



BARTŁOMIEJ SZCZYGIEL  (Warszawa)
LESZEK MARCINKOWSKI * (Warszawa)

A review note on arbitrary precision arithmetic

Abstract In this paper, we present a note on arbitrary precision. We give two simple examples showing the need of using arbitrary precision arithmetic. Next, we discuss how to use arbitrary precision arithmetic types in MATLAB/OCTAVE and further present short descriptions of several basic, in particular C/C++, packages for using arbitrary precision arithmetic in numerical codes for scientific computations. Finally, we discuss the contribution of one of the authors in the development of a library for arbitrary precision floating point numbers briefly.

2010 Mathematics Subject Classification: Primary: 65G50; Secondary: 68W30.

Key words and phrases: arbitrary precision arithmetic, floating point numbers, scientific computing.

1. Introduction The classical standard IEEE 32-bit floating-point arithmetic gives sufficiently accurate results in most practical or scientific applications. For other applications, the standard IEEE 64-bit floating-point arithmetic is wanted. Quite often a mix of these arithmetics gives satisfactory results. E.g. we can use lower precision in general and switch to higher precision just in sensitive numerical parts of computational code.

However, in some complicated more demanding scientific applications, especially in computational mathematical physics, a higher accuracy is required. Many scientists and engineers think that even 64-bit arithmetic is not accurate enough and that in their high-scale computations, they need higher accuracy. As the authors of [3] report, in the ATLAS experiment at the Large Hadron Collider when the scientists tried to track charged particles with very high precision, the scientists noted that a change in some used numerical libraries caused some particle collisions to disappear and some were misidentified. This suggests that their code has high numerical sensitivity.

We refer to [2] and [3], cf. also [1], and references therein for many other examples from mathematical physics. In the next section, we just list some areas where it is especially evident that we need a higher precision accuracy.

2. Why do we need higher accuracy arithmetic First, we give two very simple but characteristic motivational examples.

* The corresponding author.

2.1. Ill-condition linear systems A first example is a very simple computational problem, i.e. for given $n + 1$ equidistant points $x_k = k * h$ for $k = 0, \dots, n$ with $h = 1/n$ on the segment $[0, 1]$ find the coefficient of Lagrange interpolation polynomial $p(x) = \sum_{k=0}^n p_k x^k$ in the standard basis (representation) interpolating given values y_k , cf. e.g. § 2 in [22] or [10], i.e.

$$p(x_k) = y_k \quad k = 0, \dots, n.$$

Naturally, the simplest natural approach is to solve the linear system:

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & x_1^n \\ \vdots & & & & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^n \end{pmatrix} \begin{pmatrix} p_0 \\ p_1 \\ \vdots \\ p_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}.$$

Let us try to find the polynomial for $n = 20$ and $y_j = \sum_{k=0}^n x_j^k$, i.e. the solution should be the constant coefficient vector with $p(k) = 1$ for all k . When trying to solve the system using the GNU OCTAVE, cf. [6] the command: `p=polyfit(x,y,n)` gives that

$$p = \begin{pmatrix} 1.017850, & 0.821746, & 1.828503, \\ -1.379965, & 5.732167, & -5.911756, \\ 8.680575, & -5.635452, & 5.515481, \\ -1.437400, & 2.045784, & 0.644181, \\ 1.095348, & 0.980117, & 1.003167, \\ 0.999625, & 1.000032, & 0.999998, \\ 1.000000, & 1.000000, & 1.000000 \end{pmatrix}^T$$

We see that the computed result is completely wrong. There are negative values in p . Naturally, we can easily explain why this happens. We know that it is because the linear system is extremely ill-conditioned, in this case, we have that the condition number of the matrix equals to approximately $9.086e * 10^{16}$. We computed the condition number using the GNU OCTAVE, cf. [6], command: `cond(V)`, while the Vandermonde matrix by `V=vander(x)`. Here x is the vector with the equidistant $n+1$ nodes. The condition number in GNU Octave is computed utilizing SVD (Singular Value Decomposition), cf. e.g. Theorem (6.4.10) in [22]. The computations were done in default octave precision, i.e. in IEEE 64bit precision named also double precision. We see that the condition number is of the order of the double-precision arithmetic. It is worth mentioning that the used version of GNU Octave, cf. [6], uses the QR decomposition, cf. e.g. § 4.7 in [22], to solve the system when a user calls `polyfit()`. If a user calls the default linear solve operator i.e. `\` then Octave utilizes LU decomposition computed by the Gaussian elimination algorithm with partial pivoting, cf. e.g. § 4.1 in [22], which may give us a bit different digits in the numerical solutions. The interpolation problem fortunately may be solved in another way, e.g. using a divided difference algorithm

