Comparison of Numerical Performance of *Mathematica* 10.4 and Maple 2015.2

Summary

While Maple claims support for a significant subset of the numerical computations performed by *Mathematica*, in most cases, much faster methods have been implemented in *Mathematica*.

Over a test suite of 585 tasks, covering different operations on different types of numerical data, *Mathematica* was faster in 574 cases. The median difference measured found *Mathematica* to be more than 31 times faster than Maple.

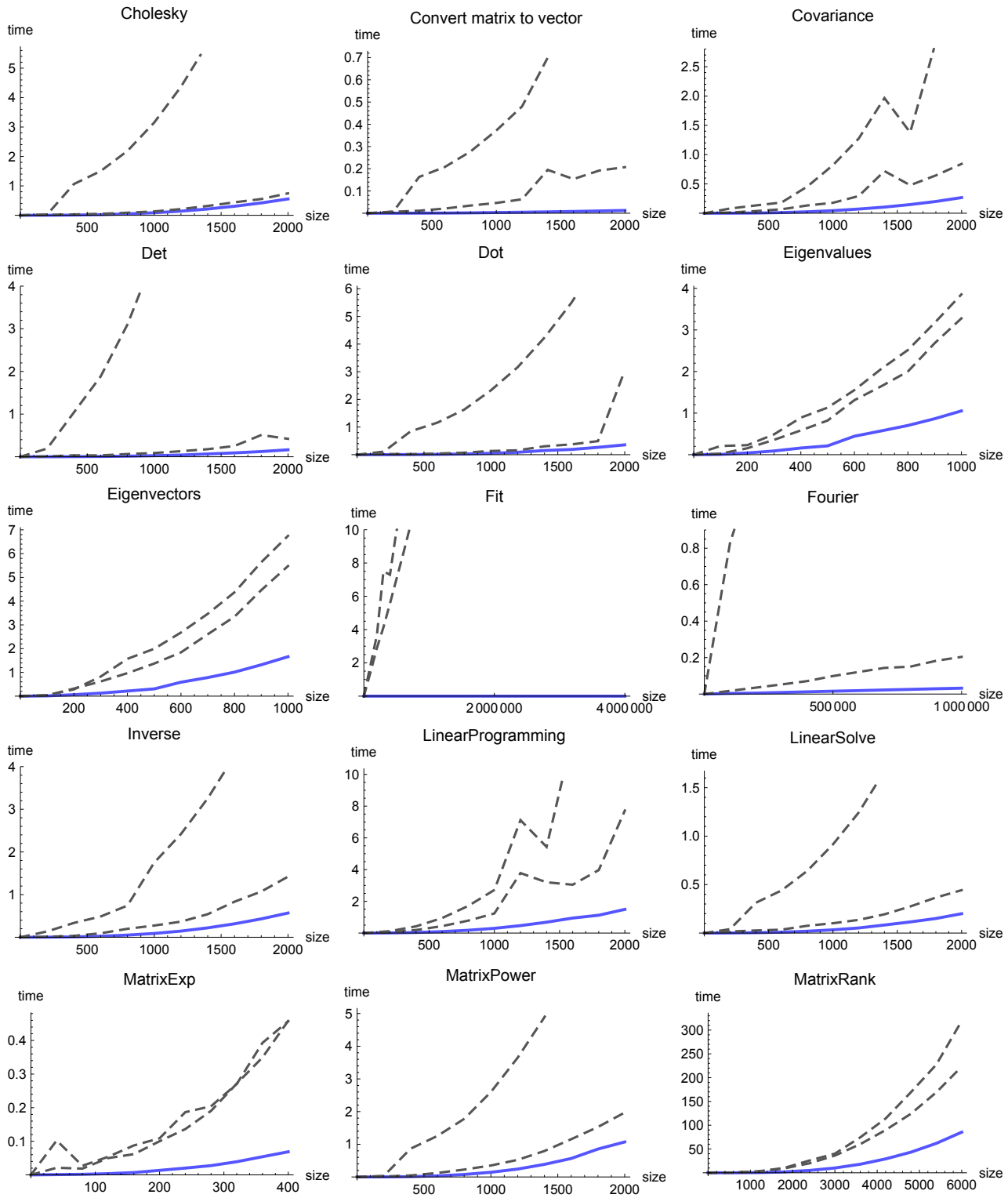
Category	Tests	Median *Mathematica* speed
Real data operations	20	23 times faster
Real data operations (Manual type override)	20	3 times faster
Complex number data operations	16	5 times faster
Complex number data operations (Manual type override)	16	5 times faster
Sparse real data operations	7	50 000 times faster
Integer data operations	14	40 times faster
Integer data operations (Manual type override)	13	36 times faster
Extended precision data operations (50 digits)	16	9 times faster
Extended precision data operations (1000 digits)	16	4 times faster
Random number generation	7	7 times faster
Elementary & special functions	90	2586 times faster
Elementary & special functions (Manual type override)	66	15 times faster
Complex elementary & special functions	90	53 times faster
Complex elementary & special functions (Manual type override)	90	75 times faster
Elementary & special functions (50 digits)	90	30 times faster
High precision function evaluation	9	31 times faster
Exact functions	4	7 times faster
GPU operations	1	3 times faster
Total	585	31 times faster

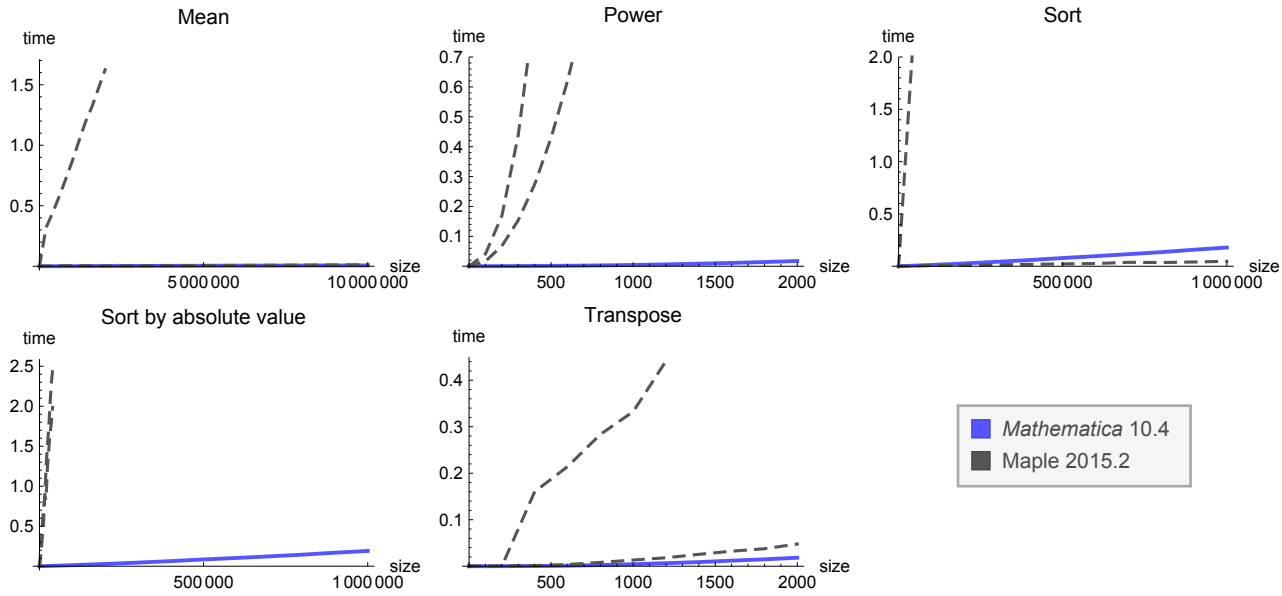
Tests were performed using *Mathematica* 10.4.0 and Maple 2015.2 running on Windows 7 64-bit with 3.07 GHz quad-core Intel i7 processor with 24 GB of RAM. CUDA tests used a Tesla C2050/C2070 GPU with 448 cores and 2.5 GB of RAM.

Machine-Precision Real Linear Algebra

To achieve the very best performance in Maple, you must often use manual type control. This requires declaring items in a dataset to all be of the same number type as it is initialized. The penalty of this approach is that such Maple code will fail if any value is used that does not comply with the type declaration. Since it is not always easy to predict whether operations will potentially yield complex numbers, large numbers, or symbolic results, the default data type for all Maple operations is “anything,” and most user code uses this type. Both manual and default Maple data type timings are shown in these comparisons.

Mathematica operations are 3 times faster than Maple’s inflexible (`float[8]`) data type and 23 times faster than its default type.



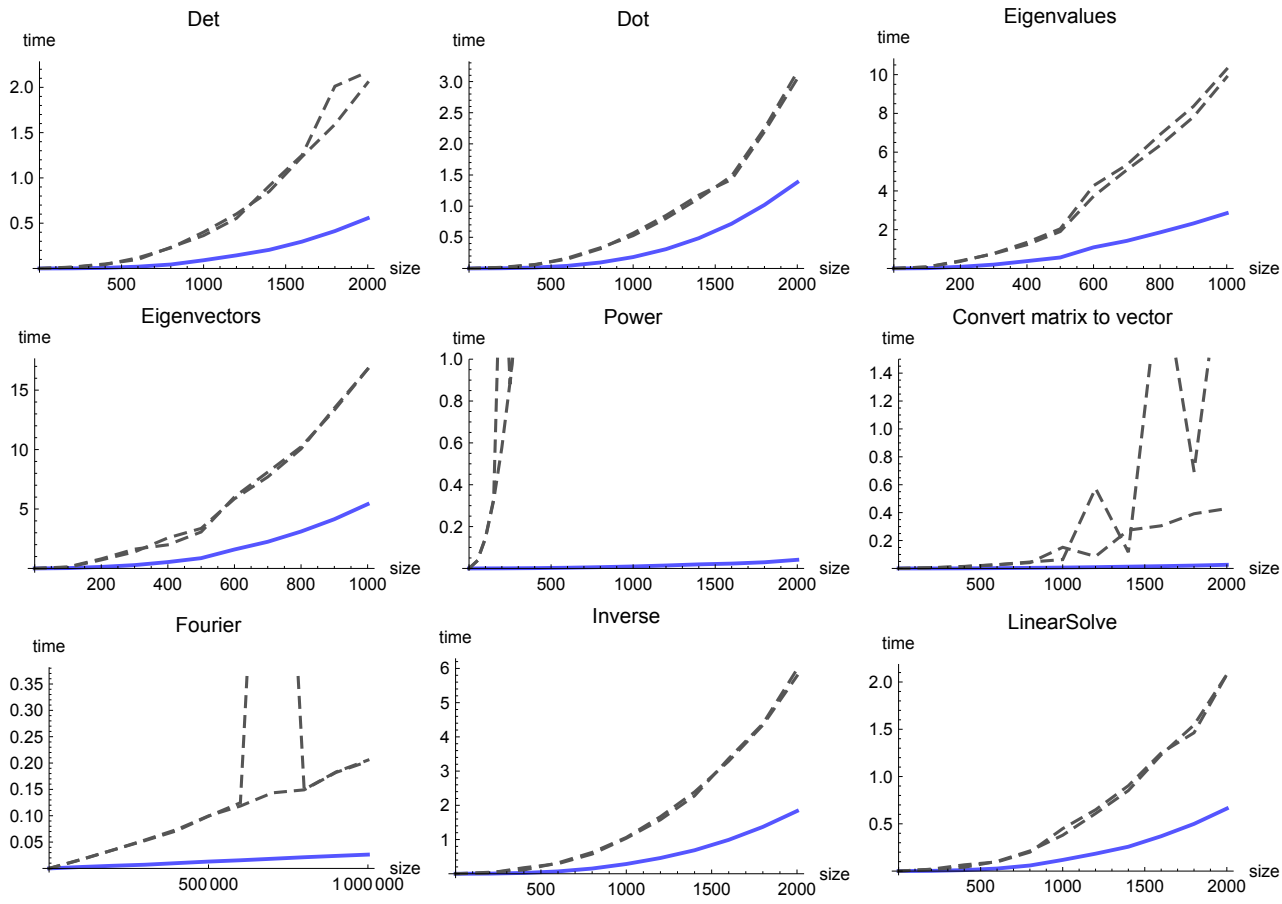


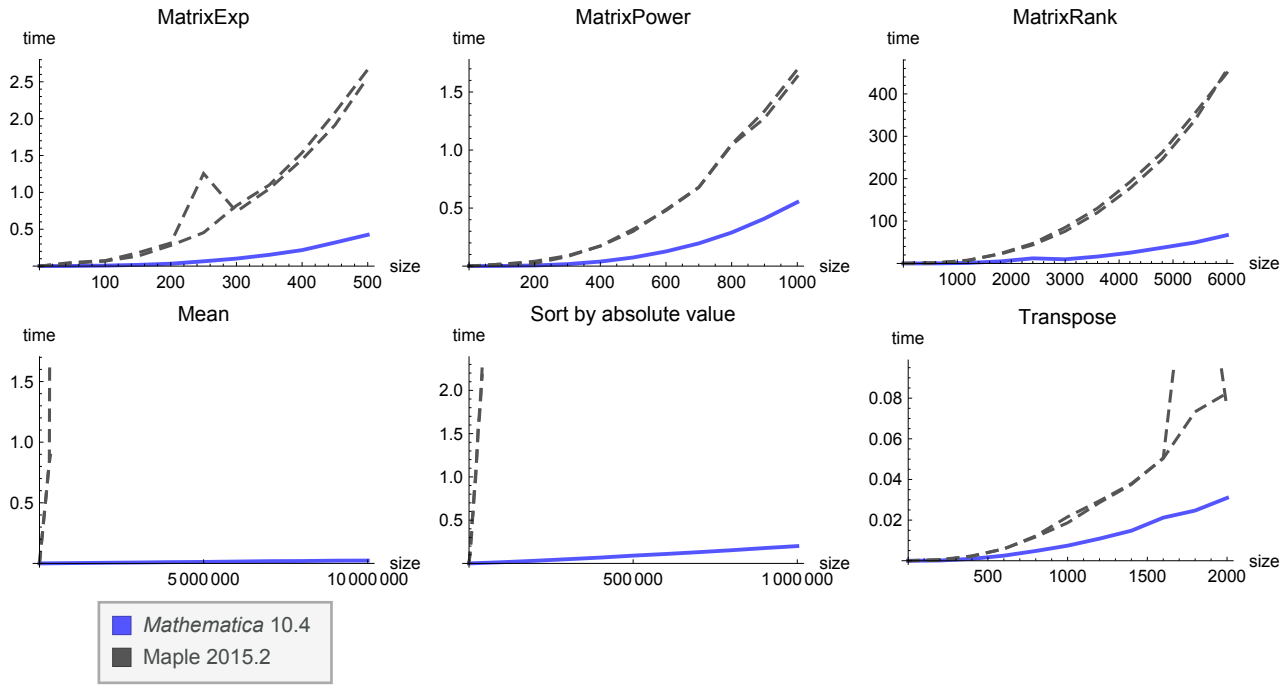
Implementation notes

All tests were performed using the documented commands in each system.

