Computer Physics Communications 270 (2022) 108167

Contents lists available at ScienceDirect

Computer Physics Communications

www.elsevier.com/locate/cpc

Implementation of high-precision computation capabilities into the open-source dynamic simulation framework YADE ☆

Janek Kozicki [a,b,*], Anton Gladky [c], Klaus Thoeni [d]

[a] Faculty of Applied Physics and Mathematics, Gdańsk University of Technology, 80-233 Gdańsk, Poland
[b] Advanced Materials Center, Gdańsk University of Technology, 80-233 Gdańsk, Poland
[c] Institute for Mineral Processing Machines and Recycling Systems Technology, TU Bergakademie Freiberg, 09599 Freiberg, Germany
[d] Centre for Geotechnical Science and Engineering, The University of Newcastle, 2308 Callaghan, Australia

ARTICLE INFO

Article history:
Received 20 April 2021
Received in revised form 9 August 2021
Accepted 6 September 2021
Available online 10 September 2021

Keywords:
Arbitrary accuracy
Multiple precision arithmetic
Dynamical systems 05.45.-a
Computational techniques 45.10.-b
Numerical analysis 02.60.Lj
Error theory 06.20.Dk

ABSTRACT

This paper deals with the implementation of arbitrary precision calculations into the open-source discrete element framework YADE published under the GPL-2+ free software license. This new capability paves the way for the simulation framework to be used in many new fields such as quantum mechanics. The implementation details and associated gains in the accuracy of the results are discussed. Besides the "standard" `double` (64 bits) type, support for the following high-precision types is added: `long double` (80 bits), `float128` (128 bits), `mpfr_float_backend` (arbitrary precision) and `cpp_bin_float` (arbitrary precision). Benchmarks are performed to quantify the additional computational cost involved with the new supported precisions. Finally, a simple calculation of a chaotic triple pendulum is performed to demonstrate the new capabilities and the effect of different precisions on the simulation result.

1. Introduction

The advent of a new era of scientific computing has been predicted in the literature, one in which the numerical precision required for computation is as important as the algorithms and data structures in the program [6,50,51,64]. An appealing example of a simple computation gone wrong was presented in the talk "Are we just getting wrong answers faster?" of Stadtherr in 1998 [85]. An exhaustive list of such computations along with a very detailed analysis can be found in [51].

Many examples exist where low-precision calculations resulted in disasters. The military identified an accumulated error in multiplication by a constant factor of 0.1, which has no exact binary representation, as the cause for a Patriot missile failure on 25 February 1991, which resulted in several fatalities [12]. If more bits were used to represent a number, the explosion of an Ariane 5 rocket launched by the European Space Agency on 4 June 1996 could have been prevented [58,56,49,11] as it was a result of an inappropriate conversion from a 64 bit floating point number into a 16 bit signed integer. Indeed, the 64 bit floating point number was too big to be represented as a 16 bit signed integer. On 14 May 1992 the rendezvous between the shuttle Endeavour and the Intelsat 603 spacecraft nearly failed. The problem was traced back to a mismatch in precision [74,43]. More catastrophic failures related to the lack of precision are discussed in [74,43]. In 2012 it was predicted that most future technical computing will be performed by people with only basic training in numerical analysis or none at all [59,36,6,51]. High-precision computation is an attractive option for such users, because even if a numerically better algorithm with smaller error or faster convergence is known for a given problem (e.g. Kahan summation [36] for avoiding accumulating errors[1]), it is often easier and more efficient to increase the precision of an existing algorithm rather than deriving and implementing a new one [6,50] — a feat which is made possible by the work presented in this paper. It shall however be noted that increasing precision is not the answer to all types of problems, as recently a new kind of a pathological systematic error of up to 14% has been discovered

☆ The review of this paper was arranged by Prof. Hazel Andrew.

* Corresponding author at: Faculty of Applied Physics and Mathematics, Gdańsk University of Technology, 80-233 Gdańsk, Poland.
E-mail addresses: jkozicki@pg.edu.pl (J. Kozicki), Anton.Gladky@iart.tu-freiberg.de (A. Gladky), klaus.thoeni@newcastle.edu.au (K. Thoeni).

[1] For n summands and ε Unit in Last Place (ULP) error, the error in regular summation is $n\varepsilon$, error in Kahan summation is 2ε, while error with regular summation in twice higher precision is $n\varepsilon^2$. See proof of Theorem 8 in [36].

https://doi.org/10.1016/j.cpc.2021.108167

Table 1
Selected modules and features in YADE.

cmake flag	Description
	High-precision support in present YADE version[37].
(always on)	Discrete Element Method [98,53].
(always on)	Deformable structures [28,15,95].
ENABLE_CGAL	Polyhedral particles, polyhedral particle breakage [29,34].
ENABLE_LBMFLOW	Fluid-solid interaction in granular media with coupled Lattice Boltzmann/Discrete Element Method [60].
ENABLE_POTENTIAL_PARTICLES	Arbitrarily shaped convex particle described as a 2nd degree polynomial potential function [14].
	Selected YADE features with high-precision support.
ENABLE_VTK	Exporting data and simulation geometry to ParaView [98]
(always on)	Importing geometry from CAD/CAM software (`yade.ymport`) [98].
ENABLE_ASAN	AddressSanitizer allows detection of memory errors, memory leaks, heap corruption errors and out-of-bounds accesses [83].
ENABLE_OPENMP	OpenMP threads parallelization, full support for `double`, `long double`, `float128` types[8].
	Modules under development for high-precision support.
ENABLE_MPI	MPI environment for massively parallel computation [98].
ENABLE_VPN	Thermo-hydro-mechanical coupling using virtual pore network [54,55].
ENABLE_NRQM	Quantum dynamics simulations of diatomic molecules including photoinduced transitions between the coupled states [46,45].

in a certain type of Bernoulli map calculations which cannot be mitigated by increasing the precision of the calculations [13]. In addition, switching to high-precision generally means longer run times [44,31].

Nowadays, high-precision calculations find application in various different domains, such as long-term stability analysis of the solar system [57,90,6], supernova simulations [39], climate modeling [41], Coulomb n-body atomic simulations [8,33], studies of the fine structure constant [99,100], identification of constants in quantum field theory [16,7], numerical integration in experimental mathematics [9,62], three-dimensional incompressible Euler flows [18], fluid undergoing vortex sheet roll-up [7], integer relation detection [4], finding sinks in the Henon Map [48] and iterating the Lorenz attractor [1]. There are many more yet unsolved high-precision problems [86], especially in quantum mechanics and quantum field theory where calculations are done with 32, 230 or even 10000 decimal digits of precision [73,84,16]. Additionally Debian, a Linux distribution with one of the largest archives of packaged free software is now moving numerous numerical computation packages such as Open MPI, PETSc, MUMPS, SuiteSparse, ScaLAPACK, METIS, HYPRE, SuperLU, ARPACK and others into 64-bit builds [26]. In order to stay ahead of these efforts, simulations frameworks need to pave the way into 128-bit builds and higher.

The open-source dynamic simulation framework YADE [53,98] is extensively used by many researchers all over the world with a large, active and growing community of more than 25 contributors. YADE, which stands for "Yet Another Dynamic Engine", was initially developed as a generic dynamic simulation framework. The computation parts are written in C++ using flexible object models, allowing independent implementation of new algorithms and interfaces. Python (interpreted programming language, which wraps most of C++ YADE code) is used for rapid and concise scene construction, simulation control, postprocessing and debugging. Over the last decade YADE has evolved into a powerful discrete element modeling package. The framework benefits from a great amount of features added by the YADE community, for example particle fluid coupling [35,60,63], thermo–hydro-mechanical coupling [20,54,55], interaction with deformable membrane-like structures, cylinders and grids [28,15,95], FEM-coupling [47,32,38], polyhedral particles [14,29,34], deformable particles [40], brittle materials [81,27], quantum dynamics of diatomic molecules [46,45] and many others. A more extensive list of publications involving the use of YADE can be found on the framework's web page [94]. A list of selected available YADE modules and features is presented in Table 1. Although its current focus is on discrete element simulations of granular material, its modular design allows it to be easily extended to new applications that rely on high-precision calculations.

The present work deals with the implementation of high-precision support for YADE which will open the way for YADE to be used in many new research areas such as quantum mechanics [46,45], special relativity, general relativity, cosmology, quantum field theory and conformal quantum geometrodynamics [78,25]. The programming techniques necessary for such extension are presented and discussed in Section 2. Relevant tests and speed benchmarks are performed in Sections 3 and 4. A simple chaotic triple pendulum simulation with high precision is presented in Section 5. Finally, conclusions are drawn and it is discussed how this new addition to the framework will enable research in many new directions.

2. Implementation of arbitrary precision

2.1. General overview

Since the beginning of YADE [53], the declaration '`using Real=double;`'[2] was used as the main floating point type with the intention to use it instead of a plain `double` everywhere in the code. The goal of using `Real` was to allow replacing its definition with other possible precisions.[3] Hence, the same strategy was followed for other types used in the calculations, such as vectors and matrices. Per definition the last letter in the type name indicates its underlying type, e.g. '`Vector3r v;`' is a 3D vector $\vec{v} \in \widetilde{\mathbb{Q}}^3 \in \mathbb{R}^3$, and `Vector2i` is a 2D vector of integers (where $\widetilde{\mathbb{Q}}$ is a subset of rational numbers $\mathbb{Q}$, which are representable by the currently used precision: $\widetilde{\mathbb{Q}} \in \mathbb{Q} \in \mathbb{R}$; the name `Real` is used instead of `Rational` or `FloatingPoint` for the sake of brevity).

[2] Originally YADE was written in C++03, hence, before the switch to C++17 it was '`typedef double Real;`'.
[3] See for example: https://answers.launchpad.net/yade/+question/233320.

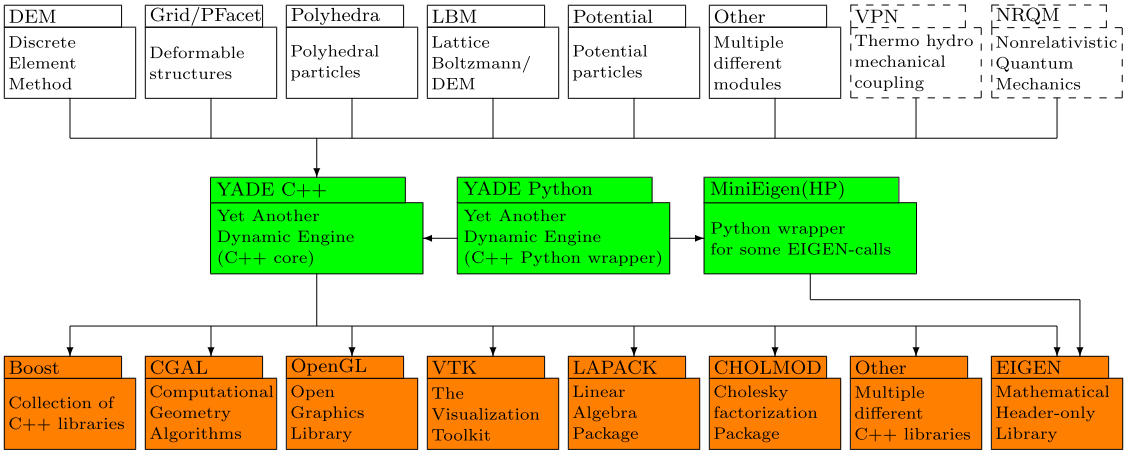

Fig. 1. Simplified dependency tree of the open-source framework YADE. External dependencies are marked in orange. The green boxes indicate parts of the framework that needed to be adapted for high-precision. Selected YADE modules which support high-precision are in the top row, dashed lines indicate modules under development (also see Table 1). (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

Table 2
List of high-precision types supported by YADE.