which should solve it for the much higher degree of the polynomial, cf. § 2.1.3 in [22]. Nevertheless, there are many other problems where a higher precision than the double one is necessary.

If we repeat solving the problem for $n = 10$ in IEEE 32-bit precision arithmetic also named single precision, we get

$$p = \begin{pmatrix} 1.161945, & 0.172458, & 2.812910, \\ -1.223598, & 2.672307, & 0.206553, \\ 1.235126, & 0.958561, & 1.003882, \\ 0.999855, & 1.000001 \end{pmatrix}^T$$

Again we see that the result is wrong since instead of the constant coefficient vector we got the negative value of one coefficient. It is not surprising that the situation got worse for smaller n , or equivalently we see that using double precision improved the situation, i.e. we could solve the problem for larger polynomial degrees. If someone wants to repeat the tests one can use the code from § 2.4 on page 47 in [23], which is freely available [online](#).

2.2. Solving Ordinary Differential Equation for a long time We have a very simple toy example, namely, we consider the following boundary value ODE problem: find $u : [0, T] \rightarrow R$ such that

$$-u'' + u = 0 \tag{1}$$

$$u(0) = u(T) = 1 \tag{2}$$

Naturally, it is a linear ODE with constant coefficients and we can easily compute the analytical solution: $u_T(t) = (e^{t-T} + e^{-t}) / (1 + e^{-T})$. Nevertheless, let's try to solve it numerically by the shooting method, cf. e.g. § 18.1 in [18] or § 7.3.2 in [22], for $T = 17$. If we apply the shooting method, we first look for s such that the solution of the Initial Value Problem with the ODE (1) and the following Initial Value:

$$\begin{aligned} u(0) &= 1 \\ u'(0) &= s \end{aligned} \tag{3}$$

solves our BVP (1)-(2). The shooting method is based on solving the IVP (1)-(3) with two different values of the initial value of derivative s in (3), e.g. $s_0 = 0, s_1 = 1$. We then get two functions u_0, u_1 . Then it is easy to compute $s = \frac{1 - u_0(T)}{u_1(T) - u_0(T)}$ such that the solution of IVP (1)-(3) with this s solves the original BV (2). The two IVPs (1)-(3) can be solved by a good IVP solver. We will use a reliable black box GNU Octave solver `lsode()` which is a solver from `lsode` math Fortran library, cf. e.g. [9, 19]. Let's consider the computed solution u_h for some $T \in \{1, \dots, 16\}$.

In Tables 1-2 we see the following errors in s and $u_h(T)$ (let $u_h(T)$ denote the computed solution by `lsode()` at $t = T$ for the initial values of the

Table 1: The first column contains the values of the end of the interval. The second column is the real value of the derivative of the analytical solution at the left end. The third column contains the computed value of the derivative at the end left by the shooting method.

T	$u'_T(0)$	$s_{h,T}^*$
1.0	$-4.621171573e - 01$	$-4.621171552e - 01$
4.0	$-9.640275801e - 01$	$-9.640275826e - 01$
7.0	$-9.981778976e - 01$	$-9.981780246e - 01$
10.0	$-9.999092043e - 01$	$-9.999094813e - 01$
13.0	$-9.999954794e - 01$	$-9.999959072e - 01$
16.0	$-9.999997749e - 01$	$-1.000000358e + 00$

Table 2: The first column contains the values of the end of the interval. While the next two columns contain the relative error in s and in $u(T)$.