Machine-Precision Complex Linear Algebra

For complex numbers, there is little advantage to manually setting the data type in Maple. *Mathematica* is more than 5 times faster.



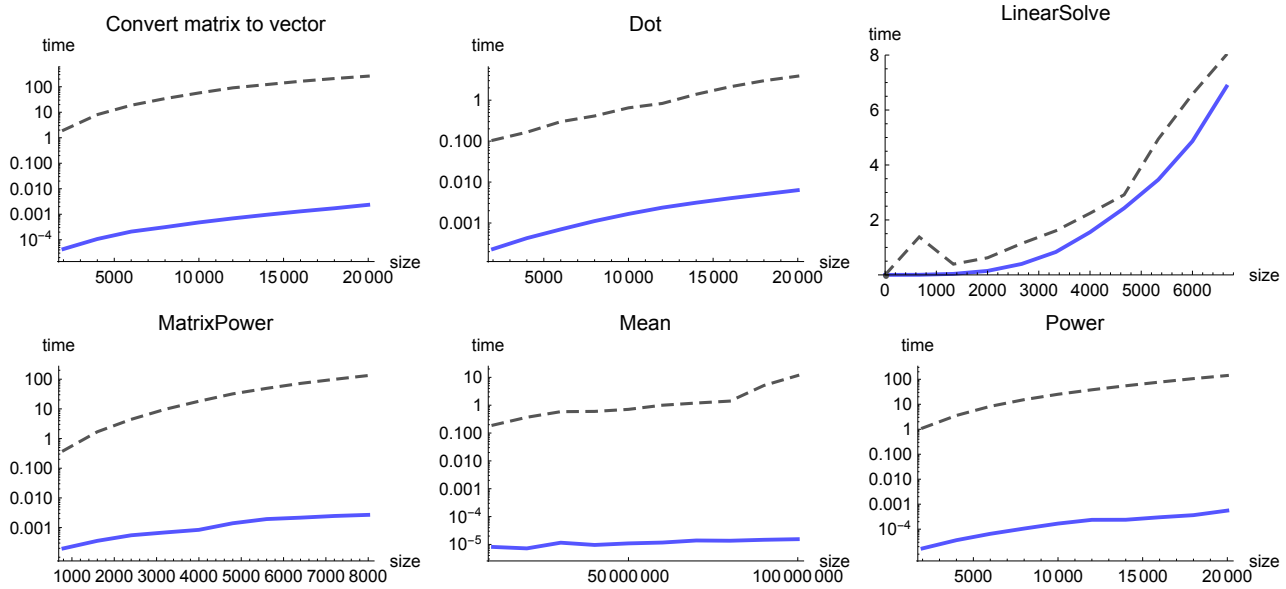


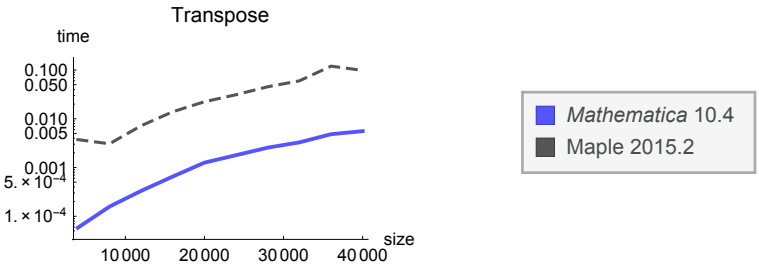
Implementation notes

Functions in Maple's `Statistics` packages work only on real numbers. For this reason, the `Mean` operation has been implemented using Maple's `add` command.
All other tests were performed using the documented command in each system.

Sparse Data

Maple and *Mathematica* both provide sparse data storage; however, Maple appears to have only implemented sparse methods for `LinearSolve` and `Transpose`. During the test, Maple's `LinearSolve` command generates a large number of messages stating, "Warning, solution did not converge within 200 iterations," suggesting that it may not have completed the task for some problem sizes.
Note that, apart from `LinearSolve`, data is shown on a log scale, since the difference in performance is too great to display on a linear scale.





Implementation notes

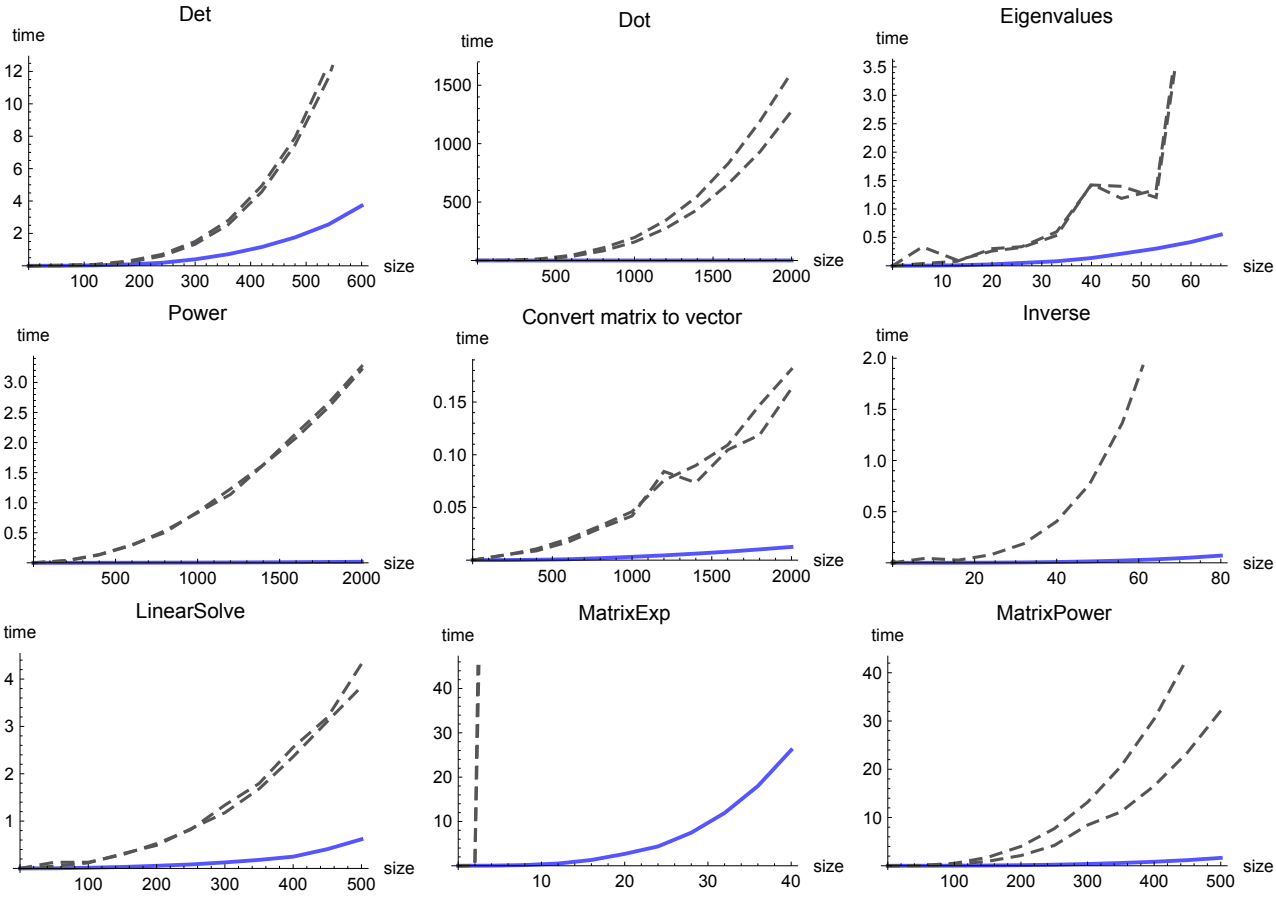
Tests for which neither system has sparse methods have been omitted since these would be a repeat of the dense tests above.

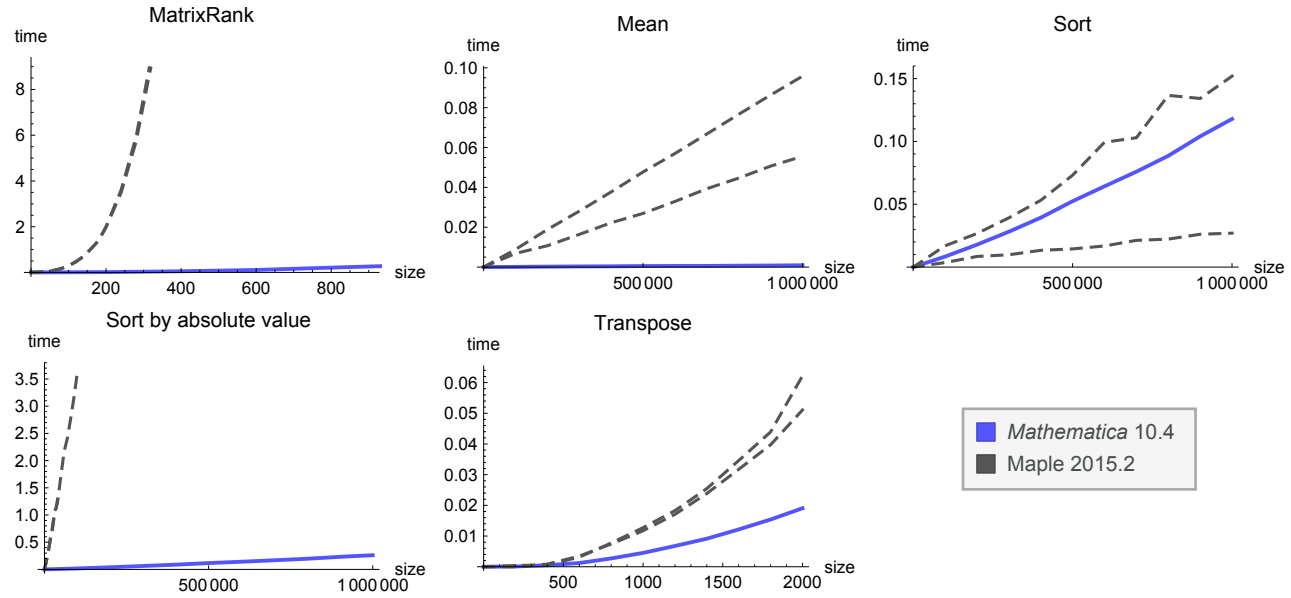
Integer Linear Algebra

Both *Mathematica* and Maple can perform exact integer arithmetic; however, again the Maple implementation does not seem to include many optimized integer algorithms. One of the biggest differences is seen with perhaps the most important matrix operation—multiplication (`Dot`)—where *Mathematica* was more than 3000 times faster.

Maple users can choose between a general data type or `integer[8]`, but there appears to be little benefit in using the more limiting data type.

The median difference measures *Mathematica* to be over 36 times faster.





