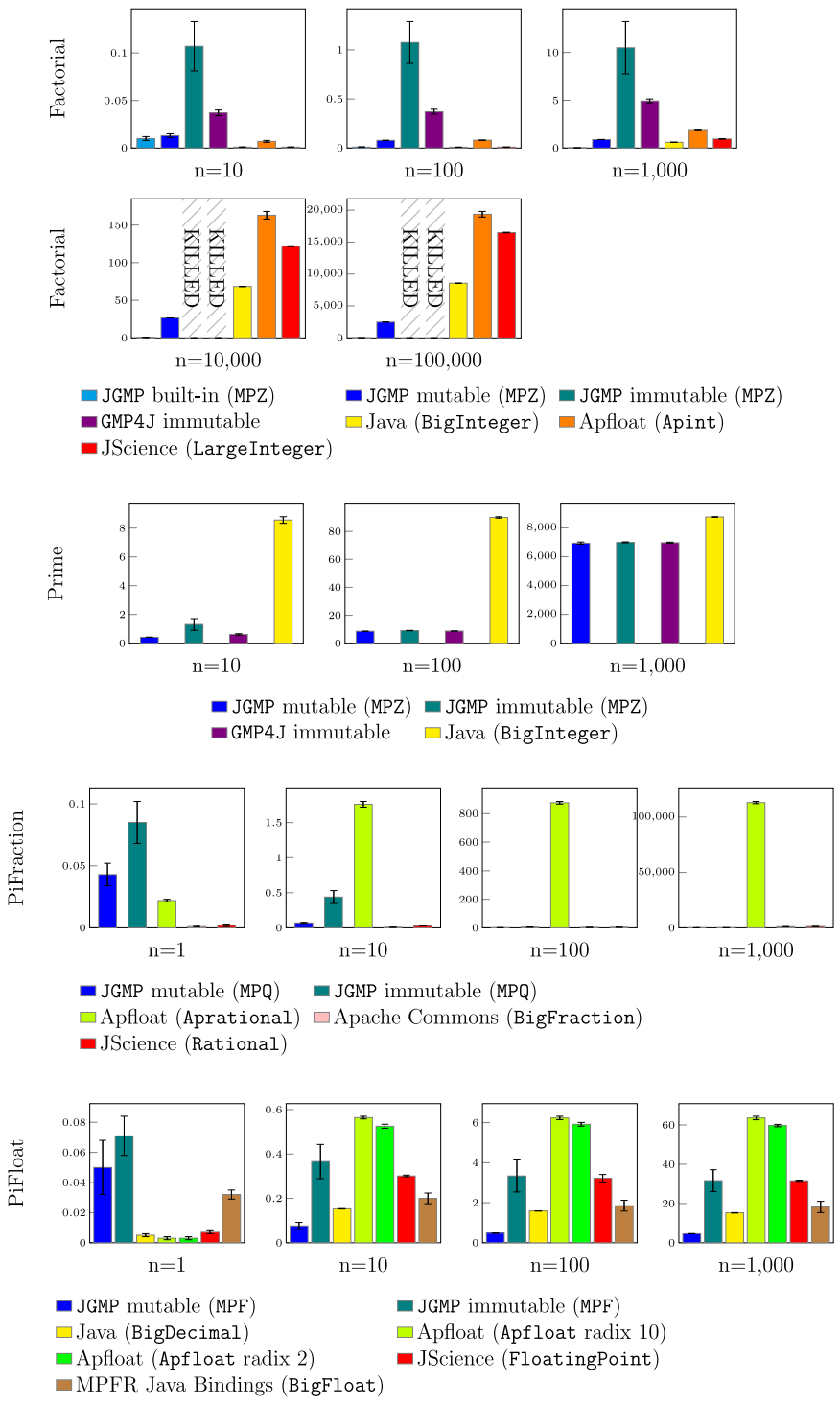

Fig. 2. Graphs for the benchmarks in Table 1.

the garbage collector before each computation is quite high), but are better with large values of n.