T	$\frac{ s_{h,T}^* - u'_T(0) }{ s_{h,T} }$	$ u_h(T) - 1 $
1.0	$-4.37e - 09$	$7.25e - 08$
4.0	$-2.62e - 09$	$8.93e - 07$
7.0	$-1.27e - 07$	$6.61e - 05$
10.0	$-2.77e - 07$	$3.04e - 03$
13.0	$-4.28e - 07$	$9.46e - 02$
16.0	$-5.83e - 07$	$2.59e + 00$

derivative at the left end of the time interval: $s_{h,T}^*$ computed by the shooting method):

We see in the both tables that the error in s is not that bad, it grows but we get at least seven-digit accuracy, while the error of the numerical solution at the right end grows very rapidly and for the last values of T gives us completely wrong solutions. Both errors are above two (the absolute error here is equal to the relative one). Naturally, it shows that solving the IVP (1)-(3) requires higher precision arithmetic, and this is a very simple but a good example of the need for higher accuracy in the case of a relatively long computation time. For short time intervals, our problem can be solved in double precision, but for a bit longer time we need higher accuracy. Again it is worth mentioning that this problem can be solved by another numerical approach, namely, we can use a Finite Difference Method, cf. e.g. § 7.4 in [22], or a Finite Element Method, cf. e.g. [5], for the original BVP (1)-(2).

Those two simple examples can be easily resolved by different approaches, but it is very important to note that most scientists in many branches of

science like e.g. chemistry are not experts in advanced numerical methods or even basic numerical algorithms. The simplest approach which requires much higher precision may be for them a natural choice. Seeking the help of experts may be very difficult if impossible and may hinder the development of their research, in which scientific computation is just an important but not a crucial part.

In general, higher precision arithmetic is needed among others e.g.

- ill-conditioned linear problems,
- computing periodic orbits in dynamical systems,
- simulations for a very long time,
- large scale summations,
- parallel computations - to reproduce the results, cf. e.g. [21],
- computer-assisted mathematical proofs, - sometimes one use alternatively interval arithmetic, cf. e.g. [14, 15, 16, 20],
- high-precision computations e.g. computing zeros of Riemann Zeta Function, cf. [4].

3. Libraries for High-Precision arithmetic In this section, we will briefly discuss some numerical libraries which enable us to use high or even arbitrary-precision arithmetic.

Some commercial packages like Maple, cf. [11] or Mathematica, cf. [12], enable users to use higher precision arithmetic. We will give some more details about the possibilities of using higher-precision arithmetic in MATLAB, cf. [13], which seems the most popular commercial numerical computation software package, and its free clone GNU OCTAVE, cf. [6].

3.1. Matlab and GNU octave In both software packages, the variable precision arithmetic is available in their respective extra symbolic packages, i.e. in Symbolic Math toolbox in MATLAB and Symbolic package in OCTAVE, cf. [17]. The main function is `vpa(x,N)`. It creates a variable-precision floating point number of N digits. X may be a string, a sym (symbolic expression), or a double. A simple example in OCTAVE/MATLAB (executed in OCTAVE):

```
>> vpa(sqrt(3))
```

```
Symbolic pkg v2.9.0:
```

```
Python communication link active ,
```

SymPy v1.7.1.

```
ans = (sym) 1.7320508075688771931
766041234368
```

```
>> sqrt(3)
```

```
ans = 1.732050807568877
```

The last two lines show the standard double result of $\sqrt{3}$, i.e. $\sqrt{3}$.

Other useful commands in OCTAVE/MATLAB are `digits(N)`, `vpasolve()`, `sym()`. The first sets/gets the number of digits in variable precision arithmetic. The second, i.e. `vpasolve(eqn,x,x0)` solves numerically a symbolic equation for variable `x` using initial guess `x0`, while `sym` defines symbols and numbers as symbolic expressions. We will give another simple example:

```
>> x=sym('x')
x = (sym) x
>> eqn=x^2==2;
>> vpasolve(eqn,x,1.0)
ans = (sym) 1.414213562373095048801
6887242097
>> vpasolve(eqn,x,-1.0)
ans = (sym) -1.414213562373095048801
6887242097
>> vpasolve(eqn,x)
ans = (sym) 1.414213562373095048801
6887242097
>> digits(45)
>> vpasolve(eqn,x)
ans = (sym) 1.414213562373095048801
68872420969807856967188
```

The last parameter, i.e. an initial guess is optional. As we see we can increase the default vpa precision (or decrease) using the `digits()` function.