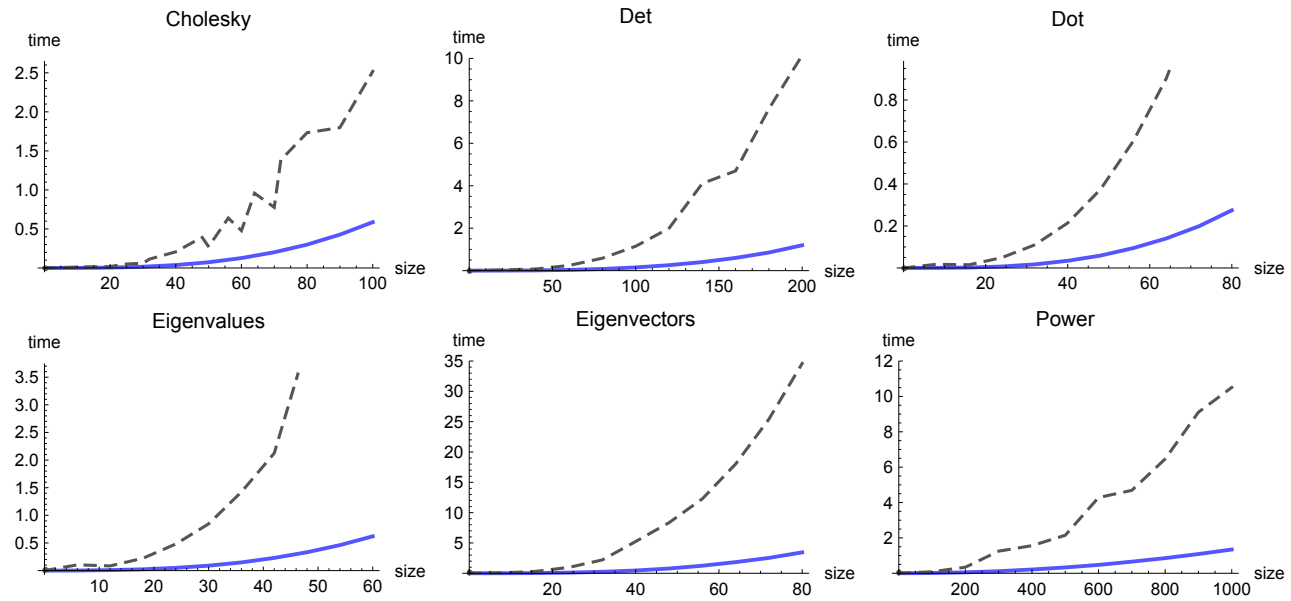
Implementation notes

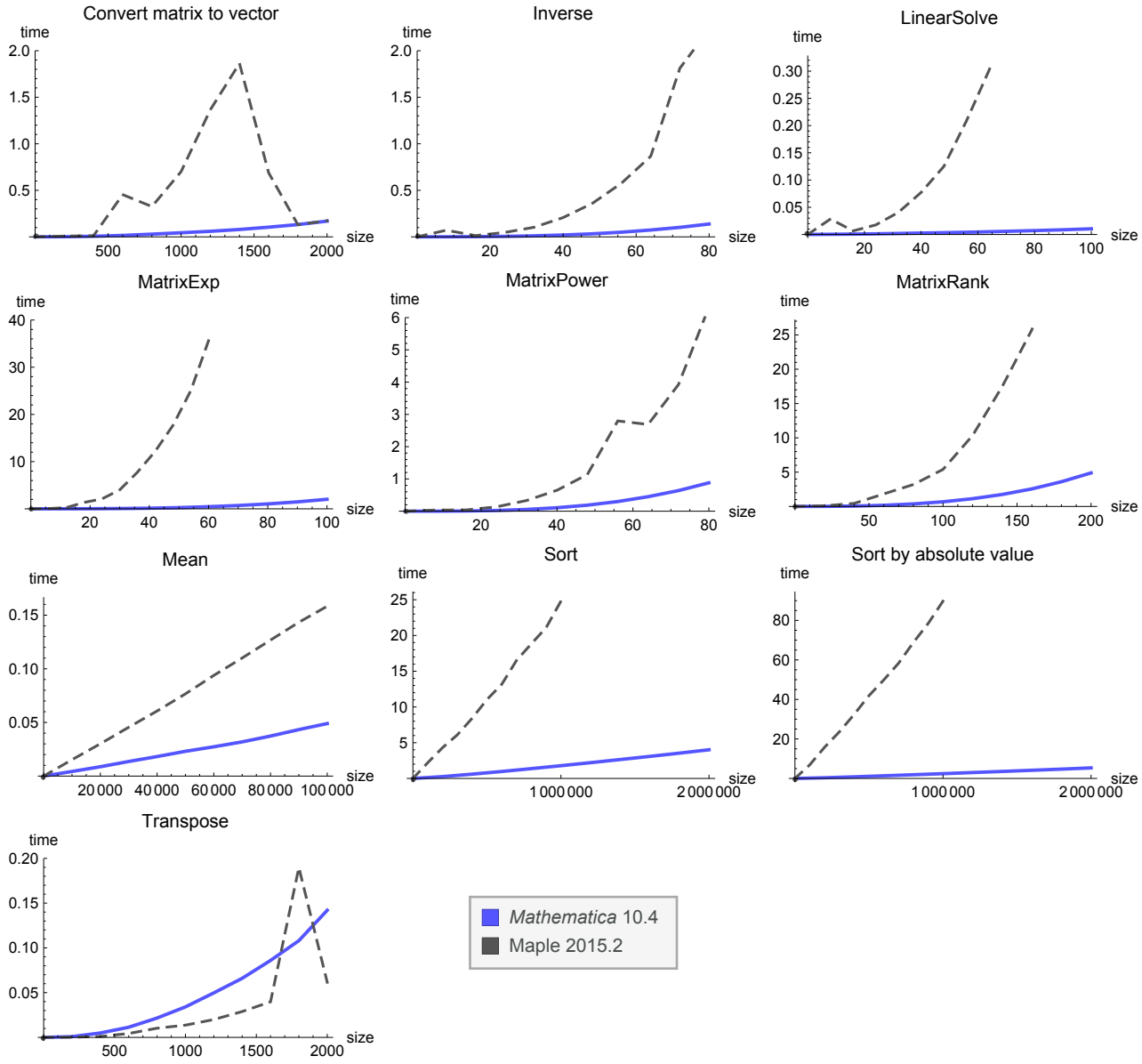
Functions in Maple's `Statistics` packages are limited to machine-precision numbers only. For this reason the `Fit` test has been omitted and `Mean` has been implemented using the `add` command.

Maple cannot evaluate the Inverse task using the `integer[8]` type, so only default data type results are shown for this test.

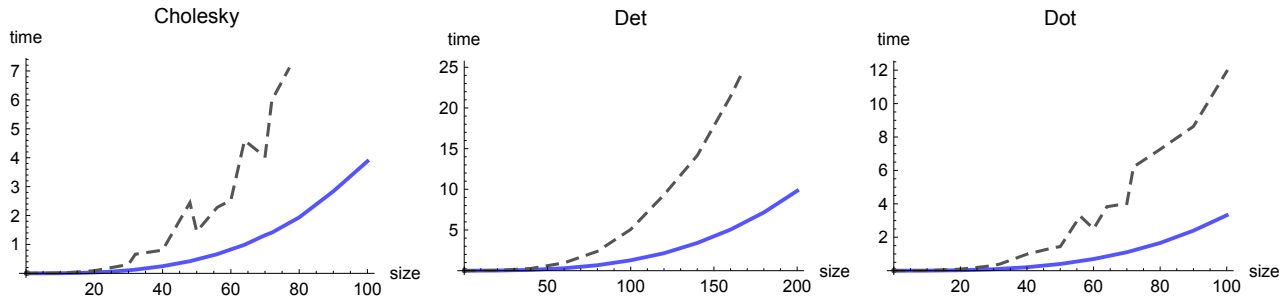
Extended-Precision Data

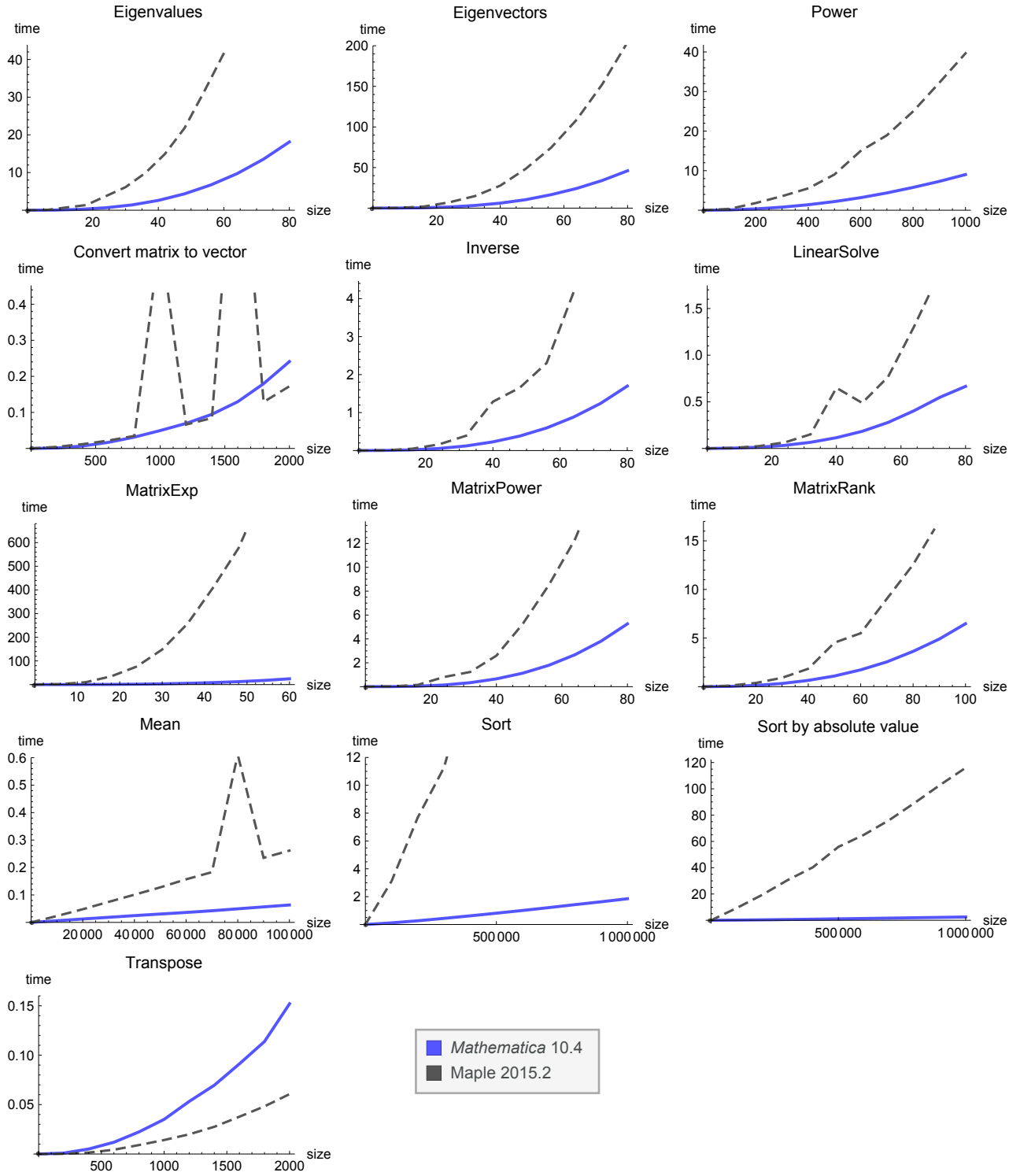
Both *Mathematica* and Maple handle arbitrary-precision arithmetic. Only *Mathematica* tracks the number of reliable digits in the results of such calculations, and yet despite doing this additional validation work, *Mathematica* was 9 times faster for 50-decimal-place matrix computations.





Maple's relative performance improves slightly when calculating at 1000-digit precision.



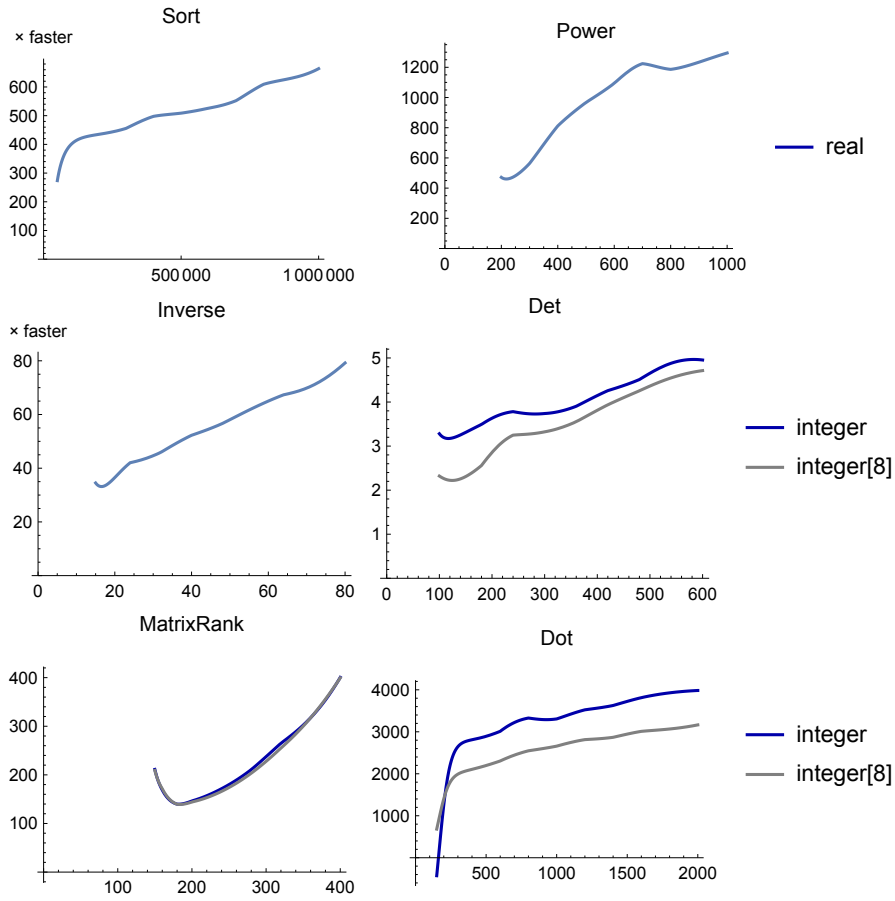


Implementation notes

Some Maple functions are limited to machine-precision numbers. For this reason, tests for `Fit` and `Fourier` have been omitted and `Mean` has been implemented using `add`.