Type	Total bits	Decimal places	Exponent bits	Significant bits	Notes
`float`	32	6	8	24	only for testing
`double`	64	15	11	53	hardware accelerated
`long double`†	80	18	15‡	64	hardware accelerated
`boost float128`§	128	33	15	113	may be hardware accelerated
`boost mpfr`§	N	$N\log_{10}(2)$	–	–	MPFR library as wrapped by Boost
`boost cpp_bin_float`§	N	$N\log_{10}(2)$	–	–	uses Boost only, but is slower

† The specifics of `long double` depend on the particular compiler and hardware; the values in this table correspond to the most common x86 platform and the `g++` compiler.
‡ All types use 1 bit to store the sign and all types except `long double` have an implicit first bit=1, hence here the sum $15+64\neq 80$.
§ The complete C++ type names for the Boost high-precision types are as follows: `boost::multiprecision::float128`, `boost::multiprecision::mpfr_float_backend` and `boost::multiprecision::cpp_bin_float`.

In the presented work, the goal to use high precision is achieved by using the C++ operator overloading functionality and the `boost::multiprecision` library. A simplified dependency diagram of YADE is shown in Fig. 1. The layered structure of YADE remains nearly the same as in the original paper by Kozicki and Donzé [53]. It is built on top of several well established libraries (marked with orange in Fig. 1) as discussed in Section 2.3. Some changes were necessary in the structure of the framework (marked with green in Fig. 1) as highlighted in Section 2.5. The top row in Fig. 1 indicates selected YADE modules with respective citations listed in Table 1. It should be noted that YADE relies on many external libraries to expand its functionality which can result in a demanding server setup.

The Boost library [24] provides convenient wrappers for other high-precision types with the perspective of adding more such types in the future.[4] The new supported `Real` types are listed in Table 2. A particular `Real` type can be selected during compilation of the code by providing a `cmake` argument either `REAL_PRECISION_BITS` or `REAL_DECIMAL_PLACES`.[5]

The process of adding high-precision support to YADE was divided into several stages which are described in the subsections below.[6]

2.2. Preparations

To fully take advantage of the C++ Argument Dependent Lookup (ADL), the entire YADE codebase was moved into `namespace yade`,[7] thus using the C++ standard capabilities to modularize the namespaces for each software package. Similarly, the libraries used by YADE such as Boost [24], CGAL [93] and EIGEN [37] reside in their respective `boost`, `CGAL` and `Eigen` namespaces. After this change, all potential naming conflicts between math functions or types in YADE and these libraries were eliminated.

Before introducing high precision into YADE it was assumed that `Real` is actually a Plain Old Data (POD) `double` type. It was possible to use the old C-style `memset`, `memcpy`, `memcmp` and `memmove` functions which used raw-memory access. However, by doing so the modern C++ structure used by other high-precision types was completely ignored. For example, the MPFR type may reserve memory and inside its structure store a pointer to it. Trying to set its value to zero by invoking `memset` (which sets that pointer to `nullptr`) leads to a memory leak and a subsequent program failure. In order to make `Real` work with other types, this assumption had to be removed. Hence, `memset` calls were replaced with `std::fill` calls, which when invoked with a POD type reduce to a (possibly faster) version of `memset` optimized for a particular type in terms of chunk size used for writing to the memory. In addition, C++ template specialization mechanisms allow for invoking with a non-POD type which then utilizes the functionality provided by this specific type, such as calling the spe-

[4] At the time of writing, the quad–double library with 62 decimal places (package `libqd-dev`) is in preparation, see: https://github.com/boostorg/multiprecision/issues/184.
[5] See http://yade-dem.org/doc/HighPrecisionReal.html for detailed documentation.
[6] Also see the consolidated merge request: !383.
[7] See: !284.

cific constructors. All places in the code which used these four raw-memory access functions were improved to work with the non-POD `Real` type.[8] For similar reasons one should not rely on storing an address of the n^{th} component of a `Vector3r` or `Vector2r`.[9]

Next, all remaining occurrences of `double` were replaced with `Real`[10] and the high-precision compilation and testing was added to the gitlab Continuous Integration (CI) testing pipeline, which guarantees that any future attempts to use `double` type in the code will fail before merging such changes into the main branch. Next the `Real` type was moved from global namespace into `yade` namespace[11] to eliminate any potential problems with namespace pollution.[12]

2.3. Library compatibility

In order to be able to properly interface YADE with all other libraries it was important to make sure that mathematical functions (see Table 4) are called for the appropriate type. For example, the EIGEN library would have to call the high-precision `sqrt` function when invoking a `normalize` function on a `Vector3r` in order to properly calculate vector length. Several steps were necessary to achieve this. First, an inline redirection[13] to these functions was implemented in namespace `yade::math` in the file `MathFunctions.hpp`. Next, all invocations in YADE to math functions in the `std` namespace were replaced with calls to these functions in the `yade::math` namespace.[14] Functions which take only `Real` arguments may omit `math` namespace specifier and use ADL instead. Also some fixes were done in EIGEN and CGAL,[15] although they did not affect YADE directly since it was possible to workaround them.

The C++ type traits is a template metaprogramming technique which allows one to customize program behavior (also called polymorphism) depending on the type used [2,67,89,96]. This decision is done by the compiler (conditional compilation) due to inspecting the types in the compilation stage (this is called static polymorphism). Advanced C++ libraries provide hooks (numerical traits) to allow library users to inform the library about the used precision type. The numerical traits were implemented in YADE for the libraries EIGEN and CGAL[16] as these were the only libraries supporting such a solution at the time of writing this paper. EIGEN and CGAL are fully compatible and aware of the entire high-precision code infrastructure in YADE. Similar treatment would be possible for the Coinor [77,61] library (used by the class `PotentialBlock`) if it would provide numerical traits. Additionally, an OpenGL compatibility layer has been added by using inline conversion of arguments from `Real` to `double` for the OpenGL functions as OpenGL drawing functions use `double`.[17] The VTK compatibility layer was added using a similar approach. Virtual functions were added to convert the `Real` arguments to `double`[18] in classes derived from the VTK parent class (such as `vtkDoubleArray`).

The LAPACK compatibility layer was provided as well, this time highlighting the problems of interfacing with languages which do not support static polymorphism. The routines in LAPACK are written in a mix of Fortran and C, and have no capability to use high-precision numerical traits like EIGEN and CGAL. The only way to do this (apart from switching to another library) was to down-convert the arguments to `double` upon calling LAPACK routines (e.g. a routine to solve a linear system) then up-converting the results to `Real`. This was the first step to phase out YADE's dependency on LAPACK. With this approach the legacy code works even when high precision is enabled although the obtained results are low-precision.[19] Additionally, this allows one to test the new high-precision code against the low-precision version when replacing these function calls with appropriate function calls from another library such as EIGEN in the future. Fortunately, only two YADE modules depend on LAPACK: potential particles and the flow engine [21]. The latter also depends on CHOLMOD, which also supports the `double` type only, hence it is not shown in Table 1. Nevertheless, a similar solution as currently implemented for LAPACK can be used in the future to remove the current dependency on CHOLMOD.

2.4. Double, quadruple and higher precisions

Sometimes a critical section of the computations in C++ would work better if performed in a higher precision[1]. This would also guarantee that the overall results in the default precision are correct. The `RealHP<N>` types serve this purpose. In analogy to `float` and `double` types used on older systems, the types `RealHP<2>`, `RealHP<4>` and `RealHP<N>` correspond to double, quadruple and higher multipliers of the `Real` precision selected during compilation, e.g. with `REAL_DECIMAL_PLACES`[5], respectively. A simple example where this can be useful is solving a system of linear equations where some coefficients are almost zero. The old rule of thumb to *"perform all computation in arithmetic with somewhat more than twice as many significant digits as are deemed significant in the data and are desired in the final results"* works well in many cases [50]. Nevertheless, maintaining a high quality scientific software package without being able to use, when necessary, arithmetic precision twice as wide can badly inflate costs of development and maintenance [50]. On the one hand, there might be additional costs for the theoretical formulation of such tricky single-precision problems. On the other hand, the cost of extra demand for processor cycles and memory when using `RealHP<N>` types is picayune when compared with the cost of a numerically adept mathematician's time [51]. Hence, the new `RealHP<N>` makes high and multiple-precision simulations more accessible to the researcher community.

The support for higher precision multipliers was added in YADE[20] in such a way that `RealHP<1>` is the `Real` type from Table 2 and every higher number `N` is a multiplier of the `Real`

[8] See: !381, with one exception which is yet to be evaluated and hence `mpfr` and `cpp_bin_float` types work with OpenMP, but do not take full advantage of CPU cache size in class `OpenMPArrayAccumulator`.

[9] See: !406.

[10] These changes were divided into several smaller merge requests: !326, !376, !394; there were also a couple of changes such as `MatrixXd` → `MatrixXr` and `Vector3d` → `Vector3r`.

[11] See: !364.

[12] Usually such errors manifest themselves as very unrelated problems, which are notoriously difficult to debug, e.g. due to the fact that an incorrect type (with the same name) is used; see: #57 and https://bugs.launchpad.net/yade/+bug/528509.

[13] The recommended practice in such cases is to use the Argument Dependent Lookup (ADL) which lets the compiler pick the best match from all the available candidate functions [2,67,89,96]. No ambiguity is possible, because such situations would always result in a compiler error. This was done by employing the C++ directives `using std::function;` and `using boost::multiprecision::function;` for the respective *function* and then calling the function unqualified (without namespace qualifier) in the `MathFunctions.hpp` file.

[14] See: !380, !390, !391, !392, !393, !397.

[15] See: https://gitlab.com/libeigen/eigen/-/issues/1823 and https://github.com/CGAL/cgal/issues/4527.

[16] See files `EigenNumTraits.hpp`, `CgalNumTraits.hpp` and !412.

[17] See: !412 and file `OpenGLWrapper.hpp`. If the need for drawing on screen with precision higher than `double` arises (e.g. at high zoom levels) it will be rectified in the future.

[18] See: !400 and `VTKCompatibility.hpp`. If the VTK display software will start supporting high precision, this solution can be readily improved.

[19] See: !379 and `LapackCompatibility.cpp`.

[20] See: !496.

precision. All other types follow the same naming pattern: `Vector3rHP<1>` is the regular `Vector3r` and `Vector3rHP<N>` uses the precision multiplier `N`. A similar concept is used for CGAL types (e.g. `CGALtriangleHP<N>`). One could then use an EIGEN algorithm for solving a system of linear equations with a higher `N` using `MatrixXrHP<N>` to obtain the result with higher precision. Then, after the critical code section, one could potentially continue the calculations in the default `Real` precision. On the Python side the mathematical functions for the higher precision types are accessible via `yade.math.HP2.*`. By default only the `RealHP<2>` is exported to Python. One can export to Python all the higher types for debugging purposes by adjusting `#define YADE_MINIEIGEN_HP` in the file `RealHPConfig.hpp`.

On some occasions it is useful to have an intuitive up-conversion between C++ types of different precisions, say for example to add `RealHP<1>` to `RealHP<2>`. The file `UpconversionOfBasicOperatorsHP.hpp` serves this purpose. After including this header, operations using two different precision types are possible and the resultant type of such operation will always be the higher precision of the two types. This header should be used with caution (and only in `.cpp` files) in order to still be able to take advantage of the C++ static type checking mechanisms. As mentioned in the introduction, this type checking whether a number is being converted to a fewer digits representation can prevent mistakes such as the explosion of the rocket Ariane 5 [58,56,49,11].