There are also external libraries for high-precision arithmetic in MATLAB (or OCTAVE). The first one is an open source high precision matrix library for Matlab and GNU/Octave: GEM library, cf. <https://gem-library.github.io/gem/>. The library implements two data types:

- `gem` - high-precision dense matrices,
- `sgem` - high precision sparse matrices

and many Matlab/Octave functions are overloaded.

It is worth noting that the `vpa` class of Matlab (or OCTAVE) does not support sparse matrices formats and is quite slow compared to GEM.

Another worth recommending extra toolbox for MATLAB is ADVAN-PIX which is a Multiprecision Computing Toolbox, the MATLAB extension for computing with arbitrary precision. <https://www.advanpix.com/>. This commercial toolbox adds to MATLAB a new higher-precision floating-point numeric type and a large number of mathematical functions that can compute with arbitrary precision, e.g. solvers for linear equations, nonlinear solvers, singular value decomposition (SVD), numerical integration, optimization ODEs, etc. It is possible to replace the standard floating point type of MATLAB with multi-precision numbers and thus run old MATLAB codes without extensive changes.

3.2. C/C++, Python etc There are several packages for higher precision that can be used in C/C++ codes. We will mention the most important ones and some interfaces (wrappers) to some of them:

- GNU Multiple Precision Floating-Point Reliable Library (GNU MPFR library), cf. <https://www.mpfr.org/>, [7], is a GNU C library for multiple-precision floating-point computations with correct rounding. It is dependent on GNU Multi-Precision Library (GNU GMP see below). - the current newest version is version 4.1.1 released on November 17, 2022. The MPFR is free, released under the GNU Lesser GPL license. It has several extensions and interfaces mentioned below. It implements not only simple arithmetic operations like “+” and “/” but also has a whole set of mathematical functions like `sin`, `sqrt`, `pow`, `log` etc. MPFR does not track the accuracy of numbers in the program or expression,
- MPFI a C multiple-precision interval arithmetic library, based on MPFR, cf. <https://gitlab.inria.fr/mpfi/mpfi>. The MPFI has some of the mathematical functions provided by MPFR,
- GNU MPC library is a library for multiple-precision complex arithmetic based on the MPFR and GMP libraries,
- MPFRCPP is a C++ interface for MPFR. It has classes, templates, and function objects,
- MPFR C++ is another interface to MPFR. This wrapper introduces a new C++ type for high-precision floating point numbers – `mpreal` which contains low level `mpfr_t` a custom C-language type of MPFR that enables a representation of floating-point numbers, This enables a user to use MPFR computations as simply as can be done with built-in types `double` or `float`,
- `gmpfrxx` C++ is another wrapper for both GMP (see below) and MPFR,
- Boost C++ libraries also have an interface for MPFR, it is a part of its Multiprecision library,

- SageMath, cf. <https://www.sagemath.org/>, is a free open-source math software system built on top of many open-source packages like SymPy, NumPy, and SciPy. It is a computer system based on Python language. It contains Arbitrary Precision Real Numbers, it is a binding to MPFR arbitrary-precision floating-point library,
- GPM - The GNU Multiple Precision Arithmetic Library, cf. <http://gmplib.org/> - it is a free library for arbitrary precision arithmetic including signed integers, rational numbers, and fl numbers. The current stable release is 6.2.1 released in November 2020,
- PARI/GP, cf. <https://pari.math.u-bordeaux.fr/>, is a computer algebra system designed for fast computations in number theory, but it also has functions to compute with matrices, polynomials, power series, algebraic numbers, etc. PARI is also available as a C library,
- MPFUN2020, ARPEC, cf. <https://www.davidhbailey.com/dhbsoftware/> - high precision software packages developed by David H. Bailey. - the first one is in 2 versions - a stand-alone one and the 2nd based on MPFR. ARPREC is an older arbitrary precision library for Fortran and C++,
- CLN - Class Library for Numbers, cf. <https://www.ginac.de/CLN/> - CLN is a C++ library for computations with all types of numbers in any arbitrary precision. The current version is 1.3.6,
- Arb - a C library for arbitrary-precision ball arithmetic, cf. <https://arblib.org/>. It is for rigorous real and complex arithmetic with any precision. Numerical errors are estimated using ball arithmetic, which is a type of interval arithmetic,

and others...