Scalability of Data Operations

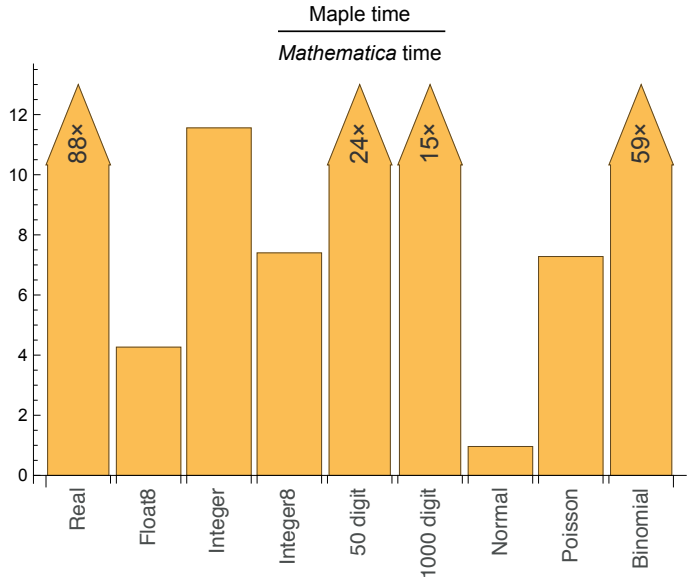
The ratio of Maple's performance to *Mathematica*'s is mostly fairly independent of problem size; however, a few of the operations tested in the preceding sections have been implemented in Maple with poorly scaling algorithms. The larger the problem size, the more *Mathematica* will outperform Maple for these operations.



Random Numbers

Median performance for random number generation shows *Mathematica* to be 7 times faster than Maple.

Matrices with 2000x2000 elements were generated for each number type using a uniform distribution [-10,10]. Normal, Poisson, and Binomial distributions were used to create a vector of 10^7 samples.



Implementation notes

Documented Maple commands were used in all cases.
Maple's default type for reals and its special `float[8]` data generation were both compared to the default data type in *Mathematica*.
Maple's default type for integers and its special `integer[8]` data generation were both compared to the default data type in *Mathematica*.
Maple's Poisson and Binomial distributions (and other discrete distributions) generate real numbers, while *Mathematica* correctly generates integers.

Function Evaluation

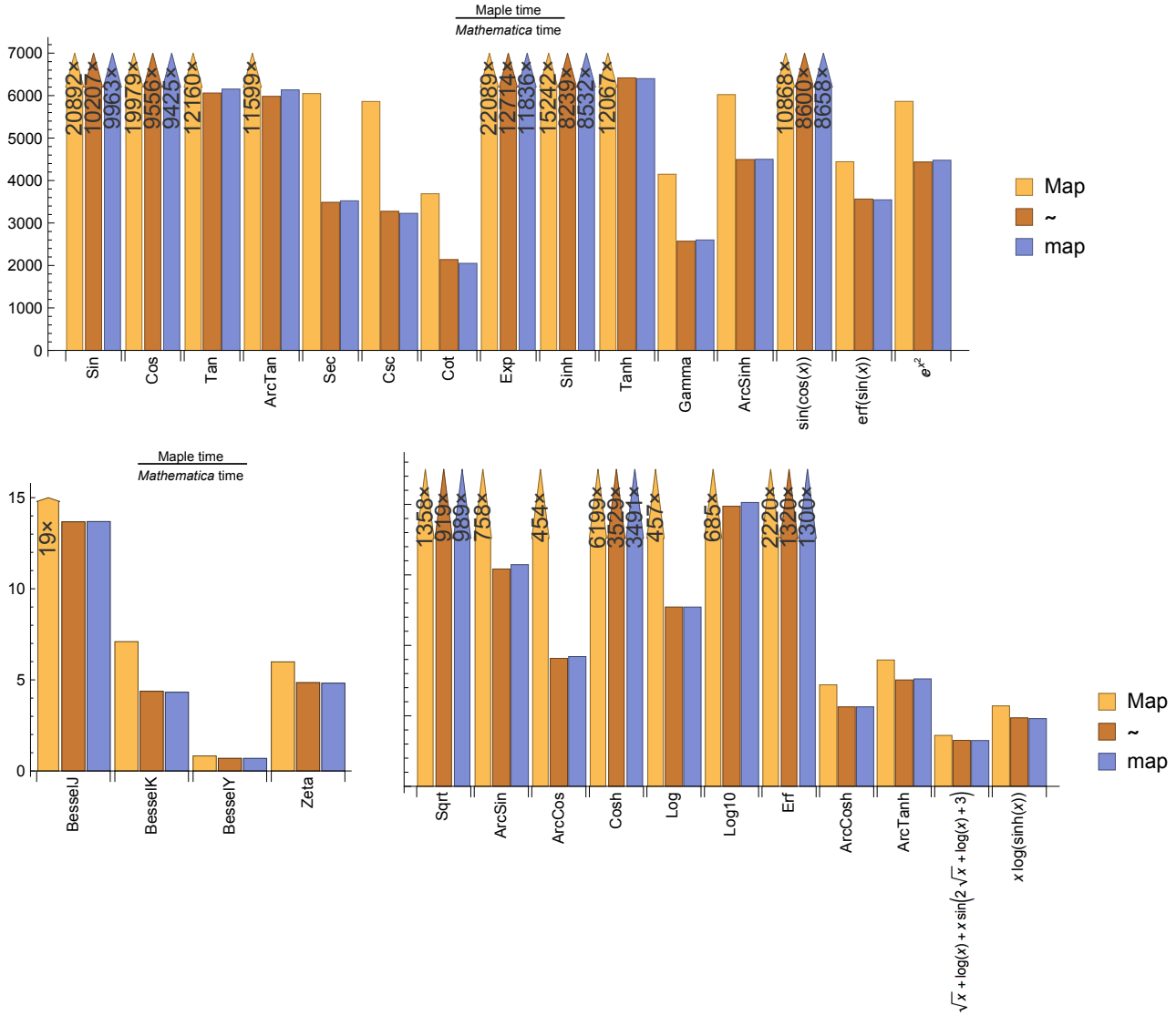
Getting Maple to apply other functions to data with optimal performance is a complicated task with a choice of tools for mapping the functions over the data and a choice of data types. Different circumstances require different choices and are often a trade-off between performance and flexibility or code robustness. In the following tests, each combination of the Maple commands `map`, `Map`, and ~ together with appropriate default and manual data specifications have been compared to *Mathematica*'s `Map` command applied to the default data type.

Maple's `map` command is functionally equivalent to *Mathematica*'s `Map`, while Maple's `Map` is destructive (It replaces the original data, so extra work would be needed if one wanted to preserve or reuse the input data.).

Reals

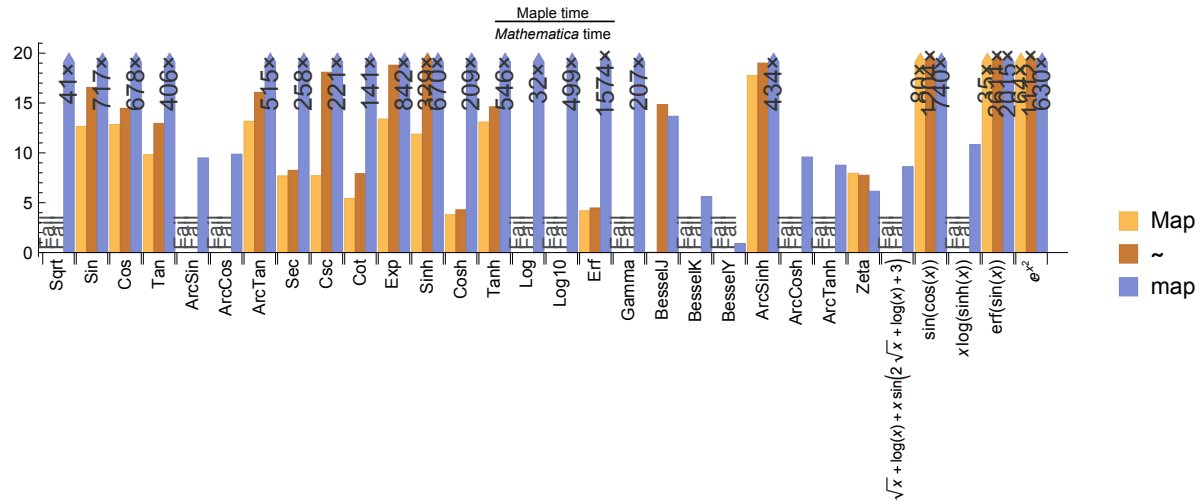
Using the default data type to represent data (as most users would), Maple is generally extremely slow compared to *Mathematica*, with the exception of Bessel functions.

Times to evaluate functions over 10^7 real numbers are compared below.



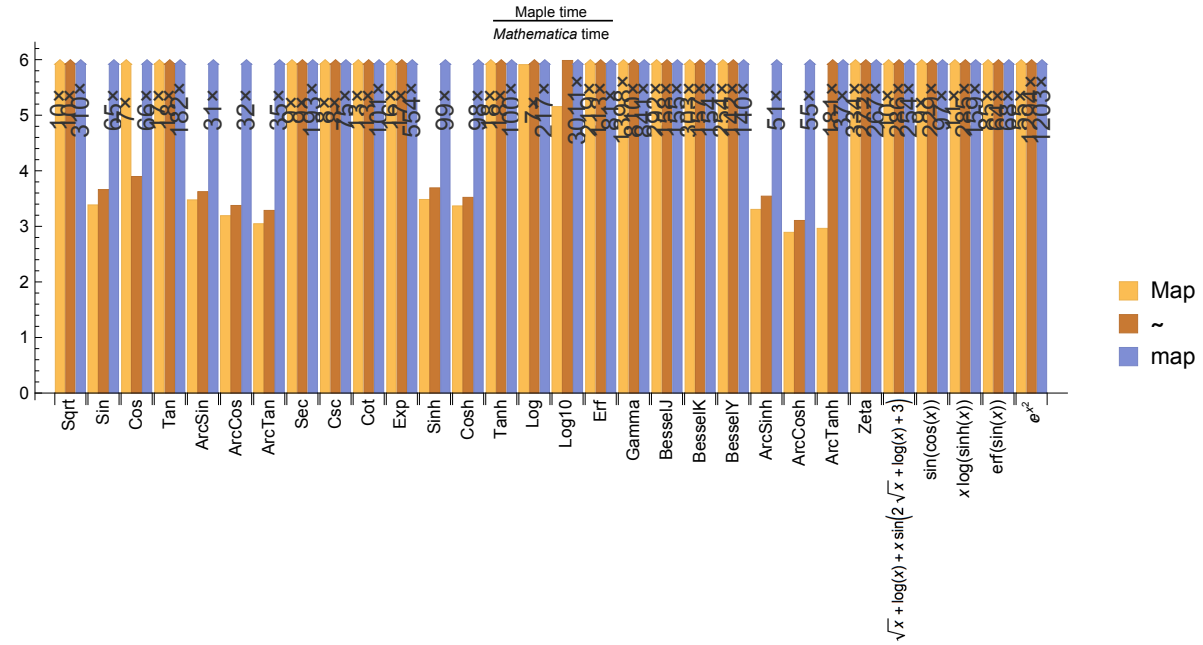
Better performance can be achieved in Maple by manually specifying (or converting) data to the `float[8]` data type. However, this comes at considerable cost to flexibility: firstly you must ensure that all the input data consists of machine reals. Code will fail if you attempt to store a value which is complex, symbolic, string, or NaN value. More importantly, code will also fail completely if any single computation, using `Map` or ~, *returns* a value which is not real. This means that you can use this approach only when you can completely predict the domain of both the input and output data.

Using the input data set in the interval [-10,10], Maple's `Map` and ~ commands fail for the 12 tests which return complex results, and `map`, while more robust, is typically more than 15 times slower than *Mathematica*. It is also important to note that while `Map` and ~ operations are generally much faster than `map` for applying single functions, this advantage largely disappears for applying composite functions to the data.

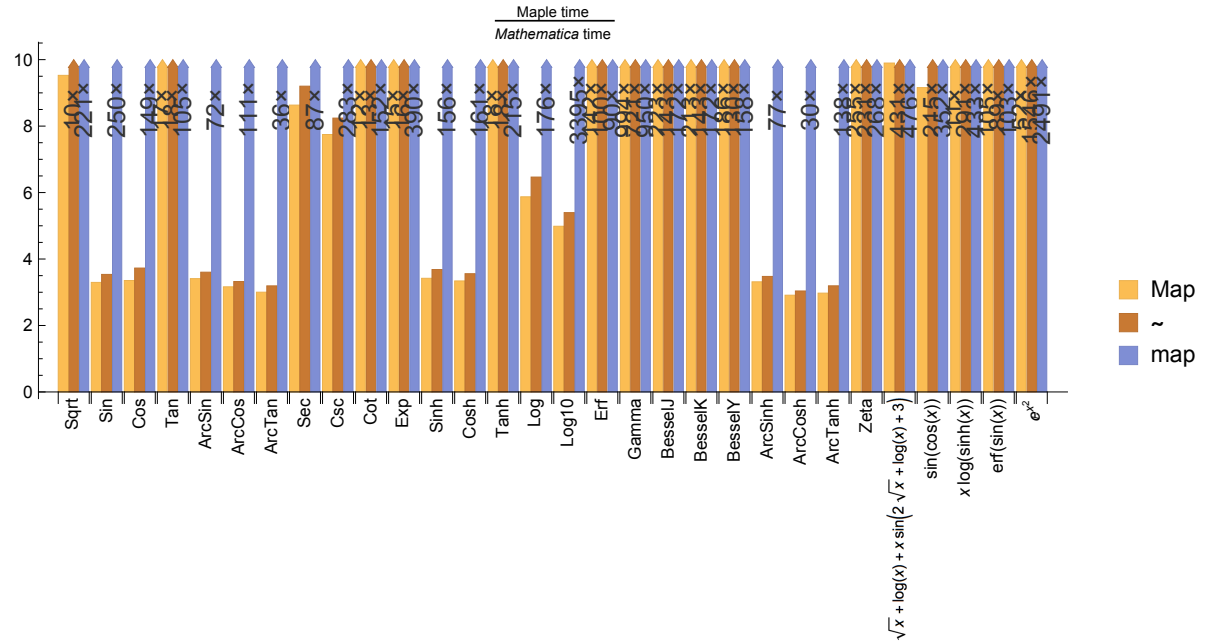


Complexes

For operations on complex numbers, Maple is often only modestly slower than *Mathematica* when using `Map` or ~ to apply single built-in functions. For composite functions and some special functions, the difference is much greater. When using `map` (the direct equivalent of *Mathematica*'s `Map` command), the difference is much greater.