Finally, in Table 3 we show some results concerning the behavior of the garbage collector for the same benchmarks in Table 2. If neither the allocation monitor nor the manual `System.gc()` calls are in use, the number of garbage collector events is quite small. For a big n, it is actually too small, leading to memory exhaustion. If we manually call `System.gc()`, the number of events is generally quite high, with the single exception of $n = 100\,000$ with the immutable API. This explains the low performance of this approach for small values of n. The results with the allocation monitor are a good trade-off between the other approaches. However, since the allocation monitor pays the cost of an additional overhead for each memory allocation, the execution time may be higher than the manual method even if the garbage collector is invoked less frequently.

Obviously, instead of calling `System.gc()` before each computation, with the help of the results in Table 3, it is possible to design a smarter policy which reduces the number of garbage collector events and leads to a better performance.

Table 2
Benchmark results (in ms) comparing JGMP on the *Factorial* example with different policies for the garbage collector: without any explicit call (*no*), using the allocation monitor (*monitor*) and with manual call before each computation (*manual*). Benchmark conditions, as well as the results for JGMP without allocation monitor, are the same as in Table 1. The allocation monitor is configured to keep the allocated native memory size under 2 GB and the garbage collector used is G1.

n	gc calls	built-in	mutable	immutable
10	no	0.010 ± 0.002	0.013 ± 0.002	0.107 ± 0.026
	monitor	0.014 ± 0.002	0.030 ± 0.004	0.175 ± 0.029
	manual	9.636 ± 0.207	10.146 ± 1.380	9.351 ± 0.123
100	no	0.010 ± 0.003	0.078 ± 0.001	1.075 ± 0.213
	monitor	0.026 ± 0.003	0.167 ± 0.004	1.684 ± 0.291
	manual	9.463 ± 0.120	9.896 ± 0.957	9.486 ± 0.145
1,000	no	0.023 ± 0.002	0.888 ± 0.009	10.499 ± 2.739
	monitor	0.039 ± 0.001	2.084 ± 0.022	16.008 ± 2.130
	manual	9.599 ± 0.152	9.880 ± 0.133	13.400 ± 0.388
10,000	no	0.718 ± 0.001	26.503 ± 0.072	killed
	monitor	0.769 ± 0.003	43.504 ± 0.245	181.680 ± 6.547
	manual	10.385 ± 0.156	35.321 ± 0.305	82.890 ± 0.740
100,000	no	23.416 ± 0.168	2485.123 ± 4.904	killed
	monitor	24.086 ± 0.081	2724.031 ± 14.217	4231.681 ± 37.543
	manual	33.388 ± 0.945	2466.160 ± 18.080	3302.939 ± 51.672

Table 3
Benchmark results comparing JGMP on the *Factorial* example with different policies for the garbage collector as in Table 1. The *count* column reports the total number of calls, and the *time* column the time spent in the garbage collector in ms.

n	gc calls	built-in		mutable		immutable	
		count	time	count	time	count	time
10	no	132	130491	146	42370	129	127061
	monitor	175	48654	119	19167	182	46230
	manual	21987	242868	21813	242211	21529	241032
100	no	135	134024	49	6319	130	125125
	monitor	153	35446	22	2112	187	43919
	manual	21958	242358	21141	240931	21155	239207
1,000	no	57	16646	12	415	138	104802
	monitor	80	11592	12	300	178	52693
	manual	21923	241210	20511	226433	15970	201453
10,000	no	0	–	15	162	killed	
	monitor	15	684	20	215	81	27932
	manual	20579	228356	6726	74206	2986	78069
100,000	no	0	–	1	10	killed	
	monitor	1	13	8	76	350	17044
	manual	7266	80638	125	1356	96	18998

4.2. Related works

There are several GitHub projects providing GMP wrappers for Java (see Table 4), but they only implement arbitrary precision integers. Moreover, most of them are inactive projects.

The `gmp-java` [21] project is a script which, given the source code of the GNU Compiler for Java (`GCJ`) (version 4.6.1), extracts the implementation of the `BigInteger` class, which is based on the GMP library, and compile it under a new name, to avoid name clashes. Therefore, the output library has exactly the same interface as the standard `BigInteger` class. The `GMP-java` [22] library provides wrappers for the `mpz` type, but for a limited selection of operations which does not include, for example, any root or number theoretic function. The `GMP4J` [20] is also a wrapper for the `mpz` type, but with more features than the previous ones. It implements most of the GMP operations, expert for some number theoretical ones. It is designed to be a drop-in replacement for the standard `BigInteger` class.