4. Eigen library In this section, we will briefly describe the contribution of the first author of this paper to the development of the possibility of using high-precision arithmetic in an open-source library Eigen, cf. [8]. Eigen is a C++ template library for numerical linear algebras, in particular, it supports all types of matrices, even sparse matrices, all standard numerical types, as well as standard matrix factorization or decompositions like LU, Cholesky, QR SVD, eigendecompositions, etc

The first author B.S. modified the library MPFR C++, which is a wrapper (interface) for the C library MPFR (which cannot be used in EIGEN as a standard C library). MPFR C++ enables a user to utilize numbers from MPFR as other numerical floating point types, so that any other library can use these numbers. Unfortunately, MPFR C++ uses dynamically allocated memory for a mantissa of a given precision fl variable. This breaks memory coherence and adds another memory jump on every access. The version of B.S.

creates a new generic type, a template for a variable of given precision, where the mantissa is part of the variable. This should improve the way memory is accessed (jump to memory is still there, but in the area right next to it, which is likely to be in the cache already). The disadvantage is the larger size of the variable.

5. Summary In this note, we discuss an urgent need of using arbitrary precision arithmetic and present briefly how to use it in MATLAB/OCTAVE. We also present a short overview of the main **C/C++** libraries for arbitrary precision arithmetic.

Acknowledgments: The first author Bartłomiej Szczygiel was partially supported by the Ministry of Science and Education and Science, Poland - Project No. DI2016 0001 46.

- [1] D. H. Bailey and J. M. Borwein. High-precision numerical integration: progress and challenges. *J. Symbolic Comput.*, 46(7):741–754, 2011. ISSN 0747-7171. doi: [10.1016/j.jsc.2010.08.010](https://doi.org/10.1016/j.jsc.2010.08.010). Cited on p. 153.
- [2] D. H. Bailey, R. Barrio, and J. M. Borwein. High-precision computation: mathematical physics and dynamics. *Appl. Math. Comput.*, 218(20): 10106–10121, 2012. ISSN 0096-3003. doi: [10.1016/j.amc.2012.03.087](https://doi.org/10.1016/j.amc.2012.03.087). MR 2921767. Cited on p. 153.
- [3] D. H. Bailey, R. Barrio, and J. M. Borwein. High-precision arithmetic in mathematical physics. *Mathematics*, 3(2):337–367, 2015. doi: [10.3390/math3020337](https://doi.org/10.3390/math3020337). Cited on p. 153.
- [4] G. Beliakov and Y. Matiyasevich. A parallel algorithm for calculation of determinants and minors using arbitrary precision arithmetic. *BIT*, 56(1):33–50, 2016. ISSN 0006-3835. doi: [10.1007/s10543-015-0547-z](https://doi.org/10.1007/s10543-015-0547-z). Cited on p. 157.
- [5] D. Braess. *Finite elements*. Cambridge University Press, Cambridge, third edition, 2007. ISBN 978-0-521-70518-9; 0-521-70518-5. doi: [10.1017/CBO9780511618635](https://doi.org/10.1017/CBO9780511618635). URL <https://doi.org/10.1017/CBO9780511618635>. Theory, fast solvers, and applications in elasticity theory, Translated from the German by Larry L. Schumaker. Cited on p. 156.
- [6] J. W. Eaton, D. Bateman, S. Hauberg, and R. Wehbring. *GNU Octave version 6.1.0 manual: a high-level interactive language for numerical computations*, 2020. URL <https://www.gnu.org/software/octave/doc/v6.1.0/>. Cited on pp. 154 and 157.
- [7] L. Fousse, G. Hanrot, V. Lefèvre, P. Péliissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct