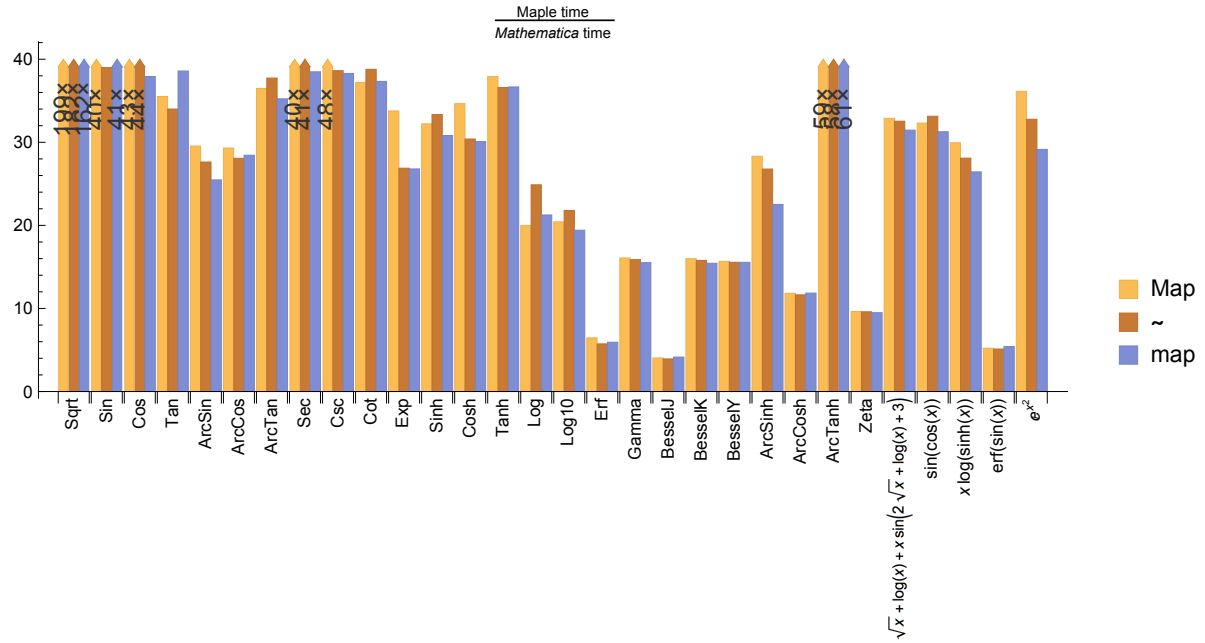


There is a small benefit to using the `complex[8]` data type for Maple's `Map` and ~ commands at the cost of slowing down operations using `map`.



Extended precision

When working in 50-decimal-place extended arithmetic, it makes little difference how you apply functions in Maple. The median performance is 30 times slower than *Mathematica.*



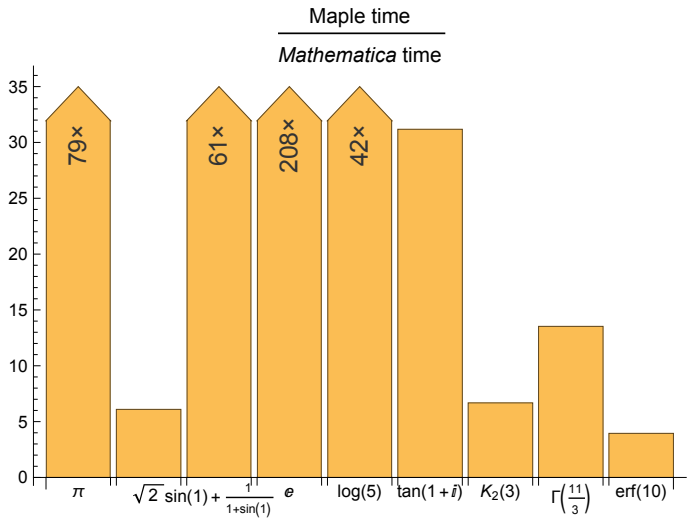
Getting the best performance out of Maple requires significant system knowledge and carefully chosen manual intervention.
In contrast, all *Mathematica* computations were performed using the same function and the same automatic data type, and still outperformed Maple in every test except `BesselY`.

Implementation notes

All tests performed on a vector of 10^7uniform random numbers in the interval [10,10] or [-10,10]+I*[-10,10] for complex data.

High-Precision Evaluation

When evaluating exact numeric expressions to very high precision, *Mathematica* provides automatic precision tracking to ensure that it achieves the target number of correct digits (as opposed to just using input with the target number of digits). The examples in this test were too simple for this to matter, but despite this extra verification work, *Mathematica* evaluated the following expressions to high precision with a median performance 31 times faster than Maple.



Implementation notes

`BesselK`, `Gamma`, and `Erf` calculations were evaluated to 5000 digits.
Other expressions were evaluated to 1 000 000 digits.

Exact Numeric Functions

While calculating exact numeric results (in terms of rationals, radicals, and constants such as π) is arguably symbolic computation, Maple's performance does not appear to improve.

Function	Largest test value	*Mathematica* speed
BernoulliB	6000	2 times faster
Fibonacci	500 000	3 times faster
HarmonicNumber	10 000	Maple cannot compute for n > 500
Zeta	100 000	11 times faster
Binomial	10^{14}	88 times faster

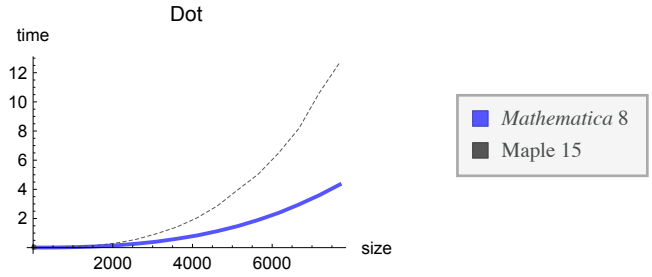
Implementation notes

Maple does not automatically evaluate `Zeta(n)` for n>50, so `expand(Zeta(n))` is used.

GPU Performance

Maple's CUDA support is extremely limited with only a single function, `Dot`, implemented and only for double-precision CUDA hardware. It has no OpenCL or single-precision support. *Mathematica* can run arbitrary CUDA or OpenCL code and has many built-in CUDA accelerated functions.

Testing the one function that Maple does support shows that the *Mathematica* implementation is 3 times faster.



Implementation notes

CUDA tests were performed using Maple 15 and *Mathematica* 8.

General Testing Methodology

Tests were performed using Windows 7 64-bit with 3.07 GHz quad-core Intel i7 processors with 24 GB of RAM. CUDA tests used a Tesla C2050/C2070 GPU with 448 cores and 2.5 GB of RAM. Tests were performed using Maple 2015.1 and *Mathematica* 10.3 default installations, except CUDA tests which used Maple 15 and *Mathematica* 8.

Some tests have been scaled back to smaller problems in the Maple version to enable Maple to perform them in the available time and memory. Where standard Maple functions cannot perform the requested computations, tests have been excluded, or, where trivial, alternative Maple implementations have been created. See source code for details. Median values are calculated only from tests that Maple is able to perform.

For most functions, the test suite finds the average time for 5 evaluations for each size of problem, but high-precision evaluations were performed only once in each test run, since both systems cache the results for later reuse.

Maple is unable to perform the entire test suite in 20GB of RAM, so where tests crashed or failed to complete within 24 hours, Maple was restarted and the test repeated. Such failures have not been included in the results. *Mathematica* tests were performed in a single run.

Relative performance ratios use the largest problem size for each test performed, except where the Maple test has had to be scaled back. In these cases, the available data has been fitted to the curve ax^2+bx, and this model used to estimate a time for the largest problem size timed in *Mathematica*.

Source code for the tests is included so that results can be replicated independently. The entire test program takes approximately 150 minutes in *Mathematica* and several days in Maple.

Revision Notes

Significant changes since version 3 of this benchmark:
• Increased size of test suite from 579 tests to 585 tests
• Reinstated use of Maple functions `Eigenvalues` and `Eigenvectors` which no longer crash
• Increased the scale of some smaller tests

Significant changes since version 2 of this benchmark:
• Increased size of test suite from 508 tests to 579 tests
• Increased scale of some tests
• Used new random number generators in Maple 18
• Used deprecated “eigenvalues” and “eigenvectors” commands to work around new crashes in Maple 18
• Increased memory on test machine

Significant changes since version 1 of this benchmark:
• Increased size of test suite from 164 tests to 508 tests
• Tested Maple “Map” and “In place” (~) command as well as “map”
• Changed data range from [0,1] to [-10,10]
• Reduced the repetitions from 10 to 5 and steps from 20 to 10
• Fixed faulty LinearProgramming tests to give the same tasks to both systems and ensure feasibility
• Improved localization within Maple test code
• Removed programming tests which did not relate to numeric computation
• Used more modern test computer with more memory

Appendix: Test source code

Mathematica code

Test utilities

```
$HistoryLength = 0;
steps = 10;
repeats = 5;
maxVector = 10^6;
maxMatrix = 2000;
maxSparseMatrix = 20 000;

makeData[type_, i_] := Switch[type,
  "RealVector", RandomReal[{-10, 10}, i],
  "RealMatrix", RandomReal[1, {i, i}],
  "ExtendedMatrix", RandomReal[1, {i, i}, WorkingPrecision → 50],
  "ExtendedMatrix1000", RandomReal[1, {i, i}, WorkingPrecision → 1000],
  "ExtendedVector", RandomReal[1, {i}, WorkingPrecision → 50],
  "ExtendedVector1000", RandomReal[1, {i}, WorkingPrecision → 1000],
  "IntegerMatrix", RandomInteger[100, {i, i}],
  "IntegerVector", RandomInteger[100, i],
  "SparseMatrix",
  SparseArray[Table[{RandomInteger[{1, i}], RandomInteger[{1, i}]} → Random[],
    {i^2 / 10 000}], {i, i}],
  "SparseVector", SparseArray[Table[RandomInteger[{1, i}] → Random[],
    {Floor[i / 10 000 ]}], {i}],
  "MediumSparseMatrix", SparseArray[Table[{RandomInteger[{1, i}],
      RandomInteger[{1, i}]} → Random[], {i^2 / 200 }], {i, i}],
  "ComplexMatrix", RandomComplex[{0, 1 + I}, {i, i}],
  "ComplexVector", RandomComplex[{0, 1 + I}, {i}],
  _, Print[type]
 ]
```

```
timedReport[path_, fn_, hi_, type_] := Block[{data}, Export[
    "Mathematica" <> ToString[$VersionNumber] <> path <> ".txt", Table[{Floor[t],
      Mean[Table[
        data = makeData[type, Floor[t]];
        AbsoluteTiming[fn[data]][[1]], {repeats}]]}
     , {t, hi/steps, hi, hi/steps}], "Table"]];

highPrecisionReport[path_, fns_, n_] :=
 Export["Mathematica" <> ToString[$VersionNumber] <> path <> ".txt",
  Map[AbsoluteTiming[N[#, n]][[1]] &, fns]]

NumericTest[path_, fns_List, n_, type_] :=
  Block[{data}, Export["Mathematica" <> ToString[$VersionNumber] <> path <> ".txt",
    Table[data = makeData[type, n];
     AbsoluteTiming[i[data]][[1]], {i, fns}]]];

evaluateTest[path_, expr_Hold] :=
 Export["Mathematica" <> ToString[$VersionNumber] <> path <> ".txt",
  Apply[List, First /@ Map[AbsoluteTiming, expr]]]
```

Function evaluation

```
fnList = {Sqrt, Sin, Cos, Tan, ArcSin, ArcCos, ArcTan,
   Sec, Csc, Cot, Exp, Sinh, Cosh, Tanh, Log, Log10, Erf, Gamma,
   BesselJ[0, #] &, BesselK[1, #] &, BesselY[3, #] &, ArcSinh, ArcCosh,
   ArcTanh, Zeta, # Sin[Log[#] + 2 * Sqrt[#] + 3] + Sqrt[#] + Log[#] &,
   Sin[Cos[#]] &, # Log[Sinh[#]] &, Erf[Sin[#]] &, Exp[#^2] &};

NumericTest["ElementaryFunctions", fnList, 10 maxVector , "RealVector"];
NumericTest["ElementaryFunctionsComplex", fnList, maxVector , "ComplexVector"];
NumericTest["ElementaryFunctionsExtended",
  fnList, maxVector / 10, "ExtendedVector"];
NumericTest["ElementaryFunctionsExtended1000", fnList,
  maxVector / 100, "ExtendedVector1000"];
NumericTest["ElementaryFunctionsSparse", fnList,
  100 000 maxVector , "SparseVector"];
```