Table 4
GMP Bindings for Java.

URL	last commit	native interface
https://github.com/infinity0/gmp-java [21]	12 years ago	JNI
https://github.com/dfdeshom/GMP-java [22]	10 years ago	JNI
https://github.com/altmind/GMP4J [20]	10 years ago	JNI
https://github.com/mathybit/java-gmp [23]	5 years ago	JNA
https://github.com/square/jna-gmp [24]	2 years ago	JNA

Both `java-gmp` [23] and `jna-gmp` [24] have been specifically developed for accelerating the standard `BigInteger` class: they do not directly expose to the user a class corresponding to the `mpz` native type and only implements a limited selection of integer operations which are useful for cryptography applications. Despite the limitations, some of these libraries have been used several times in other projects. For instance, `jna-gmp` [24] is used in 23 GitHub projects (all in Java except one in Scala).

GMP has also been used by the Kaffe virtual machine [25] and the GNU Classpath [26] implementation of the Java standard library. Both projects are now essentially abandoned.

5. Conclusions

JGMP implements a set of wrappers for the GMP (GNU Multiple Precision Arithmetic) library which is functionally complete, extensively tested and performant. The library is memory and thread safe. It provides both mutable and immutable versions of most operations, and it is specifically designed to adhere to Java naming conventions. This allows to have an easy and complete way to deal with arbitrary precision arithmetic in Java and other Java bytecode based languages. The library is available in the Maven central repository with group id `it.unich.jgmp` and artifact id `jgmp`, so that it is very simple to use in a Java project. The JGMP library has been tested on Linux, Windows 10 and macOS (on the `x86-64` architecture).

The additional package `it.unich.jgmpbenchmarks` provides some benchmarks which compute factorials, probable prime numbers, and approximations of π. Contrary to the common thought that calling an external library in Java may slow down the execution, the benchmarks in the previous section show that using the JGMP library may improve the performance and outperforms the other libraries in most cases, in particular when you have to compute with very large numbers.

We think that JGMP has all the qualities to take the role of the standard Java wrapper for the GMP library, and it has a great potential to be widely used in many Java projects, so that other developers and researchers which wants to use GMP in Java projects do not need to waste time implementing ad-hoc and incomplete solutions.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

References

[1] The GNU Multiple Precision Arithmetic Library URL https://gmplib.org.

[2] Bailey David H. High-precision floating-point arithmetic in scientific computation. Comput Sci Eng 2005;7(3):54–61. http://dx.doi.org/10.1109/MCSE.2005.52.

[3] Bailey David H, Barrio R, Borwein JM. High-precision computation: Mathematical physics and dynamics. Appl Math Comput 2012;218(20):10106–21. http://dx.doi.org/10.1016/j.amc.2012.03.087.

[4] Wolfram Research, Inc. Mathematica, Version 13.2. URL https://www.wolfram.com/mathematica/.

[5] Maplesoft. Maple URL https://www.maplesoft.com/products/Maple/.

[6] Journault Matthieu, Miné Antoine, Monat Raphaël, Ouadjaout Abdelraouf. Combinations of reusable abstract domains for a multilingual static analyzer. In: Verified Software. Theories, Tools, and Experiments. 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13–14, 2019, Revised Selected Papers. Lecture Notes in Computer Science, 12031, Berlin Heidelberg: Springer; 2020, p. 1–18. http://dx.doi.org/10.1007/978-3-030-41600-3_1.

[7] Vojdani Vesal, Apinis Kalmer, Rõtov Vootele, Seidl Helmut, Vene Varmo, Vogler Ralf. Static race detection for device drivers: the Goblint approach. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. New York, NY, USA: ACM; 2016, p. 391–402. http://dx.doi.org/10.1145/2970276.2970337.