- rounding. *ACM Transactions on Mathematical Software*, 33(2):13:1–15, 2007. doi: [10.1145/1236463.1236468](https://doi.org/10.1145/1236463.1236468). Cited on p. 159.
- [8] G. Guennebaud, B. Jacob, et al. Eigen v3.4.0. <http://eigen.tuxfamily.org>, August 2021. Cited on p. 160.
- [9] A. C. Hindmarsh. ODEPACK, a systematized collection of ODE solvers. In S. R. S., C. M., P. R., A. W. F., and V. R., editors, *Scientific Computing. Applications of Mathematics and Computing to the Physical Sciences*, pages 55–64. North-Holland, Amsterdam, 1983. Cited on p. 155.
- [10] D. Kincaid and W. Cheney. *Numerical analysis*. Brooks/Cole Publishing Co., Pacific Grove, CA, second edition, 1996. ISBN 0-534-33892-5. Mathematics of scientific computing. MR 1388777. Cited on p. 154.
- [11] Maple. Maple, Maplesoft, a division of Waterloo Maple Inc., 2022. URL <https://hadoop.apache.org>. Cited on p. 157.
- [12] Mathematica. Mathematica, Version 13.2, Wolfram Research, Inc., 2022. URL <https://www.wolfram.com/mathematica>. Cited on p. 157.
- [13] MatLab. Matlab, version 9.13 (r2022a), The MathWorks Inc., 2022. Cited on p. 157.
- [14] K. R. Meyer and D. S. Schmidt, editors. *Computer aided proofs in analysis*, volume 28 of *The IMA Volumes in Mathematics and its Applications*, 1991. Springer-Verlag, New York. ISBN 0-387-97426-1. doi: [10.1007/978-1-4613-9092-3](https://doi.org/10.1007/978-1-4613-9092-3). Cited on p. 157.
- [15] R. E. Moore. Interval tools for computer aided proofs in analysis. In *Computer aided proofs in analysis (Cincinnati, OH, 1989)*, volume 28 of *IMA Vol. Math. Appl.*, pages 211–216. Springer, New York, 1991. doi: [10.1007/978-1-4613-9092-3_17](https://doi.org/10.1007/978-1-4613-9092-3_17). Cited on p. 157.
- [16] M. T. Nakao, M. Plum, and Y. Watanabe. *Numerical verification methods and computer-assisted proofs for partial differential equations*, volume 53 of *Springer Series in Computational Mathematics*. Springer, Singapore, 2019. ISBN 978-981-13-7668-9; 978-981-13-7669-6. doi: [10.1007/978-981-13-7669-6](https://doi.org/10.1007/978-981-13-7669-6). MR 3971222. Cited on p. 157.
- [17] Octave. GNU Octave Symbolic Package, August 2022. URL <https://gnu-octave.github.io/packages/symbolic/>. Cited on p. 157.
- [18] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes. The art of scientific computing*. Cambridge University Press, Cambridge, third edition, 2007. ISBN 978-0-521-88068-8. Section 18.1 The shooting method. MR 2371990. Cited on p. 155.

- [19] K. Radhakrishnan and A. C. Hindmarsh. Description and use of lsode, the Livermore solver for ordinary differential equations. Technical Report UCRL-ID-113855, Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States), 12 1993. Cited on p. 155.
- [20] S. M. Rump. Verification methods: rigorous results using floating-point arithmetic. *Acta Numer.*, 19:287–449, 2010. ISSN 0962-4929. doi: [10.1017/S096249291000005X](https://doi.org/10.1017/S096249291000005X). MR 2652784. Cited on p. 157.
- [21] V. Stodden, D. H. Bailey, J. Borwein, R. J. LeVeque, W. Rider, and W. Stein. Setting the default to reproducible. Reproducibility in computational and experimental mathematics. Technical report, The Institute for Computational and Experimental Research in Mathematics, February 2013. URL http://faculty.washington.edu/rjl/pubs/icerm2012/icerm_report.pdf. Cited on p. 157.
- [22] J. Stoer and R. Bulirsch. *Introduction to numerical analysis*, volume 12 of *Texts in Applied Mathematics*. Springer-Verlag, New York, third edition, 2002. ISBN 0-387-95452-X. doi: [10.1007/978-0-387-21738-3](https://doi.org/10.1007/978-0-387-21738-3). Translated from the German by R. Bartels, W. Gautschi and C. Witzgall. Cited on pp. 154, 155, and 156.
- [23] J. Valdman. *Vybrané úlohy řešené na počítači: diskrétní a numerická matematika*. Jihočeská univerzita v Českých Budějovicích, 2022. ISBN 978-80-7394-905-1. In Czech. Cited on p. 155.