2.5. Backward compatibility with older YADE scripts

In the present work, preserving the backward compatibility with existing older YADE Python scripts was of prime importance. To obtain this the MiniEigen Python library had to be incorporated into YADE's codebase. The reason for this was the following: `python3-minieigen` was a binary package, precompiled using `double`. Thus any attempt of importing MiniEigen into a YADE Python script (i.e. using `from minieigen import *`) when YADE was using a non-`double` type resulted in failure. This, combined with the new capability in YADE to use any of the current and future supported types (see Table 2) would place a requirement on `python3-minieigen` that it either becomes a header-only library or is precompiled with all possible high-precision types. It was concluded that integrating its source directly into YADE is the most reasonable solution. Hence, old YADE scripts that use supported modules[37] can be immediately converted to high precision by switching to `yade.minieigenHP`. In order to do so, the following line:

```
from minieigen import *
```

has to be replaced with:

```
from yade.minieigenHP import *
```

Respectively `import minieigen` has to be replaced with `import yade.minieigenHP as minieigen`, the old name `as minieigen` being used for the sake of backward compatibility with the rest of the script.

Python has native support[21] for high-precision types using the `mpmath` Python package. However, it shall be noted that although the coverage of YADE's basic testing and checking (i.e. `yade --test` and `yade --check`) is fairly large, there may still be some parts of Python code that were not yet migrated to high precision and may not work well with the `mpmath` module. If such problems occur in the future, the solution is to put the non compliant Python function into the `py/high-precision/math.py` file.[22]

A typical way of ensuring correct treatment of `Real` in Python scripts is to initialize Python variables using `yade.math.Real(arg)`. If the initial argument is not an integer and not an `mpmath` type then it has to be passed as a string (e.g. `yade.math.Real('9.81')`) to prevent Python from converting it to `double`. Without this special initialization step a mistake can appear in the Python script where the default Python floating-point type `double` is for example multiplied or added to the `Real` type resulting in a loss of precision.[23]

3. Testing

It should be noted that it is near to impossible to be absolutely certain about the lack of an error in a code [51]. Therefore, to briefly test the implementation of all mathematical functions available in C++ in all precisions, the following test was implemented in `_RealHPDiagnostics.cpp` and run by using the Python script `testMath.py`. Each available function was evaluated 2×10^7 times with on average evenly spaced pseudo-random argument in the range $(-100, 100)$. These 2×10^7 evaluations were divided into two sets. The first 10^7 evaluations were performed with uniformly distributed pseudo-random numbers in the range $(-100, 100)$. The second 10^7 evaluations were done by randomly displacing a set of 10^7 equidistant points in the range $(-100, 100)$, where each point was randomly shifted by less than ± 0.5 of the distance between the points. This random shift was to lower the chances of duplicate calculations on the same argument after adjusting to the function domain. The 2×10^7 arguments were subsequently modified to match the domain argument range of each function using simple operations, such as `abs(•)` or `fmod(abs(•),2)-1`. The obtained result for each evaluation was then compared against its respective `RealHP<4>` type with four times higher precision. Care was taken to exactly use the same argument for a higher precision function call. The arguments were randomized and adjusted to the function domain range in the lower precision `RealHP<1>`, then the argument was converted to the higher precision by using `static_cast`, thereby ensuring that all the extra bits in the higher precision are set to zero.

The difference expressed in terms of Units in the Last Place (ULP) [51] was calculated. The obtained errors are listed in Table 4. During the tests a bug in the implementation of the `tgamma` function for `boost::multiprecision::float128` was discovered but it was immediately fixed by the Boost developers.[24] Some other bug reports[25] instigated a discussion about possible ways to fix the few problems found with the `cpp_bin_float` type which can be seen in the last column of Table 4. A smaller version of this test, with only 2×10^4 pseudo-random evaluations[26] was then added to the standard `yade --test` invocation.

Finally, an AddressSanitizer [83,10] was employed to additionally check the correctness of the implementation in the code and to quickly locate memory access bugs. Several critical errors were fixed due to the reports of this sanity checker. This tool is now integrated into the Continuous Integration (CI) pipeline for the whole YADE project to prevent introduction of such errors in the future [10] (`make_asan_HP` job in the GitLab CI pipeline).

[21] See: `ToFromPythonConverter.hpp` file.

[22] See also: !414.

[23] See: !604 and commit 494548b82d, where a small change in the Python script enabled it to work for high precision.

[24] See: https://github.com/boostorg/math/issues/307.

[25] See: https://github.com/boostorg/multiprecision/issues/264 and https://github.com/boostorg/multiprecision/issues/262.

[26] Because this test is time consuming it is not possible to run the test involving 2×10^7 evaluations in the GitLab CI pipeline after each `git push`.

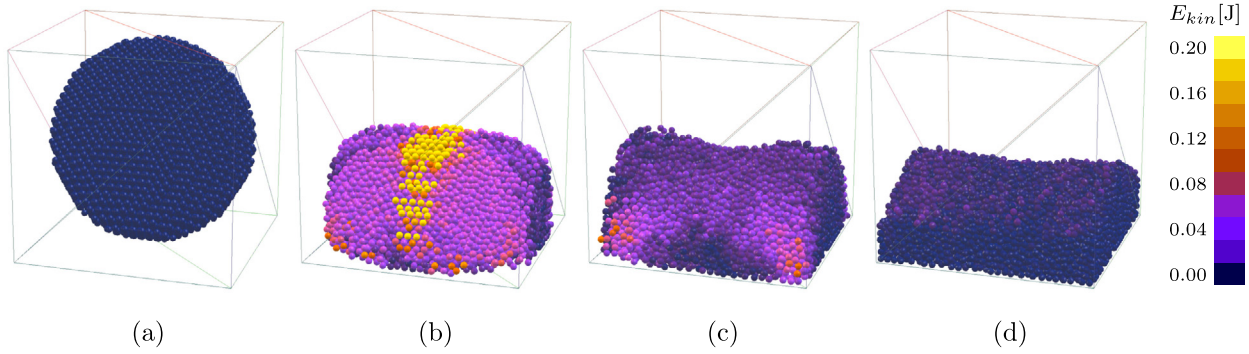

Fig. 2. Snapshots of the benchmark `yade --stdperformance -j16` with `mpfr` 150 at different time steps: (a) $t = 0$ s, (b) $t = 0.3$ s (3,000 iterations), (c) $t = 0.4$ s (4,000 iterations), (d) $t = 0.7$ s (7,000 iterations). The particles are colored by kinetic energy.

4. Benchmark

A benchmark[27] `yade --stdperformance -j16` (16 OpenMP threads [23,71]) on a PC with two `Intel E5-2687W v2 @ 3.40 GHz` processors (each of the two having 8 cores resulting in a total of 16 cores or 32 threads if hyperthreading is enabled) was performed to assess performance of higher precision types. The benchmark consists of a simple gravity deposition of spherical particles into a box, a typical simulation performed in YADE. A spherical packing with 10,000 spheres is released under gravity within a rectangular box. The spheres are allowed to settle in the box (Fig. 2). The simulation runs for 7,000 iterations and the performance is reported in terms of iterations per wallclock seconds. This standardized test (hence `--stdperformance` in the name) is constructed in such a way that almost all the computation happens on the C++ side, only the calculation of the wallclock time is done in Python. Obviously doing more calculations in Python will make any script slower. Hence, any calculation in Python should be kept to a minimum.

Since the benchmark results strongly depend on other processes running on the system, the test was performed at highest process priority after first making sure that all unrelated processes are stopped (via `pkill -SIGSTOP` command). The benchmark was repeated at least 50 times for each precision type and compiler settings. The average calculation speed $\bar{x}$ (in iterations per seconds) was determined for each precision type and data points not meeting the criterion $2\sigma < x_i - \bar{x} < 2\sigma$ (σ is the standard deviation) were considered outliers. On average 4% of data points per bar was removed, the largest amount removed was 10% (5 data points) on two occasions. Hence, each bar in Fig. 3 represents the average of at least 45 runs using 7,000 iterations each.

A summary of all the benchmark results is shown in Fig. 3 along with the relevant standard deviations. The performance is indicated in terms of iterations per seconds. The tests were performed[28,30] for the seven different precision types (Fig. 3A–G) listed in Table 3 for three different optimization settings: default `cmake` settings (Fig. 3a), with SSE vectorization enabled (Fig. 3b), and with maximum optimizations offered by the compiler but without vectorization[29] (Fig. 3c). The lack of significant improvement in the third case (Fig. 3c) shows that the code is already well optimized and the compiler cannot optimize it any further, except for `long double` and `float128` types and the `gcc` compiler where a 2% speed gain can be observed (Fig. 3Cc and Dc versus Ca and Da). It is interesting to note that the `clang` compiler systematically produced a code that runs about 4 to 9% faster than the `gcc` or `icpc` compilers. Intel compiler users should be careful, because the `-fast` switch might result in a performance loss of around 2 to 10%, depending on particular settings (Fig. 3c). Code vectorization (using the SSE assembly instruction set, an experimental feature, Fig. 3b) provides about 1 to 3% speed gain, however this effect is often smaller than the σ error bars. Enabling intel hyperthreading (HT) did not affect the results more than the standard deviation error of the benchmark. The `float128` results for the intel compiler stand out with a 5% speed gain (Fig. 3D). However, not all mathematical functions are currently available for this precision in `icpc` and to get this test to work a crippled branch[30] was prepared for the tests with some of the mathematical functions disabled. The missing mathematical functions[31] were not required for these particular calculations to work. The `clang` compiler does not support[32] `float128` type yet. The average speed difference between each precision is listed in Table 3. The run time increase with precision in the MPFR library is roughly $\mathcal{O}(N \log(N))$ (where N is the number of digits used) but it is application specific and strongly depends on the type of simulation performed [44,31].

It shall be noted that currently YADE does not fully take advantage of the SSE assembly instructions (`cmake -DVECTORIZE=1`) because `Vector3r` is a three component type, while a four component class `Eigen::AlignedVector3`[33] is suggested in the

Table 3
The high-precision types used in the benchmark and corresponding speed performance relative to `double`.

Type	Decimal places	Speed relative to `double`
`float`	6	1.01× faster
`double`	15	–
`long double`	18	1.4× slower
`boost float128`†	33	4.7× slower
`boost mpfr`‡	62	13.5× slower
`boost mpfr`	150	19.1× slower
`boost cpp_bin_float`	62	24.2× slower

† Except for clang which does not yet[32] support `float128`.
‡ For future comparison with `libqd-dev`, see footnote[4].

[27] See: !388, !491 and the file: `examples/test/performance/checkPerf.py`.
[28] Also see https://gitlab.com/yade-dev/trunk/-/tree/benchmarkGcc.
[29] The test with SSE and maximum optimizations was also performed but the results were simply additive, thus they were not included here. Also sometimes they produced the following error due to memory alignment problems: http://eigen.tuxfamily.org/dox-devel/group__TopicUnalignedArrayAssert.html, because the operands of an SSE assembly SIMD instruction set must have their addresses to be a multiple of 32 or 64 bytes, and the compiler could not always guarantee this.
[30] See https://gitlab.com/yade-dev/trunk/-/tree/benchmarkIntel.
[31] See changes in `MathFunctions.hpp` in commit `3b07475e38` in `benchmarkIntel` branch.
[32] See https://github.com/boostorg/math/issues/181.
[33] Four `double` components in `Eigen::AlignedVector3` use 256 bits which matches SSE operations, the fourth component is unused and set to zero.