Real matrix operations

```
timedReport["FourierReal", Fourier, maxVector, "RealVector"];
timedReport["SortReal", Sort, maxVector, "RealVector"];
timedReport["SortCustomReal", SortBy[#, Abs] &, maxVector, "RealVector"];
timedReport["MeanReal", Mean, 10 maxVector, "RealVector"];
timedReport["DotReal", #.# &, maxMatrix , "RealMatrix"];
timedReport["InverseReal", Inverse, maxMatrix, "RealMatrix"];
Quiet@timedReport["LinearSolveReal",
    LinearSolve[#, #[[1]]] &, maxMatrix, "RealMatrix"];
timedReport["CholeskyReal", CholeskyDecomposition[Transpose[#].#] &,
  maxMatrix, "RealMatrix"];
timedReport["MatrixPowerReal", MatrixPower[#, 5] &, maxMatrix, "RealMatrix"];
timedReport["DetReal", Det, maxMatrix, "RealMatrix"];
timedReport["TransposeReal", Transpose, maxMatrix, "RealMatrix"];
timedReport["FlattenReal", Flatten, maxMatrix, "RealMatrix"];
timedReport["EigenvaluesReal", Eigenvalues, maxMatrix / 2, "RealMatrix"];
timedReport["EigenvectorsReal", Eigenvectors, maxMatrix / 2, "RealMatrix"];
timedReport["LinearProgrammingReal",
  LinearProgramming[Abs[#[[1]]], Abs[#], Abs[#[[2]]], Method → "CLP"] &,
  maxMatrix, "RealMatrix"];
timedReport["ElementPowerReal", #^5 &, maxMatrix, "RealMatrix"];
timedReport["MovingAverageReal",
  MovingAverage[#, 10] &, maxVector, "RealVector"];
timedReport["FitReal", (Function[{data}, Fit[data, x, x]][Transpose[{#, #}]]) &;,
  4 * maxVector, "RealVector"];
timedReport["MatrixExpReal", MatrixExp, maxMatrix / 5, "RealMatrix"];
timedReport["CovarianceReal", Covariance, maxMatrix, "RealMatrix"];
timedReport["MatrixRank", MatrixRank, 3 maxMatrix, "RealMatrix"];
```

Complex matrix operations

```
timedReport["FourierComplex", Fourier, maxVector, "ComplexVector"];
timedReport["SortCustomComplex", SortBy[#, Abs] &, maxVector, "ComplexVector"];
timedReport["MeanComplex", Mean, 10 maxVector, "ComplexVector"];
timedReport["DotComplex", #.# &, maxMatrix , "ComplexMatrix"];
timedReport["InverseComplex", Inverse, maxMatrix, "ComplexMatrix"];
timedReport["LinearSolveComplex",
  LinearSolve[#, #[[1]]] &, maxMatrix, "ComplexMatrix"];
timedReport["CholeskyComplex", CholeskyDecomposition[ConjugateTranspose[#].#] &,
  maxMatrix, "ComplexMatrix"];
timedReport["MatrixPowerComplex", MatrixPower[#, 5] &,
  maxMatrix / 2, "ComplexMatrix"];
timedReport["DetComplex", Det, maxMatrix, "ComplexMatrix"];
timedReport["EigenvaluesComplex", Eigenvalues, maxMatrix / 2, "ComplexMatrix"];
timedReport["EigenvectorsComplex", Eigenvectors, maxMatrix / 2, "ComplexMatrix"];
timedReport["TransposeComplex", Transpose, maxMatrix, "ComplexMatrix"];
timedReport["FlattenComplex", Flatten, maxMatrix, "ComplexMatrix"];
timedReport["ElementPowerComplex", #^5 &, maxMatrix, "ComplexMatrix"];
timedReport["MatrixExpComplex", MatrixExp, maxMatrix / 4, "ComplexMatrix"];
timedReport["MatrixRankComplex", MatrixRank, 3 maxMatrix, "ComplexMatrix"]
```

Sparse matrix operations

```
timedReport["DotSparse", #.# &, maxSparseMatrix, "SparseMatrix"];
Quiet@timedReport["LinearSolveSparse",
    LinearSolve[#, #[[1]]] &, maxSparseMatrix / 3, "MediumSparseMatrix"];
timedReport["TransposeSparse", Transpose, 2 * maxSparseMatrix, "SparseMatrix"];
timedReport["FlattenSparse", Flatten, maxSparseMatrix, "SparseMatrix"];
timedReport["ElementPowerSparse", #^5 &, maxSparseMatrix, "SparseMatrix"];
timedReport["MeanSparse", Mean, maxVector * 100, "SparseVector"];
timedReport["MatrixPowerSparse",
  MatrixPower[#, 5] &, 2 * maxSparseMatrix / 5, "SparseMatrix"];
timedReport["MovingAverageSparse", MovingAverage[#, 10] &,
  10 maxVector, "SparseVector"];
```

Integer matrix operations

```
timedReport["DotInteger", #.# &, maxMatrix , "IntegerMatrix"];
timedReport["InverseInteger", Inverse, maxMatrix / 25, "IntegerMatrix"];
timedReport["LinearSolveInteger",
  LinearSolve[#, #[[1]]] &, maxMatrix / 4, "IntegerMatrix"];
timedReport["MatrixPowerInteger", MatrixPower[#, 5] &,
  maxMatrix / 4, "IntegerMatrix"];
timedReport["DetInteger", Det, 600 , "IntegerMatrix"];
timedReport["TransposeInteger", Transpose, maxMatrix, "IntegerMatrix"];
timedReport["FlattenInteger", Flatten, maxMatrix, "IntegerMatrix"];
timedReport["MeanInteger", Mean, maxVector, "IntegerVector"];
timedReport["SortInteger", Sort, maxVector, "IntegerVector"];
timedReport["SortCustomInteger",
  SortBy[#, Abs] &, maxVector, "IntegerVector"];
timedReport["EigenvaluesInteger", Eigenvalues, maxMatrix / 30, "IntegerMatrix"];
timedReport["ElementPowerInteger", #^5 &, maxMatrix, "IntegerMatrix"];
timedReport["MatrixExpInteger", MatrixExp, 40, "IntegerMatrix"];
timedReport["MatrixRankInteger", MatrixRank, maxMatrix, "IntegerMatrix"];
```

50 digit precision operations

```
timedReport["DotExtended", #.# &, maxMatrix / 25 , "ExtendedMatrix"];
timedReport["InverseExtended", Inverse, maxMatrix / 25, "ExtendedMatrix"];
timedReport["CholeskyExtended",
  CholeskyDecomposition[Transpose[#].#] &, maxMatrix / 20, "ExtendedMatrix"];
timedReport["EigenvaluesExtended", Eigenvalues, 60, "ExtendedMatrix"];
timedReport["EigenvectorsExtended", Eigenvectors, 80, "ExtendedMatrix"];
timedReport["ElementPowerExtended", #^5 &, maxMatrix / 2, "ExtendedMatrix"];
timedReport["LinearSolveExtended",
  LinearSolve[#, #[[1]]] &, maxMatrix / 20, "ExtendedMatrix"];
timedReport["DetExtended", Det, maxMatrix / 10, "ExtendedMatrix"];
timedReport["TransposeExtended", Transpose, maxMatrix, "ExtendedMatrix"];
timedReport["FlattenExtended", Flatten, maxMatrix, "ExtendedMatrix"];
(*timedReport["FourierExtended",Fourier,maxVector/100,"ExtendedVector"];
Maple converts to floats*)
```

```
timedReport["MeanExtended", Mean, maxVector / 10, "ExtendedVector"];
timedReport["SortExtended", Sort, 2 * maxVector, "ExtendedVector"];
timedReport["SortCustomExtended",
  SortBy[#, Abs] &, 2 * maxVector, "ExtendedVector"];
timedReport["EigenvectorsExtended", Eigenvectors,
  maxMatrix / 25, "ExtendedMatrix"];
timedReport["MatrixPowerExtended", MatrixPower[#, 5] &,
  maxMatrix / 25, "ExtendedMatrix"];
timedReport["MatrixExpExtended", MatrixExp, 100, "ExtendedMatrix"];
timedReport["MatrixRankExtended", MatrixRank, maxMatrix / 10, "ExtendedMatrix"];
```

1000 digit precision operations

```
timedReport["DotExtended1000", #.# &, maxMatrix / 20 , "ExtendedMatrix1000"];
timedReport["InverseExtended1000",
  Inverse, maxMatrix / 25, "ExtendedMatrix1000"];
timedReport["CholeskyExtended1000", CholeskyDecomposition[Transpose[#].#] &,
  maxMatrix / 20, "ExtendedMatrix1000"];
timedReport["EigenvaluesExtended1000", Eigenvalues, 80, "ExtendedMatrix1000"];
timedReport["EigenvectorsExtended1000", Eigenvectors, 80, "ExtendedMatrix1000"];
timedReport["ElementPowerExtended1000",
  #^5 &, maxMatrix / 2, "ExtendedMatrix1000"];
timedReport["LinearSolveExtended1000", LinearSolve[#, #[[1]]] &,
  maxMatrix / 25, "ExtendedMatrix1000"];
timedReport["DetExtended1000", Det, maxMatrix / 10, "ExtendedMatrix1000"];
timedReport["TransposeExtended1000",
  Transpose, maxMatrix, "ExtendedMatrix1000"];
timedReport["FlattenExtended1000", Flatten, maxMatrix, "ExtendedMatrix1000"];
timedReport["MeanExtended1000", Mean, maxVector / 10, "ExtendedVector1000"];
timedReport["SortExtended1000", Sort, maxVector, "ExtendedVector1000"];
timedReport["SortCustomExtended1000",
  SortBy[#, Abs] &, maxVector, "ExtendedVector1000"];
timedReport["MatrixPowerExtended1000", MatrixPower[#, 5] &,
  maxMatrix / 25, "ExtendedMatrix1000"];
timedReport["MatrixExpExtended1000", MatrixExp, 60, "ExtendedMatrix1000"];
timedReport["MatrixRankExtended1000",
 MatrixRank, maxMatrix / 20, "ExtendedMatrix1000"]
```

Exact functions

```
evaluateTest["ExactFunctions", Hold[
   Table[BernoulliB[i], {i, 6000}],
   Table[Fibonacci[i], {i, 0, 500 000, 1000}],
   (*Table[HarmonicNumber[i],{i,10000}], Maple fails*)
   Table[Zeta[i], {i, 0, 100 000, 10 000}],
   Table[Binomial[i^2, i], {i, 1, 10 000}]
  ]];
```

Random numbers

```
evaluateTest["RandomNumbers", Hold[
   makeData["RealMatrix", 2000],
   makeData["RealMatrix", 2000],
   makeData["IntegerMatrix", 2000],
   makeData["IntegerMatrix", 2000],
   makeData["ExtendedMatrix", 2000],
   makeData["ExtendedMatrix1000", 2000],
   RandomVariate[NormalDistribution[0, 1], 10^7],
   RandomVariate[PoissonDistribution[4], 10^7],
   RandomVariate[BinomialDistribution[10, 0.2], 10^7]
  ]];
```

High precision

```
highPrecisionReport["ManyDigits",
  {Pi, Sqrt[2], Sin[1] + 1/(1 + Sin[1]), Exp[1], Log[5], Tan[1 + I]}, 1 000 000];
highPrecisionReport["FewerDigits", {BesselK[2, 3], Gamma[11 / 3], Erf[10]}, 5000];
```