REFERENCES

Przeglądowa notka o arytmetyce zmiennopozycyjnej dowolnej precyzji

Bartłomiej Szczygieł, Leszek Marcinkowski

Streszczenie W tej pracy przedstawiamy krótką przeglądową notkę na temat arytmetyki zmiennopozycyjnej dowolnej precyzji. Przedstawiamy dwa proste przykłady pokazujące konieczność wykorzystania takiej arytmetyki.

Omawiamy użycie arytmetyki zmiennopozycyjnej dowolnej precyzji w MATLABie/Octave'ie i dalej omawiamy krótko kilka podstawowych, w szczególności w C/C++, bibliotek zawierających arytmetykę dowolnej precyzji dla kodów (programów) numerycznych dla zastosowań w obliczeniach naukowych. Ostatecznie omawiamy wkład jednego z autorów w rozwój biblioteki dla arytmetyki dowolnej precyzji.

Klasyfikacja tematyczna AMS (2010): 65G50; 68W30; 65F05.


Słowa kluczowe: analiza numeryczna, arytmetyka zmiennopozycyjna dowolnej precyzji, obliczenia naukowe.




Bartłomiej Szczygiel was born in 1985 in Poland. He studied Applied Mathematics at the Faculty of Mathematics, Informatics, and Mechanics at the University of Warsaw (MIM UW). In 2016 he received a B.Sc. degree at MIM UW with the thesis *Percolation thresholds for discrete-continuous models with nonuniform probabilities of bond formation*. He later became a Ph.D. student at MIMUW. He co-authored two papers published in internationally recognized journals: *Physical Review* and *Applied Optics*. He participated in three scientific projects being a leader in one of them. As a student he also was granted two scholarships by the Warsaw Center of Mathematical Sciences and the Polish Ministry of Education, respectively. His current scientific interests are physics, computational mathematics, arbitrary precision arithmetic, and their applications.



Leszek Marcinkowski received an M.Sc. from the Faculty of Mathematics, Informatics, and Mechanics at the University of Warsaw (MIM UW) in 1994. The doctoral degree in mathematical sciences was awarded to him in 1999 by the Scientific Council of MIM UW. Habilitation in mathematical sciences was granted in 2010 by the Scientific Council of MIM UW. He had visiting research at the Department of Computer Science at the University of Colorado at Boulder, Center for Applied Scientific Computing in Lawrence Livermore National Laboratory, USA, and Shenzhen Institute of Advance Technology (SIAT) -Chinese Academy of Science, China. References to his research papers are listed in the Heidelberg Academy of Sciences bibliography database known as zbMath under [ai:Marcinkowski.Leszek](#), in MathSciNet under [ID:615388](#) and Scopus under [AU-ID:6602236217](#). His current research interests are focused on discretization methods and parallel solvers for partial differential equations. In particular, he is interested in discretizations built on nonmatching grids and domain decomposition methods.

BARTŁOMIEJ SZCZYGIEL 
 UNIVERSITY OF WARSAW
 FACULTY OF MATHEMATICS, INFORMATICS, AND MECHANICS, BANACHA 2, 02-097 WARSZAWA, POLAND
 E-MAIL: bartekltg@gmail.com

LESZEK MARCINKOWSKI 
 UNIVERSITY OF WARSAW
 FACULTY OF MATHEMATICS, INFORMATICS, AND MECHANICS, BANACHA 2, 02-097 WARSZAWA, POLAND
 E-MAIL: lmarcin@mimuw.edu.pl

(Received: 3rd of January 2023; revised: 11th of January 2023)