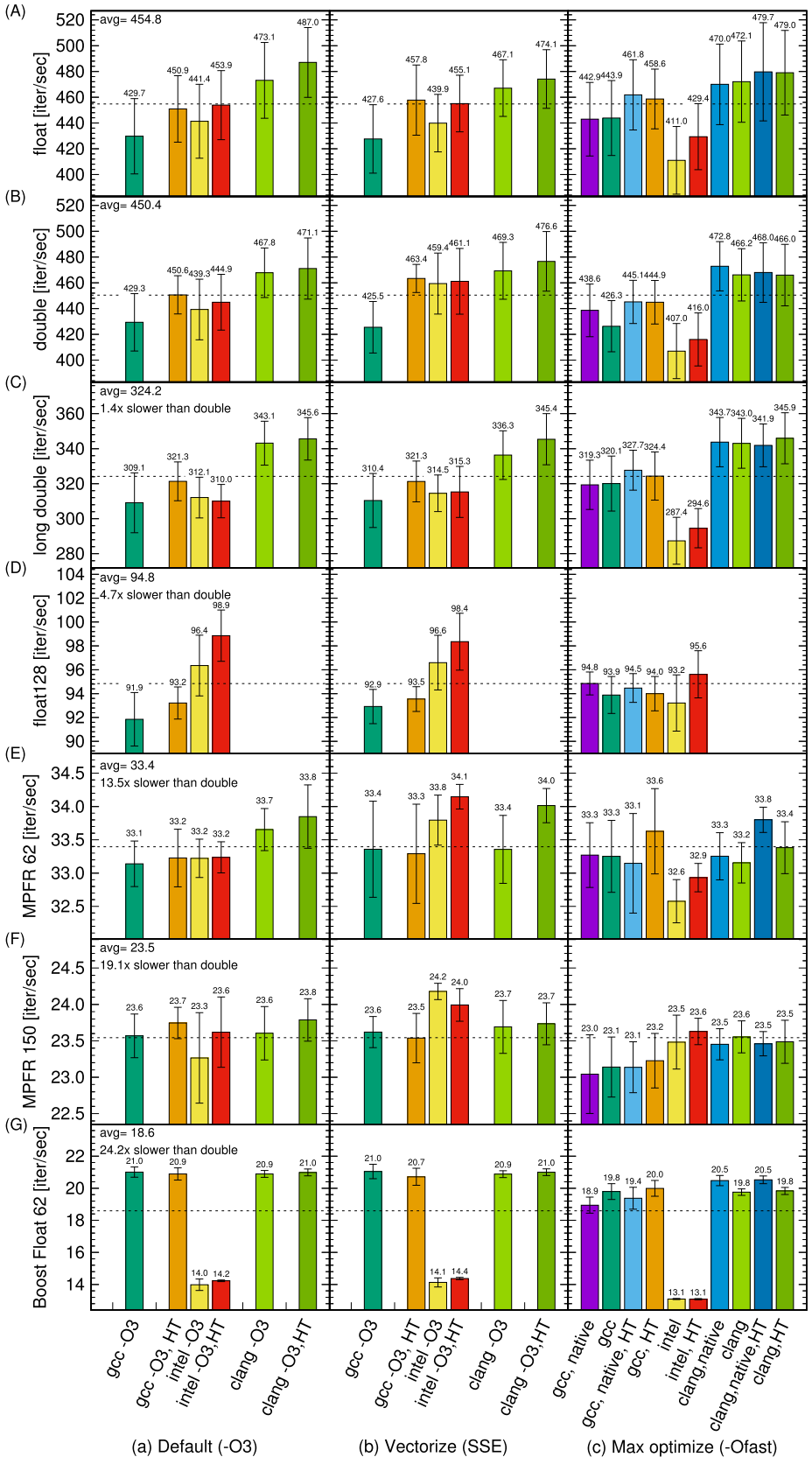

Fig. 3. Benchmark results[†] `yade --stdperformance -j16` for seven different precision types, with hyperthreading disabled[‡] and enabled (HT); for gcc version `9.3.0`, clang version `10.0.1-+rc4-1`, and intel icpc compiler version `19.0.5.281 20190815`: (a) default `cmake` settings; (b) with SSE vectorization enabled via `cmake -DVECTORIZE=1`; (c) with maximum optimizations offered by the compiler (gcc, clang: `-Ofast -fno-associative-math -fno-finite-math-only -fsigned-zeros`[§] and additionally for native: `-march=native -mtune=native`; intel `icpc -fast`[§]) but without vectorization. [†] On a PC with two `Intel E5-2687W v2 @ 3.40 GHz` processors with 16 cores and 32 threads. [‡] Via command `echo off > /sys/devices/system/cpu/smt/control` or by a BIOS setting (HT is also known as intel SMT). [§] The extra three flags are used by CGAL; in intel compilers `-fast` enforces all processor model `native` optimizations.

Table 4
Maximum error after 2×10^7 function evaluations expressed in terms of Units in the Last Place (ULP)[†], calculated by the absolute value of `boost::math::float_distance`[‡] between functions from C++ standard library or `boost::multiprecision` when compared with its respective `RealHP<4>` (having four times higher precision)[§].

	Type and number of decimal places						
	`float`	`double`	`long double`	`float128`	`mpfr`		`cpp_bin_float`
Decimal places	6	15	18	33	62	150	62
Significand bits	24	53	64	113	207	500	207
`+`	0	0	0	0	0	0	0
`-`	0	0	0	0	0	0	0
`*`	0	0	0	1	0	0	0
`/`	0	0	0	1	0	0	0
`sin`	1	1	1	1	0	0	6.43×10^7
`cos`	1	1	1	1	0	0	6.69×10^7
`tan`	1	0	2	1	0	0	7.64×10^7
`sinh`	2	2	3	2	0	0	125
`cosh`	2	1	2	1	0	0	125
`tanh`	2	2	3	2	0	0	9
`asin`	1	0	1	1	0	0	106
`acos`	1	0	1	1	0	0	13526
`atan`	1	0	1	1	0	0	8
`asinh`	2	2	3	3	0	0	16
`acosh`	2	2	3	3	1	1	17
`atanh`	2	2	3	3	0	0	23
`atan2`	1	0	1	2	0	0	10
`log`	1	1	1	1	0	0	17
`log10`	2	2	1	1	0	0	32
`log1p`	1	1	1	2	0	0	17
`log2`	1	1	1	2	0	0	25
`logb`	0	0	0	0	0	0	0
`exp`	1	1	1	1	0	0	125
`exp2`	1	1	1	1	0	0	5
`expm1`	1	1	2	2	0	0	125
`pow`	1	1	1	1	0	0	118
`sqrt`	0	0	0	1	0	0	0
`cbrt`	1	3	1	1	0	0	3
`hypot`	0	1	1	2	2	2	2
`erf`	1	1	1	1	0	0	21
`erfc`	3	4	3	3	0	0	22496
`lgamma`	6	8	7	7	0	0	70843
`tgamma`	7	7	7	7	0	0	10661
`fmod`	0	0	0	0	0	0	0
`fma`	0	0	0	1	0	0	1.95×10^5

[†] Also see [51]; please note that to obtain the number of incorrect *bits* one needs to take a $\log_2(\bullet)$ of the value in the table.
[‡] This test was performed with gcc version 9.3.0 and Boost library version 1.71.
[§] See file https://gitlab.com/yade-dev/trunk/-/blob/master/py/high-precision/_RealHPDiagnostics.cpp for implementation details. This test (with fewer evaluations) can be executed using `testMath.py` and is a part of the `yade --test` suite (file `py/tests/testMath.py`, function `testRealHPErrors`).

EIGEN library but it is not completely functional yet. In the future, this class can be improved in EIGEN and then used in YADE.

5. Simulation

5.1. Problem description

A simple simulation of a triple elastic pendulum system was performed to check the effect of high precision in practice. Triple pendulums are considered highly chaotic as they provide an irregular and complex system response [3]. Numerical modeling of such systems can show the benefits of using high precision. A single thread was used (i.e. `yade -j1`) during the simulations to avoid numerical artifacts arising from different ordering of arithmetic operations performed by multiple threads. This allowed focusing on high precision and eliminating the non-deterministic effect of parallel calculations (e.g. arithmetic operations performed in a different order result in a different ULP error in the last bits [36,76,41]) and to have a completely reproducible simulation.

The numerical setup of the model represents the chain consisting of three identical elastic pendulums (see attached Listing 1 and on `gitlab`). The pendulums are represented by a massless elastic rod (a long-range normal interaction) and mass points. The latter are modeled using spheres with radius $r = 0.001$ m and density $\rho = 1$ kg/m^3, noting that the masses of the spheres are lumped into a point. The rods are modeled by a normal interaction using cohesive interaction physics. The length of the chain is $L = 0.1$ m. Each rod is 1/30 m long and the normal stiffness of the interaction is $k = 100$ N/m. The strength (i.e. cohesion) is set to an artificial high value (10^7 N/m^2) so that the chain cannot break. Hence, the behavior of the rods can be assumed purely elastic. The initial position of the chain is $\alpha = -20°$ relative to the horizontal plane (see Fig. 4a, $t = 0$ s). Gravity $g = 9.81$ m/s^2 is acting on the chain elements as they are moving. The process was simulated with time steps Δt equal to 10^{-5} s, 10^{-6} s, 10^{-7} s, and 10^{-8} s. The results obtained using different precision are discussed in the following subsections. First, the evolution of the angles in the pendulum movement is discussed with $\Delta t = 10^{-5}$ s. Second, the effect of damping is shown with $\Delta t = 10^{-5}$ s. Then the effect of using