Maple

Test utilities

```
with(LinearAlgebra);
with(DiscreteTransforms);
with(combinat, fibonacci);
with(Statistics);
with(RandomTools[MersenneTwister]);
with(GraphTheory);
with(RandomGraphs);
with(Optimization);
with(RandomTools);
with(stats);
with(combinat);
steps:=10;
repeats:=5;
maxVector:=10^6;
maxMatrix:=2000;
maxSparseMatrix:=20000;

makeData:=proc(type, size) local dat,  i;
        iftype=“Vector”thendat:=RandomVector[row](size, generator=-10.0..10.0)
                elif type=“Matrix” then dat:=Generate((‘Matrix’)(float(range=-10..10, method=uniform), size, size))
                        elif type=“VectorFloat8” then dat:=RandomVector[row](size, generator=-10.0..10.0, datatype=float[8])
                        elif type=“MatrixFloat8” then dat:=RandomMatrix(size, size, generator=-10.0..10.0, datatype=float[8])
                        elif type=“SparseMatrix” then dat:=Matrix(size, size, storage=sparse);
                                                for i to floor((1/10000)*size^2) do
                                                dat[RandomTools[Generate](integer(range=1..size)),
                                                Generate(integer(range=1..size))]:=2*GenerateFloat()-1 end do
                        elif type=“MediumSparseMatrix” then dat:=Matrix(size, size, storage=sparse);
                                                        for i to floor((1/200)*size^2) do
                                                         dat[RandomTools[Generate](integer(range=1..size)),
                                                         Generate(integer(range=1..size))]:=20*GenerateFloat()-10 end do
                        elift ype=“SparseVector” then dat:=Vector(size, storage=sparse);
                                                for i to floor((1/10000)*size) do
                                                dat[RandomTools[Generate](integer(range=1..size))]:=20*GenerateFloat()-10 end do
                        elif type=“IntegerMatrix” then dat:=Generate((‘Matrix’)(integer(range=-10..10), size, size))
                        elif type=“IntegerVector” then dat:=Generate((‘Vector’)(integer(range=-10..10), size))
                        elif type=“IntegerMatrix8” then dat:=Generate((‘Matrix’)(integer(range=-10..10), size, size, datatype=integer[8]))
                        elif type=“IntegerVector8” then dat:=Generate((‘Vector’)(integer(range=-10..10), size, datatype=integer[8]))
                        elif type=“BigNumberMatrix” then dat:=RandomMatrix(size, size,
                                                        generator=proc(x) 20*GenerateFloat(digits=50)-10 end proc)
                        elif type=“BigNumberVector” then dat:=RandomVector[row](size,
                                                        generator=proc(x) 20*GenerateFloat(digits=50)-10 end proc)
                        elif type=“BigNumberMatrix1000” then dat:=RandomMatrix(size, size,
                                                        generator=proc(x) 20*GenerateFloat(digits=1000)-10 end proc)
```

```
                elif type="BigNumberVector1000" then dat:=RandomVector[row](size,
                                                generator=proc(x) 20*GenerateFloat(digits=1000)-10 end proc)
                eliftype="ComplexVector" then dat:=makeData("Vector", size)+I*makeData("Vector", size)
                elif type="ComplexMatrix" then dat:=makeData("Matrix", size)+I*makeData("Matrix", size)
                elif type="ComplexVector8" then dat:=makeData("VectorFloat8", size)+I*makeData("VectorFloat8", size)
                elif type="ComplexMatrix8" then dat:=makeData("MatrixFloat8", size)+I*makeData("MatrixFloat8", size)
                else print(type) endif;
dat end proc;

timedDataOperation:=proc(expr, size, type) local totaltime,  data,  i;
                        totaltime:=0;
                        for i to repeats do data:=makeData(type, size);
                        totaltime:=totaltime+time[real](expr(data)) end do;
                        totaltime/repeats end proc;

timedReport:=proc(filename, expr, hi, type)
                ExportMatrix(cat("Maple", filename, ".txt"),
                        convert([seq([floor(s),  timedDataOperation('expr', floor(s), type)], s=hi/steps..hi, hi/steps)], Matrix))
        end proc;

numericfntest:=proc(fn, n, type) local data;
                data:=makeData(type, n);
                time[real](map(fn, data)) end proc;

numerictest:=proc(file, fns, n, type) ExportMatrix(cat("Maple", file, ".txt"),  Matrix([seq(numericfntest(i, n, type), iinfns)])) end proc;
numerictestinplace:=proc(file, fns, n, type)
                        ExportMatrix(cat("Maple", file, ".txt"),  Matrix([seq(numericfntestinplace(i, n, type), iinfns)]))
                end proc;
numerictestelementwise:=proc(file, fns, n, type)
                                ExportMatrix(cat("Maple", file, ".txt"), Matrix([seq(elementwisefntest(i, n, type), iinfns)]))
                        end proc;
numericfntestinplace:=proc(fn, n, type) local data;
                        data:=makeData(type, n);
                        time[real](Map(fn, data)) end proc;
elementwisefntest:=proc(fn, n, type) local data;
                        data:=makeData(type, n);
                        time[real](fn(data)) end proc;
highPrecisionTest:=proc(file, fns, n)
                ExportMatrix(cat("Maple", file, ".txt"),  Matrix([seq(time[real](evalf(i, n)), iinfns)])) end proc;
RandomInteger:=proc(x, y) RandomTools[Generate](integer(range=1..100)) end proc;


fnList:=[sqrt, sin, cos, tan, arcsin, arccos, arctan, sec, csc, cot, exp, sinh, cosh, tanh, log, log10, erf, GAMMA,
        proc(x) BesselJ(0, x) end proc,  proc(x) BesselK(1, x) end proc,  proc(x) BesselY(3, x) end proc,
        arcsinh, arccosh, arctanh, Zeta, proc(x) x*sin(log(x)+2*sqrt(x)+3)+sqrt(x)+log(x) end proc,
        proc(x) sin(cos(x)) end proc,  proc(x) log(sinh(x)) end proc,  proc(x) erf(sin(x)) end proc,  proc(x) exp(x^2) end proc];

elementwisefnList:=[`~`[sqrt], `~`[sin], `~`[cos], `~`[tan], `~`[arcsin], `~`[arccos], `~`[arctan], `~`[sec], `~`[csc], `~`[cot], `~`[exp], `~`[sinh],
                `~`[cosh], `~`[tanh], `~`[log], `~`[log10], `~`[erf], `~`[GAMMA],  `~`[proc(x) BesselJ(0, x) end proc],
                `~`[proc(x) BesselK(1, x) end proc], `~`[proc(x) BesselY(3, x) end proc], `~`[arcsinh], `~`[arccosh],
                `~`[arctanh], `~`[Zeta], `~`[proc(x) x*sin(log(x)+2*sqrt(x)+3)+sqrt(x)+log(x) end proc],
                `~`[proc(x) sin(cos(x)) end proc], `~`[proc(x) log(sinh(x)) end proc], `~`[proc(x) erf(sin(x)) end proc],
                `~`[proc(x) exp(x^2) end proc]];
fnListSafe:=[sin, cos, tan, arctan, sec, csc, cot, exp, sinh, cosh, tanh, erf, proc(x) BesselJ(0, x) end proc, arcsinh, Zeta,
                proc(x) sin(cos(x)) end proc, proc(x) erf(sin(x)) end proc, abs, floor, proc(x) exp(x^2) end proc];
elementwisefnListSafe:=[`~`[sin], `~`[cos], `~`[tan], `~`[arctan], `~`[sec], `~`[csc], `~`[cot], `~`[exp], `~`[sinh], `~`[cosh], `~`[tanh],
                `~`[erf], `~`[proc(x) BesselJ(0, x) end proc], `~`[arcsinh], `~`[Zeta],
                `~`[proc(x) sin(cos(x)) end proc], `~`[proc(x) erf(sin(x)) end proc], `~`[proc(x) exp(x^2) end proc]];
```

Sparse matrix operations

```
timedReport("LinearSolveSparse", 'proc(x) LinearSolve(x, Column(x, 1)) end proc', (1/3)*maxSparseMatrix, "MediumSparseMatrix");
timedReport("DotSparse", 'proc(x) x.xend proc', maxSparseMatrix, "SparseMatrix");
timedReport("TransposeSparse", 'Transpose', 2*maxSparseMatrix, "SparseMatrix");
timedReport("FlattenSparse", 'proc(m) convert(m, Vector[row]) end proc', 10*maxMatrix, "SparseMatrix");
timedReport("MatrixPowerSparse", 'proc(x) x^5end proc', 4*maxMatrix, "SparseMatrix");
timedReport("MeanSparse", 'Mean', 100*maxVector, "SparseVector");
timedReport("ElementPowerSparse", 'proc(m) map(proc(x) x^5end proc, m) end proc', maxSparseMatrix, "SparseMatrix");
```

Real matrix operations

```
timedReport("FourierReal", 'FourierTransform', maxVector, "Vector");
timedReport("SortReal", 'sort', maxVector, "Vector");
timedReport("MeanReal", 'Mean', 2*maxVector, "Vector");
timedReport("DotReal", 'proc(x) x.xend proc', maxMatrix, "Matrix");
timedReport("InverseReal", 'MatrixInverse', maxMatrix, "Matrix");
timedReport("LinearSolveReal", 'proc(x) LinearSolve(x, Column(x, 1)) end proc', maxMatrix, "Matrix");
timedReport("CholeskyReal", 'proc(S) LUDecomposition*(Transpose(S).S, method='Cholesky') end proc', maxMatrix, "Matrix");
timedReport("MatrixPowerReal", 'proc(x) x^5end proc', maxMatrix, "Matrix");
timedReport("DetReal", 'Determinant', maxMatrix, "Matrix");
timedReport("TransposeReal", 'Transpose', maxMatrix, "Matrix");
timedReport("FlattenReal", 'proc(m) convert(m, Vector[row]) end proc', maxMatrix, "Matrix");
timedReport("EigenvaluesReal", 'Eigenvalues', (1/2)*maxMatrix, "Matrix");
timedReport("EigenvectorsReal", 'Eigenvectors', (1/2)*maxMatrix, "Matrix");
timedReport("LinearProgrammingReal", 'proc(x) LPSolve(abs(Row(x, 1)), [-abs(x), -abs(Row(x, 2))], [0, infinity]) end proc',
            (1/4)*maxMatrix, "Matrix");
timedReport("ElementPowerReal", 'proc(m) map(proc(x) x^5end proc, m) end proc', (1/2)*maxMatrix, "Matrix");
timedReport("FitReal", 'proc(m) Fit(b*x+a, m, m, x) end proc', maxVector, "Vector");
timedReport("MatrixExpReal", 'MatrixExponential', (1/5)*maxMatrix, "Matrix");
timedReport("CovarianceReal", 'CovarianceMatrix', maxMatrix, "Matrix");
timedReport("MatrixRankReal", 'LinearAlgebra:-Rank', 3*maxMatrix, "Matrix");
timedReport("FourierFloat8", 'FourierTransform', maxVector, "VectorFloat8");
timedReport("SortFloat8", 'sort', maxVector, "VectorFloat8");
timedReport("MeanFloat8", 'Mean', 10*maxVector, "VectorFloat8");
timedReport("DotFloat8", 'proc(x) x.xend proc', maxMatrix, "MatrixFloat8");
timedReport("InverseFloat8", 'MatrixInverse', maxMatrix, "MatrixFloat8");
timedReport("LinearSolveFloat8", 'proc(x) LinearSolve(x, Column(x, 1)) end proc', maxMatrix, "MatrixFloat8");
timedReport("CholeskyFloat8", 'proc(S) LUDecomposition*(Transpose(S).S, method='Cholesky') end proc', maxMatrix, "MatrixFloat8");
timedReport("MatrixPowerFloat8", 'proc(x) x^5end proc', maxMatrix, "MatrixFloat8");
timedReport("DetFloat8", 'Determinant', maxMatrix, "MatrixFloat8");
timedReport("TransposeFloat8", 'Transpose', maxMatrix, "MatrixFloat8");
timedReport("FlattenFloat8", 'proc(m) convert(m, Vector[row]) end proc', maxMatrix, "MatrixFloat8");
timedReport("EigenvaluesFloat8", 'Eigenvalues', (1/2)*maxMatrix, "MatrixFloat8");
timedReport("EigenvectorsFloat8", 'Eigenvectors', (1/2)*maxMatrix, "MatrixFloat8");
timedReport("LinearProgrammingFloat8", 'proc(x) LPSolve(abs(Row(x, 1)), [-abs(x), -abs(Row(x, 2))], [0, infinity]) end proc',
            (1/4)*maxMatrix, "MatrixFloat8");
timedReport("ElementPowerFloat8", 'proc(m)map(proc(x) x^5end proc, m) end proc', (1/2)*maxMatrix, "MatrixFloat8");
timedReport("FitFloat8", 'proc(m)Fit(b*x+a, m, m, x) end proc', maxVector, "VectorFloat8");
timedReport("MatrixExpFloat8", 'MatrixExponential', (1/5)*maxMatrix, "MatrixFloat8");
timedReport("CovarianceFloat8", 'CovarianceMatrix', maxMatrix, "MatrixFloat8");
timedReport("MatrixRankFloat8", 'LinearAlgebra:-Rank', 3* maxMatrix, "MatrixFloat8");

timedReport("LinearProgrammingReal", 'proc(x) LPSolve(abs(Row(x, 1)), [-abs(x), -abs(Row(x, 2))], [0, infinity]) end proc', maxMatrix, "Matrix");
timedReport("LinearProgrammingFloat8", 'proc(x) LPSolve(abs(Row(x, 1)), [-abs(x), -abs(Row(x, 2))], [0, infinity]) end proc',
            maxMatrix, "MatrixFloat8");
timedReport(“SortCustomFloat8”, ‘proc(data) sort(data, proc(x, y) abs(y)<abs(x) end proc) end proc’, (1/25)*maxVector, “VectorFloat8”);
timedReport(“SortCustomReal”, ‘proc(data) sort(data, proc(x, y) abs(y)<abs(x) end proc) end proc’, (1/25)*maxVector, “Vector”);
```