[8] Kirchner Florent, Kosmatov Nikolai, Prevosto Virgile, Signoles Julien, Yakobowski Boris. Frama-C: A software analysis perspective. Form Asp Comput 2015;27(3):573–609. http://dx.doi.org/10.1007/s00165-014-0326-7.

[9] Amato Gianluca, Di Maio Simone Di Nardo, Scozzari Francesca. Numerical static analysis with Soot. In: Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis. SOAP '13, New York, NY, USA: ACM; 2013, p. 25–30. http://dx.doi.org/10.1145/2487568.2487571.

[10] Amato Gianluca, Scozzari Francesca. Random: R-based analyzer for numerical domains. In: Bjørner Nikolaj, Voronkov Andrei, editors. Logic for Programming, Artificial Intelligence, and Reasoning 18th International Conference, LPAR-18, MÉRida, Venezuela, March 11-15, 2012. Proceedings. Lecture Notes in Computer Science, 7180, Berlin Heidelberg: Springer; 2012, p. 375–82. http://dx.doi.org/10.1007/978-3-642-28717-6_29.

[11] Calcagno Cristiano, Distefano Dino. Infer: An automatic program verifier for memory safety of C programs. In: Bobaru Mihaela, Havelund Klaus, Holzmann Gerard J, Joshi Rajeev, editors. NASA Formal Methods. Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011, Proceedings. Lecture Notes in Computer Science, 6617, Berlin Heidelberg: Springer; 2011, p. 459–65. http://dx.doi.org/10.1007/978-3-642-20398-5_33.

[12] Bertot Yves, Magaud Nicolas, Zimmermann Paul. A proof of GMP square root. J Automat Reason 2002;29(3–4):225–52. http://dx.doi.org/10.1023/A:1021987403425.

[13] Bagnara Roberto, Hill Patricia M, Zaffanella Enea. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci Comput Program 2008;72(1–2):3–21. http://dx.doi.org/10.1016/j.scico.2007.08.001.

[14] Tsai Yu-Hsin, Wu I-Wei, Liu I-Chun, Shann Jean Jyh-Jiun. Improving performance of JNA by using LLVM JIT compiler. In: Press IEEEComputer Society, editor. 2013 IEEE/ACIS 12th International Conference on Computer and Information Science. ICIS, 2013, p. 483–8. http://dx.doi.org/10.1109/ICIS.2013.6607886.

[15] Zakusylo Alexander. java-native-benchmark, JMH performance benchmark for Java's native call APIs URL https://github.com/zakgof/java-native-benchmark.

[16] Tommila Mikko. Apfloat, An arbitrary precision arithmetic library URL http://www.apfloat.org/apfloat_java/.

[17] JScience, Java Tools and Libraries for the Advancement of Sciences, URL http://jscience.org.

[18] Foundation The Apache Software. Apache Commons Numbers, Number types and utilities, URL https://commons.apache.org/proper/commons-numbers/.

[19] mpfr-java, MPFR Java Bindings, URL https://github.com/runtimeverification/mpfr-java.

[20] Gurinovich Andrew. GMP wrapper for Java URL https://github.com/altmind/GMP4J.

[21] Luo Ximin. GMP-based implementation of BigInteger for Java URL https://github.com/infinity0/gmp-java.

[22] Deshommes Didier. JNI wrapper to the GMP library URL https://github.com/dfdeshom/GMP-java.

[23] Pacurar Adrian. A Java wrapper for some libgmp integer functions commonly used in cryptography URL https://github.com/mathybit/java-gmp.

[24] Blum Scott, Wharton Jake, Wilson Jesse, McCauley Nathan, Perito Daniele, Quigley Sam, Humphries Josh, Meier Christian, Zupancic Elijah, Henecka Wilko, Ruescas David, Wilder Jacob, Allansson Per. A Java JNA wrapper around the GNU Multiple Precision Arithmetic Library, URL https://github.com/square/jna-gmp.

[25] Kaffe, The Kaffe Virtual Machine, URL https://github.com/kaffe/kaffe.

[26] GNU Classpath, Essential libraries for Java, URL https://www.gnu.org/software/classpath/.