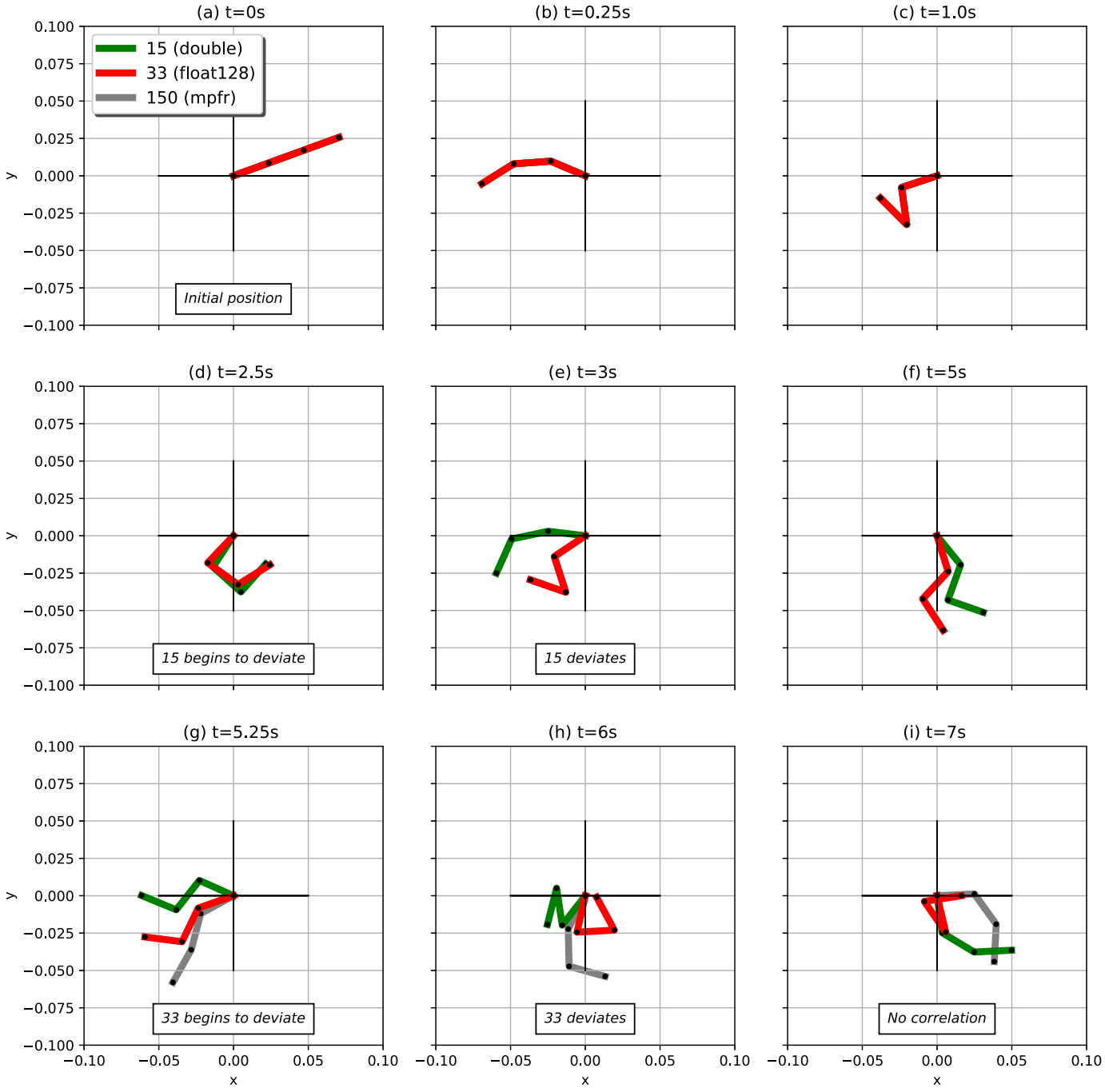

Fig. 4. Numerical simulation of the triple pendulum with 15 (double), 33 (float128) and 150 (mpfr) decimal places. The snapshots are captured at the following times: (a) $t = 0$ s, (b) $t = 0.1$ s, (c) $t = 0.25$ s, (d) $t = 2.5$ s, (e) $t = 3$ s, (f) $t = 5$ s, (g) $t = 5.25$ s, (h) $t = 6$ s, and (i) $t = 7$ seconds. The lines are showing the positions of the connected rods. Only three precisions from Table 5 are shown for clarity.

various time steps Δt is discussed. Finally, the total energy conservation is examined for various time steps.

5.2. Pendulum movement

Numerical damping was not used in this simulation series to avoid any energy loss and for the purity of the numerical results. The simulations were carried out with the different precisions listed in Table 5 and a time step of $\Delta t = 10^{-5}$ s. Angles between the two rods were constantly monitored and saved with a period of 10^{-4} s for further analysis and comparison. After the simulation was performed and the data was gathered, the Pearson correla-

Table 5
The high-precision types used in the simulation and corresponding correlation duration t_s for $\Delta t = 10^{-5}$.

Type	Decimal places	Correlation duration t_s
`float`	6	1.1 seconds
`double`	15	2.5 seconds
`long double`	18	3.1 seconds
`boost float128`	33	5.1 seconds
`boost mpfr`	62	9.9 seconds
`boost cpp_bin_float`	62	9.9 seconds

```
# The script is tested with yade_2021.01a
from yade import plot, qt, math

# initialize all floating point variables with Real(arg) to avoid precision loss
from yade.math import toHP1 as Real
# after release yade_2021.01a math.toHP1 has an alias Real, same with radiansHP1
if(math == sys.modules['math']): raise RuntimeError("Python math obscures yade.math")

### set parameters ###
L = Real('0.1')      # length [m]
n = 4                # number of nodes for the length [-]
r = L/100            # radius [m]
g = Real('9.81')     # gravity
inclination = math.radiansHP1(20)  # Initial inclination of rods [degrees]
color = [1, 0.5, 0]  # Define a color for bodies

O.dt = Real('1e-05') # time step
damp = Real('1e-1')  # damping. It is interesting to examine  damp = 0

O.engines = [  # define engines, main functions for simulation
    ForceResetter(),
    InsertionSortCollider([Bo1_Sphere_Aabb()],
                          label='ISCollider', avoidSelfInteractionMask=True),
    InteractionLoop(
        [Ig2_Sphere_Sphere_ScGeom6D()],
        [Ip2_CohFrictMat_CohFrictMat_CohFrictPhys(
            setCohesionNow=True, setCohesionOnNewContacts=False)],
        [Law2_ScGeom6D_CohFrictPhys_CohesionMoment()]
    ),
    NewtonIntegrator(gravity=(0, -g, 0), damping=damp, label='newton'),
]

# define material:
O.materials.append(CohFrictMat(young=1e5, poisson=0, density=1e1,
                               frictionAngle=math.radiansHP1(0), normalCohesion=1e7,
                               shearCohesion=1e7, momentRotationLaw=False, label='mat'))

# create spheres
nodeIds = []
for i in range(0, n):
    nodeIds.append(O.bodies.append(sphere([i*L/n*math.cos(inclination),
        i*L/n*math.sin(inclination), 0], r, wire=False, fixed=False,
        material='mat', color=color)))

# create rods
for i, j in zip(nodeIds[:-1], nodeIds[1:]):
    inter = createInteraction(i, j)
    inter.phys.unp = -(O.bodies[j].state.pos-O.bodies[i].state.pos).norm() + \
        O.bodies[i].shape.radius+O.bodies[j].shape.radius

O.bodies[0].dynamic = False  # set a fixed upper node
qt.View()                    # create a GUI view
Gl1_Sphere.stripes = True    # mark spheres with stripes
rr = qt.Renderer()           # get instance of the renderer
rr.intrAllWire = True        # draw wires
rr.intrPhys = True           # draw the normal forces between the spheres.
```

Listing 1: The triple pendulum simulation script.

tion coefficient [88] was calculated for all data sets. The simulation with 150 decimal places (type `mpfr` 150) was used as the reference solution as it has the highest number of decimal places. The product of the two angles between the three rods was used as an input parameter for the calculation of the correlation coefficient. The data for the correlation was placed in chunks with each having 500 elements (10^{-4} s $\times 500 = 0.005$ s of the simulation). Then the `scipy.stats.pearsonr` function from `SciPy` [97] was employed for the calculation of the Pearson correlation coefficient p. For further reference, the point in time when the correlation p between two simulations falls below $p < 0.9$ is marked as t_s. The latter corresponds to the time from the start of the simulation until the correlation is lost and in the following it is denoted correlation duration.

Fig. 4 shows snapshots of the evolution of the movement for three different precisions. At the beginning of the simulation the angles between the rods are the same. However, from a certain point in time onward the angles are starting to differ. Indeed, at $t = 2.5$ s (Fig. 4d), the green line (15 decimal places, type `double`) has clearly another state compared to the other two with 33 and 150 decimal places (types `float128` and `mpfr` 150 respectively). The snapshot at $t = 5.25$ s (Fig. 4g) demonstrates the beginning of the deviation of the simulation with 33 decimal places (type `float128`). Thereafter all the pendulums are moving very differently and no correlation is observed. It can be concluded that using higher precision increases the time when accurate calculation results are obtained which is also reflected by the correlation duration t_s listed in Table 5.

Fig. 5 presents the correlation coefficient as a function of time. The graphs provide a more accurate representation of the point in time when the correlation disappears. One can see that there is a positive linear correlation with $p = 1$ at the beginning of the simulations. This means initially the rods are moving identically. The lowest precision curve `float` is starting to jump between corre-

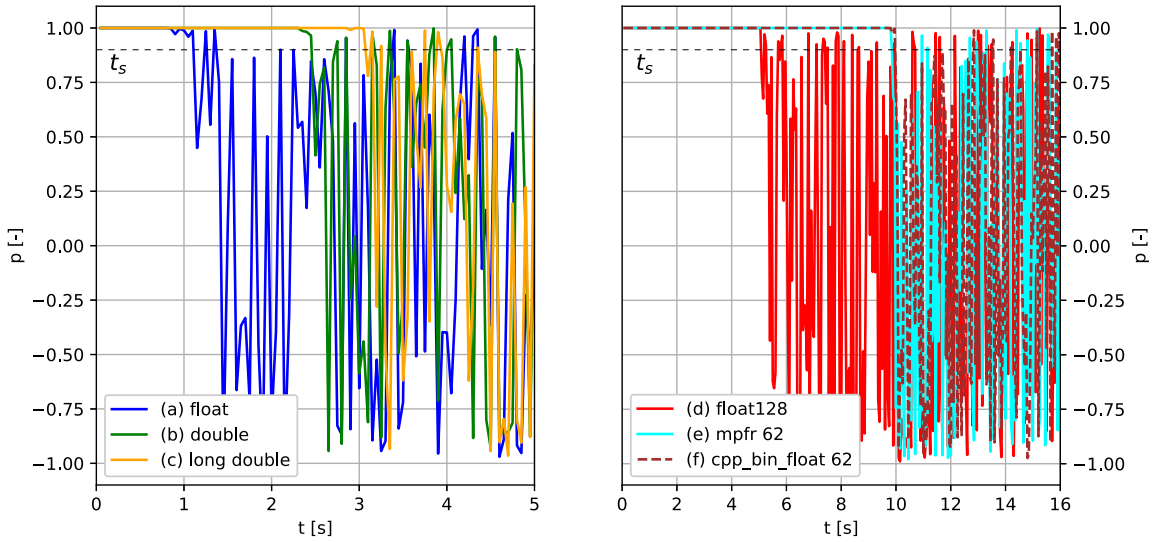

Fig. 5. Pearson correlation coefficient p as a function of time t between the results obtained with `mpfr` 150 and the following different precisions: (a) `float`; (b) `double`; (c) `long double`; (d) `float128`; (e) `mpfr` 62; (f) `cpp_bin_float` 62. Note that the timescales on both figures are different. The black dashed lines mark the threshold $p = 0.9$ which is used to calculate the correlation duration t_s.

lation values of $p = 1$ and $p = -1$ at around $t = 1.1$ s (Fig. 5a), which means that no visible correlation is observed any more. For all the higher precision simulations the drop off happens later and progressively with increasing precision as summarized in Table 5.

Type `double` and `long double` have a correlation of $p < 0.9$ after 2.5 s (Fig. 5b, comparing 15 with 150 decimal places) and 3.1 s (Fig. 5c, 18 vs. 150 decimal places) respectively. The same tendency is seen from Boost `float128` type (Fig. 5d, 33 vs. 150 decimal places) which deviates at approximately 5.1 s. Both simulations with 62 decimal places start to deviate at around 9.9 s. This clearly demonstrates that the level of precision, i.e. the number of decimal places, has an influence on the accuracy of the simulation results. Sometimes the decrease of the correlation happens suddenly and sometimes it starts to decrease slowly before it decreases rapidly. It can clearly be seen that the simulations with higher precision are showing better results and are closer to the reference solution calculated with 150 decimal places. Nevertheless, higher precision requires much more time for the simulation and more computing resources.

5.3. Effect of damping

To study the importance of precision in combination with other parameters, the same simulations as in Section 5.2 were carried out with a numerical damping coefficient equal to $5 \cdot 10^{-3}$. Numerical damping is generally applied to dissipate energy. In this particular test case, numerical damping can be interpreted as the slowing down of the pendulum oscillation. The global damping mechanism was used, as described in the original DEM publication of Cundall and Strack [22]. Global damping acts on the absolute velocities of the simulation bodies and is implemented in the `NewtonIntegrator` class in the source code of YADE. Global damping slows down all affected bodies based on their current velocities.

The damping coefficient was chosen so that an effect of different precisions can be seen on the whole system. If the damping coefficient is too high ($> 10^{-1}$), the system loses its whole energy very quickly and no visible differences are seen. Too small damping coefficients ($< 10^{-3}$) lead to very slow energy dissipation.