Complex matrix operations

```
timedReport("FourierComplex", 'FourierTransform', maxVector, "ComplexVector");
timedReport("SortComplex", 'sort', maxVector, "ComplexVector");
timedReport("MeanComplex", 'proc(x) add(i, i=x)/Dimension(x) end proc', (1/3)*maxVector, "ComplexVector");
timedReport("DotComplex", 'proc(x) x.xend proc', maxMatrix, "ComplexMatrix");
timedReport("InverseComplex", 'MatrixInverse', maxMatrix, "ComplexMatrix");
timedReport("LinearSolveComplex", 'proc(x) LinearSolve(x, Column(x, 1)) end proc', maxMatrix, "ComplexMatrix");
timedReport("CholeskyComplex", 'proc(S) LUDecomposition(map(conjugate, Transpose(S)).S,
            method='Cholesky') end proc', maxMatrix, "ComplexMatrix");
timedReport("MatrixPowerComplex", 'proc(x) x^5end proc', (1/2)*maxMatrix, "ComplexMatrix");
timedReport("DetComplex", 'Determinant', maxMatrix, "ComplexMatrix");
timedReport("EigenvaluesComplex", 'Eigenvalues', (1/2)*maxMatrix, "ComplexMatrix");
timedReport("EigenvectorsComplex", 'Eigenvectors', (1/2)*maxMatrix, "ComplexMatrix");
timedReport("TransposeComplex", 'Transpose', maxMatrix, "ComplexMatrix");
timedReport("FlattenComplex", 'proc(m) convert(m, Vector[row]) end proc', maxMatrix, "ComplexMatrix");
timedReport("ElementPowerComplex", 'proc(m) map(proc(x)x^5end proc, m) end proc', (1/4)*maxMatrix, "ComplexMatrix");
timedReport("MatrixExpComplex", 'MatrixExponential', (1/4)*maxMatrix, "ComplexMatrix");
timedReport("MatrixRankComplex", 'LinearAlgebra:-Rank', 3*maxMatrix, "ComplexMatrix");
timedReport("FourierComplex8", 'FourierTransform', maxVector, "ComplexVector8");
timedReport("SortComplex8", 'sort', maxVector, "ComplexVector");
timedReport("MeanComplex8", 'proc(x) add(i, i=x)/Dimension(x) end proc', (1/3)*maxVector, "ComplexVector8");
timedReport("DotComplex8", 'proc(x) x.xend proc', maxMatrix, "ComplexMatrix8");
```

```
timedReport("InverseComplex8", 'MatrixInverse', maxMatrix, "ComplexMatrix8");
timedReport("LinearSolveComplex8", 'proc(x) LinearSolve(x, Column(x, 1)) end proc', maxMatrix, "ComplexMatrix8");
timedReport("CholeskyComplex8", 'proc(S) LUDecomposition(map(conjugate, Transpose(S)).S,
        method='Cholesky') end proc', maxMatrix, "ComplexMatrix8");
timedReport("MatrixPowerComplex8", 'proc(x) x^5end proc', (1/2)*maxMatrix, "ComplexMatrix8");
timedReport("DetComplex8", 'Determinant', maxMatrix, "ComplexMatrix8");
timedReport("EigenvaluesComplex8", 'Eigenvalues', (1/2)*maxMatrix, "ComplexMatrix8");
timedReport("EigenvectorsComplex8", 'Eigenvectors', (1/2)*maxMatrix, "ComplexMatrix8");
timedReport("TransposeComplex8", 'Transpose', maxMatrix, "ComplexMatrix8");
timedReport("FlattenComplex8", 'proc(m) convert(m, Vector[row]) end proc', 3*maxMatrix*(1/4), "ComplexMatrix8");
timedReport("ElementPowerComplex8", 'proc(m) map(proc(x)x^5end proc, m) end proc', (1/4)*maxMatrix, "ComplexMatrix8");
timedReport("MatrixExpComplex8", 'MatrixExponential', (1/4)*maxMatrix, "ComplexMatrix8");
timedReport("MatrixRankComplex8", 'LinearAlgebra:-Rank', 3*maxMatrix, "ComplexMatrix8");
timedReport("SortCustomComplex", 'proc(data) sort(data, proc(x, y) abs(y)<abs(x) end proc) end proc',
        (1/25)*maxVector, "ComplexVector");
timedReport("SortCustomComplex8", 'proc(data) sort(data, proc(x, y) abs(y)<abs(x) end proc) end proc',
        (1/25)*maxVector, "ComplexVector8");
```

Integer matrix operations

```
timedReport("DotInteger", 'proc(x) x.x end proc', maxMatrix, "IntegerMatrix");
timedReport("InverseInteger", 'MatrixInverse', (1/25)*maxMatrix, "IntegerMatrix");
timedReport("LinearSolveInteger", 'proc(x) LinearSolve(x, Column(x, 1)) end proc', (1/4)*maxMatrix, "IntegerMatrix");
timedReport("MatrixPowerInteger", 'proc(x) x^5end proc', (1/4)*maxMatrix, "IntegerMatrix");
timedReport("DetInteger", 'Determinant', 600, "IntegerMatrix");
timedReport("TransposeInteger", 'Transpose', maxMatrix, "IntegerMatrix");
timedReport("FlattenInteger", 'proc(m) convert(m, Vector[row]) end proc', maxMatrix, "IntegerMatrix");
timedReport("MeanInteger", 'proc(x) add(i, i=x)/Dimension(x) end proc', maxVector, "IntegerVector");
timedReport("SortInteger", 'sort', maxVector, "IntegerVector");
timedReport("EigenvaluesInteger", 'Eigenvalues', (1/30)*maxMatrix, "IntegerMatrix");
timedReport("ElementPowerInteger", 'proc(m) map(proc(x) x^5 end proc, m) end proc', maxMatrix, "IntegerMatrix");
timedReport("DotInteger8", 'proc(x) x.x end proc', maxMatrix, "IntegerMatrix8");
timedReport("LinearSolveInteger8", 'proc(x) LinearSolve(x, Column(x, 1)) end proc', (1/4)*maxMatrix, "IntegerMatrix8");
timedReport("MatrixPowerInteger8", 'proc(x) x^5 end proc', (1/4)*maxMatrix, "IntegerMatrix8");
timedReport("DetInteger8", 'Determinant', 600, "IntegerMatrix8");
timedReport("TransposeInteger8", 'Transpose', maxMatrix, "IntegerMatrix8");
timedReport("FlattenInteger8", 'proc(m) convert(m, Vector[row]) end proc', maxMatrix, "IntegerMatrix8");
timedReport("MeanInteger8", 'proc(x) add(i, i=x)/Dimension(x) end proc', maxVector, "IntegerVector8");
timedReport("SortInteger8", 'sort', maxVector, "IntegerVector8");
timedReport("EigenvaluesInteger8", 'Eigenvalues', (1/30)*maxMatrix, "IntegerMatrix8");
timedReport("ElementPowerInteger8", 'proc(m) map(proc(x) x^5end proc, m) end proc', maxMatrix, "IntegerMatrix8");
steps:=3;
timedReport("MatrixExpInteger8", 'MatrixExponential', 3, "IntegerMatrix8");
timedReport("MatrixExpInteger", 'MatrixExponential', 3, "IntegerMatrix");
steps:=10;
timedReport("SortCustomInteger", 'proc(data) sort(data, proc(x, y) abs(y)<abs(x) end proc) end proc', (1/25)*maxVector, "IntegerVector");
timedReport("SortCustomInteger8", 'proc(data) sort(data, proc(x, y) abs(y)<abs(x) end proc) end proc', 100000, "IntegerVector8");
timedReport("MatrixRankInteger", 'LinearAlgebra:-Rank', maxMatrix/5, "IntegerMatrix");
timedReport("MatrixRankInteger", 'LinearAlgebra:-Rank', maxMatrix/5, "IntegerMatrix8");
```

Exact functions

```
ExportMatrix("MapleExactFunctions.txt",
        Matrix([[time[real](seq(bernoulli(i), i=1..6000)),
                time[real](seq(fibonacci(i), i=1..500000, 1000)),
                time[real](seq(expand(Zeta(i)), i=0..100000, 10000)),
                time[real](seq(binomial(i^2, i), i=1..10000))
                ]]));
```

Random numbers

```
ExportMatrix("MapleRandomNumbers.txt",
        Matrix([[time[real](makeData("Matrix", 2000)),
                time[real](makeData("MatrixFloat8", 2000)),
                time[real](makeData("IntegerMatrix", 2000)),
                time[real](makeData("IntegerMatrix8", 2000)),
                time[real](makeData("BigNumberMatrix", 2000)),
                time[real](makeData("BigNumberMatrix1000", 2000)),
                time[real](Sample(RandomVariable(Normal(0, 1)), 10^7)),
```

```
            time[real](Sample(RandomVariable(Poisson(4)), 10^7)),
            time[real](Sample(RandomVariable(BinomialDistribution(10, .2)), 10^7))
    ]]));
```

Function evaluation

```
numerictest("ElementaryFunctionsFloat8", fnList, 10*maxVector, "VectorFloat8");
numerictest("ElementaryFunctionsComplex", fnList, maxVector, "ComplexVector");
numerictest("ElementaryFunctionsComplex8", fnList, maxVector, "ComplexVector8");
numerictestinplace("ElementaryFunctionsInPlaceFloat8", fnListSafe, 10*maxVector, "VectorFloat8");
numerictestinplace("ElementaryFunctionsInPlaceComplex", fnList, maxVector, "ComplexVector");
numerictestinplace("ElementaryFunctionsInPlaceComplex8", fnList, maxVector, "ComplexVector8");
numerictestinplace("ElementaryFunctionsInPlace", fnList, 10*maxVector, "Vector");
numerictestelementwise("ElementaryFunctionsElementwiseFloat8", elementwisefnListSafe, 10*maxVector, "VectorFloat8");
numerictestelementwise("ElementaryFunctionsElementwiseComplex", elementwisefnList, maxVector, "ComplexVector");
numerictestelementwise("ElementaryFunctionsElementwiseComplex8", elementwisefnList, maxVector, "ComplexVector8");
numerictestelementwise("ElementaryFunctionsElementwise", elementwisefnList, 10*maxVector, "Vector");
numerictestinplace("ElementaryFunctionsInPlaceSparse", fnList, 10000*maxVector, "SparseVector");
numerictest("ElementaryFunctions", fnList, 10*maxVector, "Vector");
```

High precision

```
highPrecisionTest("ManyDigits", ['Pi', 'sqrt(2)', 'sin(1)+1/(1+sin(1))', 'exp(1)', 'log(5)', 'tan(1+I)'], 1000000);
highPrecisionTest("FewerDigits", ['BesselK(2, 3)', 'GAMMA(11/3)', 'erf(10)'], 5000);
```

50 digit precision operations

```
Digits:=50;
timedReport("DotExtended", 'proc(x) x.x end proc', (1/25)*maxMatrix, "BigNumberMatrix");
timedReport("InverseExtended", 'MatrixInverse', (1/25)*maxMatrix, "BigNumberMatrix");
timedReport("EigenvaluesExtended", 'Eigenvalues', 60, "BigNumberMatrix");
timedReport("LinearSolveExtended", 'proc(x) LinearSolve(x, Column(x, 1)) end proc', (1/25)*maxMatrix, "BigNumberMatrix");
timedReport("CholeskyExtended", 'proc(S) LUDecomposition*(Transpose(S).S,
                method='Cholesky') end proc', (1/20)*maxMatrix, "BigNumberMatrix");
timedReport("MatrixPowerExtended", 'proc(x) x^5 end proc', (1/25)*maxMatrix, "BigNumberMatrix");
timedReport("DetExtended", 'Determinant', (1/10)*maxMatrix, "BigNumberMatrix");
timedReport("TransposeExtended", 'Transpose', maxMatrix, "BigNumberMatrix");
timedReport("FlattenExtended", 'proc(m) convert(m, Vector[row]) end proc', maxMatrix, "BigNumberMatrix");
timedReport("SortExtended", 'sort', maxVector, "BigNumberVector");
timedReport("MeanExtended", 'proc(x) add(i, i=x)/Dimension(x) end proc', (1/10)*maxVector, "BigNumberVector");
timedReport("EigenvectorsExtended", 'Eigenvectors', (1/25)*maxMatrix, "BigNumberMatrix");
timedReport("ElementPowerExtended", 'proc(m) map(proc(x) x^5 end proc, m) end proc', (1/2)*maxMatrix, "BigNumberMatrix");
timedReport("MatrixExpExtended", 'MatrixExponential', 60, "BigNumberMatrix");
timedReport("SortCustomExtended", 'proc(data) sort(data, proc(x, y) abs(y)<abs(x) end proc) end proc', maxVector, "BigNumberVector");
timedReport("MatrixRankExtended", 'LinearAlgebra:-Rank', maxMatrix/10, "BigNumberMatrix");

numerictest("ElementaryFunctionsExtended", fnList, (1/10)*maxVector, "BigNumberVector");
numerictestelementwise("ElementaryFunctionsElementwiseExtended", elementwisefnList, (1/10)*maxVector, "BigNumberVector");
numerictestinplace("ElementaryFunctionsInPlaceExtended", fnList, (1/10)*maxVector, "BigNumberVector");
```

1000 digit precision operations

```
Digits:=1000;
timedReport("DotExtended1000", 'proc(x) x.x end proc', (1/20)*maxMatrix, "BigNumberMatrix1000");
timedReport("InverseExtended1000", 'MatrixInverse', (1/25)*maxMatrix, "BigNumberMatrix1000");
timedReport("EigenvaluesExtended1000", 'Eigenvalues', 60, "BigNumberMatrix1000");
timedReport("LinearSolveExtended1000", 'proc(x) LinearSolve(x, Column(x, 1)) end proc', (1/25)*maxMatrix, "BigNumberMatrix1000");
timedReport("CholeskyExtended1000", 'proc(S) LUDecomposition*(Transpose(S).S, method='Cholesky') end proc',
                (1/20)*maxMatrix, "BigNumberMatrix1000");
timedReport("MatrixPowerExtended1000", 'proc(x) x^5 end proc', (1/25)*maxMatrix, "BigNumberMatrix1000");
timedReport("DetExtended1000", 'Determinant', (1/10)*maxMatrix, "BigNumberMatrix1000");
timedReport("TransposeExtended1000", 'Transpose', maxMatrix, "BigNumberMatrix1000");
timedReport("FlattenExtended1000", 'proc(m) convert(m, Vector[row]) end proc', maxMatrix, "BigNumberMatrix1000");
timedReport("SortExtended1000", 'sort', maxVector, "BigNumberVector1000");
timedReport("MeanExtended1000", 'proc(x) add(i, i=x)/Dimension(x) end proc', (1/10)*maxVector, "BigNumberVector1000");
timedReport("EigenvectorsExtended1000", 'Eigenvectors', (1/25)*maxMatrix, "BigNumberMatrix1000");
```

```
timedReport("ElementPowerExtended1000", 'proc(m) map(proc(x) x^5 end proc, m) end proc', (1/2)*maxMatrix, "BigNumberMatrix1000");
timedReport("MatrixExpExtended1000", 'MatrixExponential', 60, "BigNumberMatrix1000");
timedReport("SortCustomExtended1000", 'proc(data)sort(data, proc(x, y)abs(y)<abs(x) end proc) end proc',
                    maxVector, "BigNumberVector1000");
timedReport("MatrixRankExtended1000", 'LinearAlgebra:-Rank', maxMatrix/20, "BigNumberMatrix1000");
```