Fig. 6 shows the development of the angle between the first and the second rod of the pendulum during the simulation with

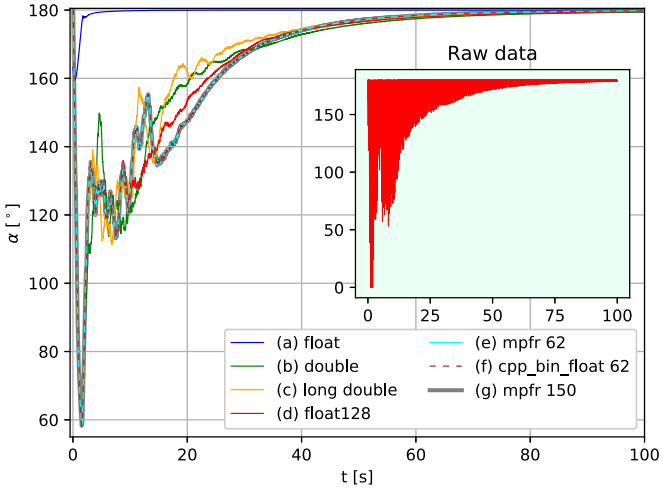

Fig. 6. Development of angle α between first and second rods as a function of time t for different precisions with a global damping coefficient equal to $5 \cdot 10^{-3}$. Curves are showing smoothed data for better visibility (main plot) and raw data for the inset; (a) `float`; (b) `double`; (c) `long double`; (d) `float128`; (e) `mpfr` 62; (f) `cpp_bin_float` 62. (g) `mpfr` 150.

a global damping coefficient equal to $5 \cdot 10^{-3}$. One can clearly see the differences in simulation results based on different precisions. The `float` precision simulation (blue curve, Fig. 6a) indicates the largest deviation from all other simulations. Higher precisions gradually provide results that are closer to the simulation with the highest precision (boost `mpfr` 150 decimal places).

Since the angle between rods is oscillating rapidly, as can be seen on the inset in Fig. 6, the raw data was smoothed using the Savitzky–Golay filter [79] for better visibility. The filter removes most of the noise, and there are no visible differences between `cpp_bin_float`, `mpfr` 62 and `mpfr` 150. The three curves are basically overlapping each other.

5.4. Effect of time step Δt

The time step Δt is one of the most important parameters influencing the simulation results. Hence, simulations with time step

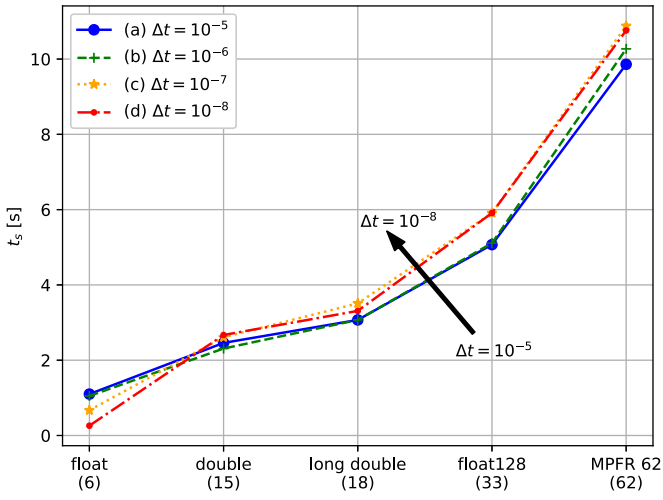

Fig. 7. Correlation duration t_s as a function of different precisions and a time step Δt. Each curve is compared to the `mpfr` 150 simulation using the same Δt; (a) $\Delta t = 10^{-5}$ s; (b) $\Delta t = 10^{-6}$ s; (c) $\Delta t = 10^{-7}$ s; (d) $\Delta t = 10^{-8}$ s.

values of 10^{-5} s, 10^{-6} s, 10^{-7} s, and 10^{-8} s were performed with different precisions. The values for the correlation duration were recorded and plotted in Fig. 7 for all precisions and time steps considered. It can be clearly seen that the correlation duration increases with increasing precision. This highlights once more that higher precision is required for higher confidence. It can also be seen that generally the correlation duration increases with decreasing time step. This is due the fact that a smaller time step results in a smaller integration error (e.g. the leapfrog integration scheme used in YADE has the error proportional to Δt^2 per iteration). This tendency is also marked on the figure with a black arrow pointing in the direction of the smaller time step Δt. The opposite result is observed for `float`. This is because 6 decimal places are not enough to work with $\Delta t = 10^{-8}$ s. It shall be noted that the implementation of a more precise time integration scheme is out of the scope of the current paper. In the current symplectic leapfrog integration scheme, as implemented in the present version of `NewtonIntegrator`, the positions and velocities are leapfrogging over each other. There is no jump-start option which would allow to start the simulation with both position and velocity defined at $t = 0$. This means that the initial velocities are declared at $t = -\frac{\Delta t}{2}$ and initial positions at $t = 0$ or initial velocities are declared at $t = 0$ and initial positions at $t = \frac{\Delta t}{2}$. Which of these two it is, is only a formal choice,[34] since both interpretations are valid. Therefore a more detailed comparison between the time steps is not carried out as the initial conditions for each simulation with a different Δt differ slightly, i.e. the starting velocity declared in the script is interpreted as being defined at $t = -\frac{10^{-5}}{2}$ or at $t = -\frac{10^{-8}}{2}$, thus resulting in slightly different simulations[36].

5.5. Energy conservation

In the following, the total energy in the system is analyzed for different precisions and time step values of 10^{-5} s, 10^{-6} s, 10^{-7} s, and 10^{-8} s. No numerical damping is considered. The total energy in the system is calculated as the sum of the elastic energy in the interactions and the kinetic and potential energy of the mass points (i.e. spheres). As already pointed out in the previous section, the symplectic leapfrog integration scheme is used. This means that velocities and positions are not known at the same time. Hence, the velocities needed for an accurate calculation of the kinetic energy are taken as an average of the velocities from the current and the next iteration. All numerical results are compared to the reference solution which was calculated using 150 decimal places, similar as in the previous sections.

Fig. 8 shows two typical results obtained from the study using a time step of $\Delta t = 10^{-6}$ s. The results obtained with the other time steps have a similar trend and are not included for brevity. The top graphs show the evolution of the energy balance where each energy component is divided by the total reference energy to give an energy ratio. The total reference energy is calculated using 150 decimal places. The total energy ratio should be equal to 1 throughout the simulation. The bottom graphs show the absolute error in total energy calculated as the absolute difference between the total energy given by a specific precision and the constant total energy calculated using 150 decimal places.

The results obtained using `float` are depicted in Fig. 8a. It can be seen that the absolute error in total energy starts to increase drastically after about 4 s. This is much later than the correlation duration. From the energy plot it can also be seen that energy is continuously added to the system from this time onward. This makes the simulation not only incorrect but also unstable. A different observation can be drawn from Fig. 8b where the results of a typical simulation with `float128` are shown. It can be seen that the energy balance is stable and the absolute error is several order of magnitudes smaller. In addition, the absolute error does not have an increasing trend. Instead, it bounces around, i.e. it increases initially and decreases thereafter, and never goes above a certain threshold value. This is also a reflection of the symplectic leapfrog integration scheme. It should be noted that the results for `double`, `long double`, `mpfr` 62 and `cpp_bin_float` 62 are very similar to Fig. 8b and, hence, not shown for brevity.

Fig. 9 summarizes the results for all precisions and all time steps. As noted previously, a detailed comparison between different time steps does not make sense because of the different initial velocities. Nevertheless, a qualitative comparison is valid. It can be seen that the maximum absolute error in energy balance for `float` is many orders of magnitudes larger than for the other precisions (note that the vertical axis uses logarithmic scale). The data also indicates that the error is almost constant for all other precisions. This clearly highlights the effectiveness and reliability of the symplectic leapfrog integration scheme implemented in YADE. However the error is many orders of magnitude larger than the ULP error of the higher precision types. For example to achieve maximum absolute error of 10^{-30} [J] for `float128`, further decreasing the time step is not practical. Different approaches, such as higher order symplectic methods, have to be employed [68,69][36]. Like in previous section, a smaller time step results in a smaller absolute error (this tendency is indicated by the black arrow), except for `float` where 6 decimal places are not enough to work with the smaller time steps.

6. Conclusions and future perspectives

The obtained results show that using high precision has a pronounced influence on the simulation results and the calculation speed. Higher precisions provide more accurate results and reduce numerical errors which in some fields can be beneficial. It can also be concluded that high precision is essential for research of highly chaotic systems (Fig. 5). Nevertheless, increasing the number of decimal places in the code leads to a higher CPU load and raises the calculation times (Fig. 3, Table 3).

Updating an existing software with a large codebase to bring the flexibility of arbitrary precision can be a challenging and error-prone process which might require drastic refactoring. A good test

[34] See also: https://gitlab.com/yade-dev/trunk/-/merge_requests/555#note_462560944 and `checks/checkGravity.py`.

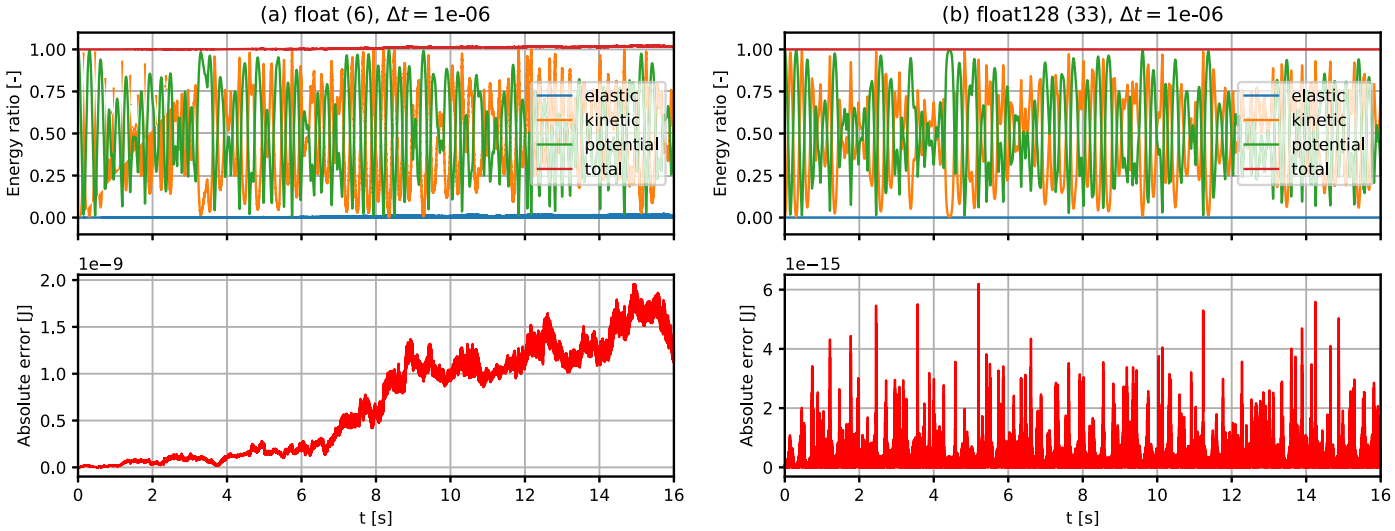

Fig. 8. Evolution of energy balance plotted as energy ratios and corresponding absolute error compared to the `mpfr` 150 simulation for precisions (a) `float` and (b) `float128`.

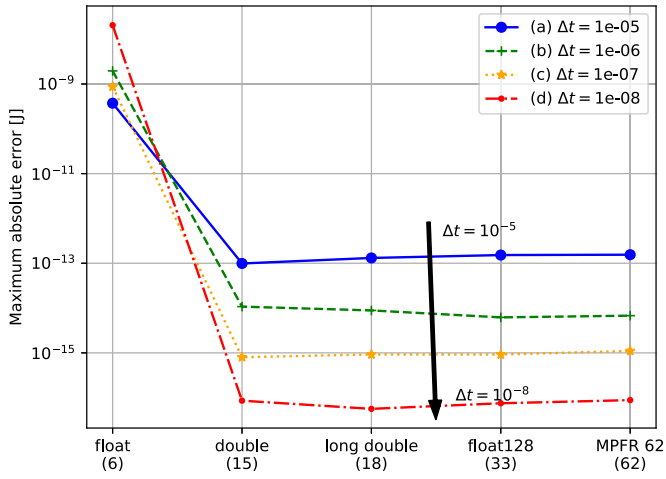

Fig. 9. Maximum absolute error in energy balance during the first 16 s as a function of different precisions and for different time steps Δt. Each curve is compared to the `mpfr` 150 simulation using the same Δt; (a) $\Delta t = 10^{-5}$ s; (b) $\Delta t = 10^{-6}$ s; (c) $\Delta t = 10^{-7}$ s; (d) $\Delta t = 10^{-8}$ s. Note that the axis for the maximum absolute error is in log scale.

coverage of the code (unit and integration tests: the Continuous Integration pipeline in YADE) is highly recommended before beginning of such refactoring to ensure code integrity [10]. Also the employment of AddressSanitizer [83] is highly desirable to prevent heavy memory errors such as heap corruptions, memory leaks, and out-of-bounds accesses.

Simulations with the triple pendulum show that the results are starting to be different after a few seconds of the simulation time because of different precisions. The higher the precision the longer the results remain true to the highest precision tested. The same effect, within each precision (Fig. 7 and Fig. 9), occurs when using smaller time steps Δt, because it has a smaller time integration error per iteration. Applying damping can significantly smooth this effect (Fig. 6).

The new high-precision functionality added to YADE does not negatively affect the existing computational performance (i.e. simulations with `double` precision), because the choice of precision is done at compilation time and is dispatched during compilation via the C++ static polymorphism template mechanisms [89,96].

Concluding this work, the main modules of YADE now fully support two aspects of arbitrary precision (see Table 1 and Fig. 1):

1. Selecting the base precision of `Real` from Table 2 (which is an alias for `RealHP<1>`).
2. Using `RealHP<N>` in the critical C++ sections of the numerical algorithms (Section 2.4)[1].

These new arbitrary precision capabilities can be used in several different ways in the management of numerical error [5]:

3. To periodically test YADE computation algorithms to check if some of them are becoming numerically sensitive.
4. To determine how many digits in the obtained intermediate and final results are reliable.
5. To debug the code in order to find the lines of code which produce numerical errors, using the method described in details in chapter 14 of [51].
6. To fix numerical errors that were found by changing the critical part of the computation to use a higher precision type like `RealHP<2>` or `RealHP<4>` as suggested in [50].

The current research focus is to:

7. Add quantum dynamics calculations to YADE using the time integration algorithm which can have the error smaller than the numerical ULP error of any of the high-precision types: the Kosloff method [91,52,80,92] based on the rapidly converging Chebychev polynomial expansion of an exponential propagator.[35]
8. Add unit systems support, because there are also software errors related to unit systems, for example in 10 November 1999 the NASA's Mars Climate Orbiter was lost in space because of mixing SI and imperial units [87,74,43].

Possible future research avenues, opened by the present work, include:

9. Add more precise time integration algorithms. The problems mentioned in Section 5.4 are well known. Albeit symplectic

[35] Since that algorithm is Taylor–free, there is no meaning to the term "order of the method" [92,80].

integrators are particularly good [75,90], a better time integration method with smaller Δt will be able to fully take advantage of the new high-precision capabilities (Section 5.5 and Fig. 9). There are three possible research directions:

(a) Use the Boost Odeint library [24] with higher order methods[1],[36]

(b) Use the work on time integrators by Omelyan et al. [68,69]. These approaches suggest that it is potentially possible to decrease the run time more than 50-fold with the same computation effort upon switching to `long double`, `float128` or higher types. The algorithms focus on reducing the truncation errors and eliminating errors introduced by the computation of forces. Such a smaller error allows to use a larger time step which will more than compensate the speed loss due to high-precision calculations. Of course, the standard considerations for the time step [17,70] would have to be re-derived.

(c) Investigate whether the exponential propagator approach presented in [80] or in [72,82] could be used in YADE as a general solution for ODEs, similarly to [69,42,19,75,65,30], regardless if that is a classical or a quantum dynamics system.

10. Enhance all auxiliary modules of YADE for the use of high precision.[37]
11. Use the interval computation approach to reduce problems with numerical reproducibility in parallel computations by using `boost::multiprecision::mpfi_float` as the backend for `RealHP<N>` type [76,66].
12. Use different rounding modes to run the same computation for more detailed testing of numerical algorithms [51].

Overall, based on the presented work, the architecture of YADE now offers an opportunity to adjust its precision according to the needs of its user. A wide operating system support and simple installation procedure enable forming multidisciplinary teams for computational physics simulations in the Unified Science Environment (USE) [64]. This will expand the spectrum of tasks that can be solved, improve the results and reduce numerical errors. Of course, this option not only complicates the architecture and the source code, but also imposes a restriction on the choice of a programming language.

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.

Acknowledgements

The authors would like to acknowledge the partial support of the COST action CA18222 “Attosecond Chemistry” and the Australian Research Council (DP190102407). The support of the Institute for Mineral Processing Machines and Recycling Systems Technology at TU Bergakademie Freiberg for providing access to computing facilities is also acknowledged. In addition, the authors thank the Department of Building Structures and Material Engineering, Faculty of Civil and Environmental Engineering, at Gdańsk University of Technology for hosting the gitlab Continuous Integration (CI) pipeline for YADE. The authors would also like to thank the two anonymous reviewers whose suggestions helped improve this manuscript.

[36] See also: Boost Odeint, #171, !555 and !557 where `RungeKuttaCashKarp54Integrator` allowed to set error tolerance of 147 correct decimal places if `mpfr` 150 is used.

[37] For the full up-to-date list of supported modules see: http://yade-dem.org/doc/HighPrecisionReal.html#supported-modules.

References

[1] A. Abad, R. Barrio, A. Dena, Phys. Rev. E 84 (2011), https://doi.org/10.1103/PhysRevE.84.016701.
[2] A. Alexandrescu, Modern C++ Design: Generic Programming and Design Patterns Applied, Addison-Wesley Longman Publishing Co., Inc., USA, 2001.
[3] J. Awrejcewicz, B. Supel, C. Lamarque, G. Kudra, G. Wasilewski, P. Olejnik, Int. J. Bifurc. Chaos 18 (2008) 2883–2915, https://doi.org/10.1142/S0218127408022159.
[4] D. Bailey, Comput. Sci. Eng. 2 (2000) 24–28, https://doi.org/10.1109/5992.814653.
[5] D. Bailey, Comput. Sci. Eng. 7 (2005) 54–61, https://doi.org/10.1109/MCSE.2005.52.
[6] D. Bailey, R. Barrio, J. Borwein, Appl. Math. Comput. 218 (2012) 10106–10121, https://doi.org/10.1016/j.amc.2012.03.087.
[7] D.H. Bailey, Comput. Sci. Eng. 7 (2005) 54–61.
[8] D.H. Bailey, A.M. Frolov, J. Phys. B, At. Mol. Opt. Phys. 35 (2002) 4287–4298, https://doi.org/10.1088/0953-4075/35/20/314.
[9] D.H. Bailey, K. Jeyabalan, X.S. Li, Exp. Math. 14 (2005) 317–329, https://doi.org/10.1080/10586458.2005.10128931.
[10] M. Bellotti, Kill It with Fire. Manage Aging Computer Systems (and Future-Proof Modern Ones), William Pollock, 2021.
[11] M. Ben-Ari, SIGCSE Bull. 33 (2001) 58–59, https://doi.org/10.1145/571922.571958.
[12] M. Blair, S. Obenski, P. Bridickas, Patriot missile defense software problem led to system failure at Dhahran, Saudi Arabia, Technical Report, United States General Accounting Office, 1992, https://www.gao.gov/assets/220/215614.pdf.
[13] B.M. Boghosian, P.V. Coveney, H. Wang, Adv. Theory Simul. 2 (2019) 1900125, https://doi.org/10.1002/adts.201900125.
[14] C. Boon, G. Houlsby, S. Utili, Powder Technol. 248 (2013) 94–102, https://doi.org/10.1016/j.powtec.2012.12.040.
[15] F. Bourrier, F. Kneib, B. Chareyre, T. Fourcaud, Ecol. Eng. (2013), https://doi.org/10.1016/j.ecoleng.2013.05.002.
[16] D. Broadhurst, Eur. Phys. J. C 8 (1999) 311–333, https://doi.org/10.1007/s100529900935.
[17] S.J. Burns, K.J. Hanley, Int. J. Numer. Methods Eng. 110 (2016) 186–200, https://doi.org/10.1002/nme.5361.
[18] R.E. Caflisch, Phys. D: Nonlinear Phenom. 67 (1993) 1–18, https://doi.org/10.1016/0167-2789(93)90195-7.
[19] M. Caliari, A. Ostermann, Appl. Numer. Math. 59 (2009) 568–581, https://doi.org/10.1016/j.apnum.2008.03.021.
[20] R.A. Caulk, E. Catalano, B. Chareyre, Comput. Phys. Commun. 248 (2020) 106991, https://doi.org/10.1016/j.cpc.2019.106991.
[21] R.A. Caulk, B. Chareyre, in: Thermal Process Engineering: Proceedings of DEM8, 2019, https://mercurylab.co.uk/dem8/wp-content/uploads/sites/4/2019/07/217.pdf.
[22] P. Cundall, O. Strack, Geotechnique (1979) 47–65, https://doi.org/10.1680/geot.1979.29.1.47.
[23] L. Dagum, R. Menon, IEEE Comput. Sci. Eng. 5 (1998) 46–55.
[24] B. Dawes, et al., Boost C++ libraries, https://www.boost.org/, 2020.
[25] F. De Martini, E. Santamato, Int. J. Theor. Phys. 53 (2013) 3308–3322, https://doi.org/10.1007/s10773-013-1651-y.
[26] Debian, Bug tracker, https://bugs.debian.org, bug numbers: openmpi #961108, PETSc #953116,#961977, mumps #961976, scotch #961184, scalapack #961186, metis #961183, hypre #961645, slepc #972069, superlu #989497, suitesparse #989550, arpack #972068; visited on 29.07.2021.
[27] F.V. Donzé, Y. Klinger, V. Bonilla-Sierra, J. Duriez, L. Jiao, L. Scholtès, Tectonophysics 805 (2021) 228779, https://doi.org/10.1016/j.tecto.2021.228779.
[28] A. Effeindzourou, B. Chareyre, K. Thoeni, A. Giacomini, F. Kneib, Geotext. Geomembr. 44 (2016) 143–156, https://doi.org/10.1016/j.geotexmem.2015.07.015.
[29] J. Eliáš, Powder Technol. 264 (2014) 458–465, https://doi.org/10.1016/j.powtec.2014.05.052.
[30] H. Fahs, Appl. Math. Model. 36 (2012) 5466–5481, https://doi.org/10.1016/j.apm.2011.12.055.
[31] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, P. Zimmermann, ACM Trans. Math. Softw. 33 (2007) 13, https://doi.org/10.1145/1236463.1236468.
[32] G. Frenning, Comput. Methods Appl. Mech. Eng. 197 (2008) 4266–4272, https://doi.org/10.1016/j.cma.2008.05.002.
[33] A.M. Frolov, D.H. Bailey, J. Phys. B, At. Mol. Opt. Phys. 37 (2004) 955, https://doi.org/10.1088/0953-4075/37/4/c02.
[34] A. Gladkyy, M. Kuna, Granul. Matter 19 (2017) 41, https://doi.org/10.1007/s10035-017-0731-8.
[35] A. Gladkyy, R. Schwarze, Granul. Matter (2014) 1–10, https://doi.org/10.1007/s10035-014-0527-z.
[36] D. Goldberg, ACM Comput. Surv. 23 (1991) 5–48, https://doi.org/10.1145/103162.103163.

[37] G. Guennebaud, B. Jacob, et al., Eigen v3, http://eigen.tuxfamily.org, 2010.
[38] N. Guo, J. Zhao, Int. J. Numer. Methods Eng. 99 (2014) 789–818, https://doi.org/10.1002/nme.4702.
[39] P.H. Hauschildt, E. Baron, J. Comput. Appl. Math. 109 (1999) 41–63, https://doi.org/10.1016/S0377-0427(99)00153-3.
[40] M. Haustein, A. Gladkyy, R. Schwarze, SoftwareX 6 (2017) 118–123, https://doi.org/10.1016/j.softx.2017.05.001.
[41] Y. He, C.H.Q. Ding, J. Supercomput. 18 (2001) 259–277, https://doi.org/10.1023/A:1008153532043.
[42] M. Hochbruck, C. Lubich, BIT Numer. Math. 39 (1999) 620–645, https://doi.org/10.1023/A:1022335122807.
[43] T. Huckle, T. Neckel, Bits and Bugs: A Scientific and Historical Review of Software Failures in Computational Science, Society for Industrial and Applied Mathematics, USA, 2019.
[44] K. Isupov, Data Brief 30 (2020) 105506, https://doi.org/10.1016/j.dib.2020.105506.
[45] P. Jasik, J. Franz, D. Kędziera, T. Kilich, J. Kozicki, J.E. Sienkiewicz, J. Chem. Phys. 154 (2021) 164301, https://doi.org/10.1063/5.0046060.
[46] P. Jasik, J. Kozicki, T. Kilich, J.E. Sienkiewicz, N.E. Henriksen, Phys. Chem. Chem. Phys. 20 (2018) 18663–18670, https://doi.org/10.1039/c8cp02551g.
[47] J.F. Jerier, B. Hathong, V. Richefeu, B. Chareyre, D. Imbault, F.V. Donze, P. Doremus, Powder Technol. 208 (2011) 537–541, https://doi.org/10.1016/j.powtec.2010.08.056.
[48] M. Joldes, V. Popescu, W. Tucker, ACM SIGARCH Comput. Archit. News 42 (2014) 63–68, https://doi.org/10.1145/2693714.2693726.
[49] J.M. Jézéquel, B. Meyer, Computer 30 (1997) 129–130, https://doi.org/10.1109/2.562936, http://se.ethz.ch/~meyer/publications/computer/ariane.pdf.
[50] W. Kahan, On the cost of floating-point computation without extra-precise arithmetic, https://people.eecs.berkeley.edu/~wkahan/Qdrtcs.pdf, 2004.
[51] W. Kahan, How futile are mindless assessments of roundoff in floating-point computation?, https://people.eecs.berkeley.edu/~wkahan/Mindless.pdf, 2006.
[52] R. Kosloff, Quantum Molecular Dynamics on Grids, Department of Physical Chemistry and the Fritz Haber Research Center, 1997, https://scholars.huji.ac.il/sites/default/files/ronniekosloff/files/wayat.pdf.
[53] J. Kozicki, F. Donzé, Comput. Methods Appl. Mech. Eng. 197 (2008) 4429–4443, https://doi.org/10.1016/j.cma.2008.05.023.
[54] M. Krzaczek, M. Nitka, J. Kozicki, J. Tejchman, Acta Geotech. 15 (2019) 297–324, https://doi.org/10.1007/s11440-019-00799-6.
[55] M. Krzaczek, M. Nitka, J. Tejchman, Int. J. Numer. Anal. Methods Geomech. 45 (2020) 234–264, https://doi.org/10.1002/nag.3160.
[56] G.L. Lann, The Ariane 5 Flight 501 Failure - a Case Study in System Engineering for Computing Systems, Technical Report. Research Report, RR-3079, INRIA, 1996, inria-00073613, https://hal.inria.fr/inria-00073613, 2006.
[57] J. Laskar, M. Gastineau, Nature 459 (2009) 817–819, https://doi.org/10.1038/nature08096.
[58] J.L. Lions, L. Lübeck, J.L. Fauquembergue, G. Kahn, W. Kubbat, S. Levedag, L. Mazzini, D. Merle, C. O'Halloran, Ariane 5 flight 501 failure, Technical Report. Report by the inquiry board, 1996, https://esamultimedia.esa.int/docs/esa-x-1819eng.pdf.
[59] E. Loh, G.W. Walster, Reliab. Comput. 8 (2002) 245–248, https://doi.org/10.1023/A:1015569431383.
[60] F. Lominé, L. Scholtès, L. Sibille, P. Poullain, Int. J. Numer. Anal. Methods Geomech. 37 (2011) 577–596, https://doi.org/10.1002/nag.1109.
[61] R. Lougee-Heimer, IBM J. Res. Dev. 47 (2003) 57–66, https://doi.org/10.1147/rd.471.0057.
[62] M. Lu, B. He, Q. Luo, Supporting extended precision on graphics processors, 19–26, https://doi.org/10.1145/1869389.1869392, 2010.
[63] R. Maurin, J. Chauchat, B. Chareyre, P. Frey, Phys. Fluids 27 (2015) 113302, https://doi.org/10.1063/1.4935703.
[64] C. McCurdy, H.D. Simon, W.T. Kramer, R.F. Lucas, W.E. Johnston, D.H. Bailey, Comput. Phys. Commun. 147 (2002) 34–39, https://doi.org/10.1016/S0010-4655(02)00200-X.
[65] R.I. McLachlan, G.R.W. Quispel, N. Robidoux, Phys. Rev. Lett. 81 (1998) 2399–2403, https://doi.org/10.1103/PhysRevLett.81.2399.
[66] J.P. Merlet, Jacobian, Manipulability, Condition Number and Accuracy of Parallel Robots, Springer, Berlin Heidelberg, 2007, pp. 175–184.
[67] S. Meyers, Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14, 1st ed., O'Reilly Media, Inc., 2014.
[68] I. Omelyan, I. Mryglod, R. Folk, Comput. Phys. Commun. 146 (2002) 188–202, https://doi.org/10.1016/S0010-4655(02)00451-4.
[69] I.P. Omelyan, Phys. Rev. E 74 (2006), https://doi.org/10.1103/PhysRevE.74.036703.
[70] I.P. Omelyan, A. Kovalenko, J. Chem. Theory Comput. 8 (2011) 6–16, https://doi.org/10.1021/ct200157x.
[71] OpenMP Architecture Review Board, OpenMP application program interface version 5.1, https://www.openmp.org/specifications/, 2021.
[72] Y. Orimo, T. Sato, A. Scrinzi, K.L. Ishikawa, Phys. Rev. A 97 (2018), https://doi.org/10.1103/PhysRevA.97.023423.
[73] K. Pachucki, M. Puchalski, Phys. Rev. A 71 (2005), https://doi.org/10.1103/PhysRevA.71.032514.
[74] I. Peterson, Fatal Defect: Chasing Killer Computer Bugs, Mckay, David, 1996.
[75] G.R.W. Quispel, G.S. Turner, J. Phys. A, Math. Gen. 29 (1996) L341–L349, https://doi.org/10.1088/0305-4470/29/13/006.
[76] N. Revol, P. Theveny, IEEE Trans. Comput. 63 (2014) 1915–1924, https://doi.org/10.1109/TC.2014.2322593.
[77] M.J. Saltzman, COIN-OR: An Open-Source Library for Optimization, Springer US, 2002.
[78] E. Santamato, F. De Martini, Found. Phys. 45 (2015) 858–873, https://doi.org/10.1007/s10701-015-9912-7.
[79] A. Savitzky, M.J.E. Golay, Anal. Chem. 36 (1964) 1627–1639.
[80] I. Schaefer, H. Tal-Ezer, R. Kosloff, J. Comput. Phys. 343 (2017) 368–413, https://doi.org/10.1016/j.jcp.2017.04.017.
[81] L. Scholtès, F.V. Donzé, J. Mech. Phys. Solids 61 (2013) 352–369, https://doi.org/10.1016/j.jmps.2012.10.005.
[82] A. Scrinzi, Comput. Phys. Commun. 270 (2022) 108146, arXiv:2101.08171.
[83] K. Serebryany, D. Bruening, A. Potapenko, D. Vyukov, AddressSanitizer: A Fast Address Sanity Checker, 2012, p. 28.
[84] M. Siłkowski, K. Pachucki, J. Chem. Phys. 152 (2020) 174308, https://doi.org/10.1063/5.0008086.
[85] M.A. Stadtherr, High performance computing: are we just getting wrong answers faster?, https://www3.nd.edu/~markst/cast-award-speech.pdf, 1998.
[86] T.P. Stefański, IEEE Antennas Propag. Mag. 55 (2013) 344–353.
[87] A.G. Stephenson, L.S. LaPiana, D.R. Mulville, P.J. Rutledge, F.H. Bauer, D. Folta, G.A. Dukeman, R. Sackheim, P. Norvig, Mars Climate Orbiter, Phase I Report, Technical Report, Mishap Investigation Board, 1999, http://sunnyday.mit.edu/accidents/MCO_report.pdf.
[88] S.M. Stigler, Stat. Sci. 4 (1989) 73–79, https://doi.org/10.1214/ss/1177012580.
[89] B. Stroustrup, Programming: Principles and Practice Using C++ (2nd Edition), 2nd ed., Addison-Wesley Professional, 2014.
[90] G.J. Sussman, J. Wisdom, Science 257 (1992) 56–62, https://doi.org/10.1126/science.257.5066.56.
[91] H. Tal-Ezer, R. Kosloff, J. Chem. Phys. 81 (1984) 3967–3971.
[92] H. Tal-Ezer, R. Kosloff, I. Schaefer, J. Sci. Comput. 53 (2012) 211–221, https://doi.org/10.1007/s10915-012-9583-x.
[93] The CGAL Project, CGAL User and Reference Manual. 5.0.2 ed., CGAL Editorial Board, 2020, https://doc.cgal.org/5.0.2/Manual/packages.html.
[94] Yade publications, in: The Yade Project, https://www.yade-dem.org/doc/publications.html, 2020.
[95] K. Thoeni, C. Lambert, A. Giacomini, S. Sloan, Comput. Geotech. 49 (2013) 158–169, https://doi.org/10.1016/j.compgeo.2012.10.014.
[96] D. Vandevoorde, N.M. Josuttis, D. Gregor, C++ Templates: The Complete Guide (2nd Edition), 2nd ed., Addison-Wesley Professional, 2017.
[97] P. Virtanen, R. Gommers, T.E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S.J. van der Walt, M. Brett, J. Wilson, K. Jarrod Millman, N. Mayorov, A.R.J. Nelson, E. Jones, R. Kern, E. Larson, C. Carey, İ. Polat, Y. Feng, E.W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E.A. Quintero, C.R. Harris, A.M. Archibald, A.H. Ribeiro, F. Pedregosa, P. van Mulbregt, Contributors S, et al., Nat. Methods 17 (2020) 261–272, https://doi.org/10.1038/s41592-019-0686-2.
[98] V. Šmilauer, et al., Yade Documentation 2nd ed., The Yade Project, 2015, http://yade-dem.org/doc/.
[99] Z.C. Yan, G.W.F. Drake, Phys. Rev. Lett. 91 (2003), https://doi.org/10.1103/PhysRevLett.91.113004.
[100] T. Zhang, Z.C. Yan, G.W.F. Drake, Phys. Rev. Lett. 77 (1996) 1715–1718, https://doi.org/10.1103/PhysRevLett.77.1715.