Comparison of Numerical Performance of *Mathematica* 9 and Maple 16

Technical Communication & Strategy Group, Wolfram Research

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 505 tasks, covering different operations on different types and sizes of data, *Mathematica* was faster in 500 cases. The median difference measured found *Mathematica* to be more than 38 times faster than Maple.

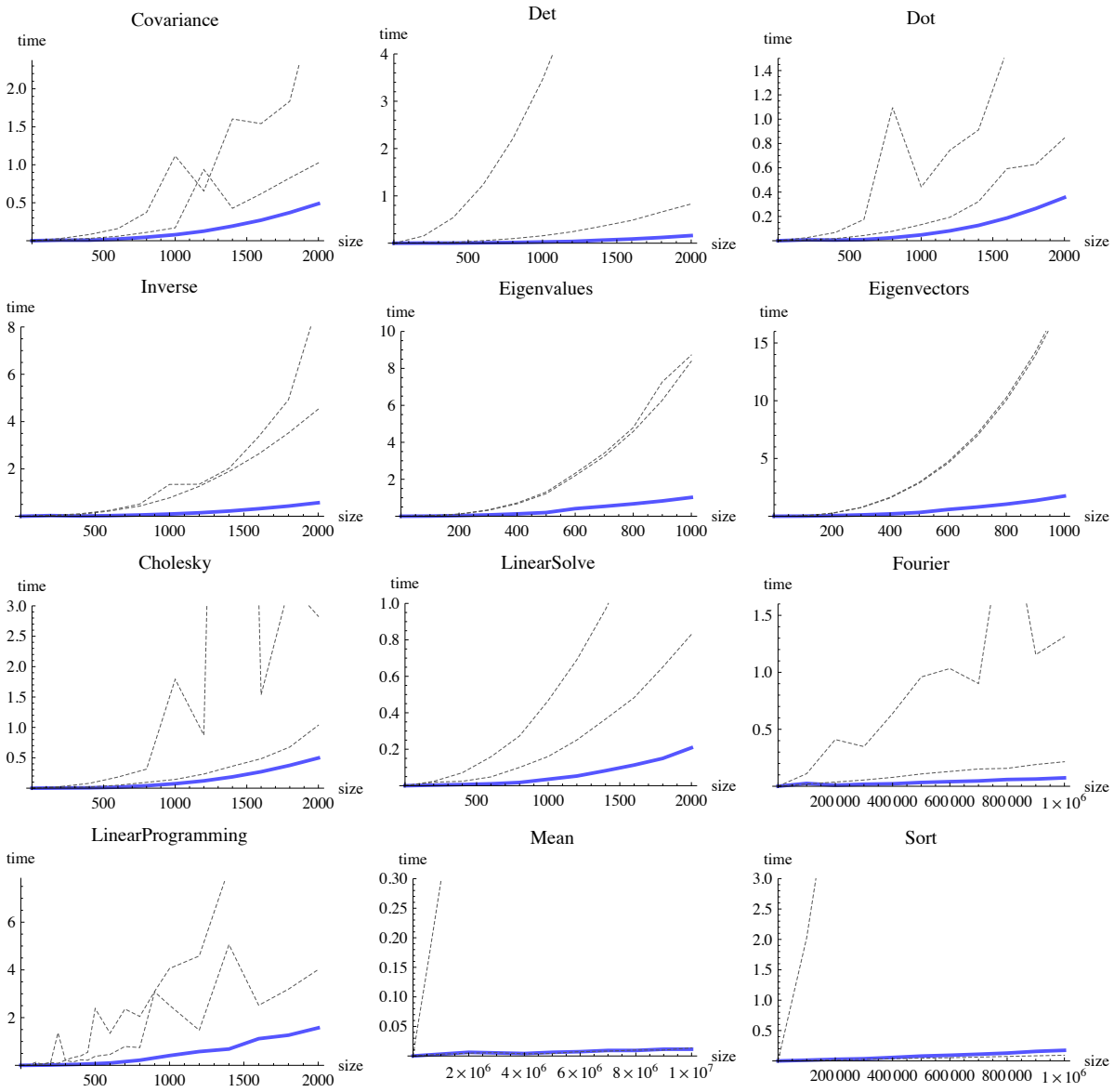
Category	Tests	Median *Mathematica* speed
Real data operations (Default data type)	18	26 times faster
Real data operations (Manual type override)	18	4.5 times faster
Complex number data operations	15	6 times faster
Complex number data operations (Manual type override)	15	5 times faster
Integer data operations	13	44 times faster
Integer data operations (Manual type override)	12	44 times faster
Sparse real data operations	7	13,512 times faster
Extended precision data operations (50 digits)	15	10 times faster
Extended precision data operations (1000 digits)	15	10 times faster
Random number generation	9	38 times faster
Elementary & special functions	75	2845 times faster
Elementary & special functions (Manual type override)	55	318 times faster
Complex elementary & special functions	75	1066 times faster
Complex elementary & special functions (Manual type override)	75	1066 times faster
Elementary & special functions (50 digits)	75	18 times faster
High−precision function evaluation	9	58 times faster
Exact functions	4	55 times faster
GPU use	1	3 times faster
Total	506	38 times faster

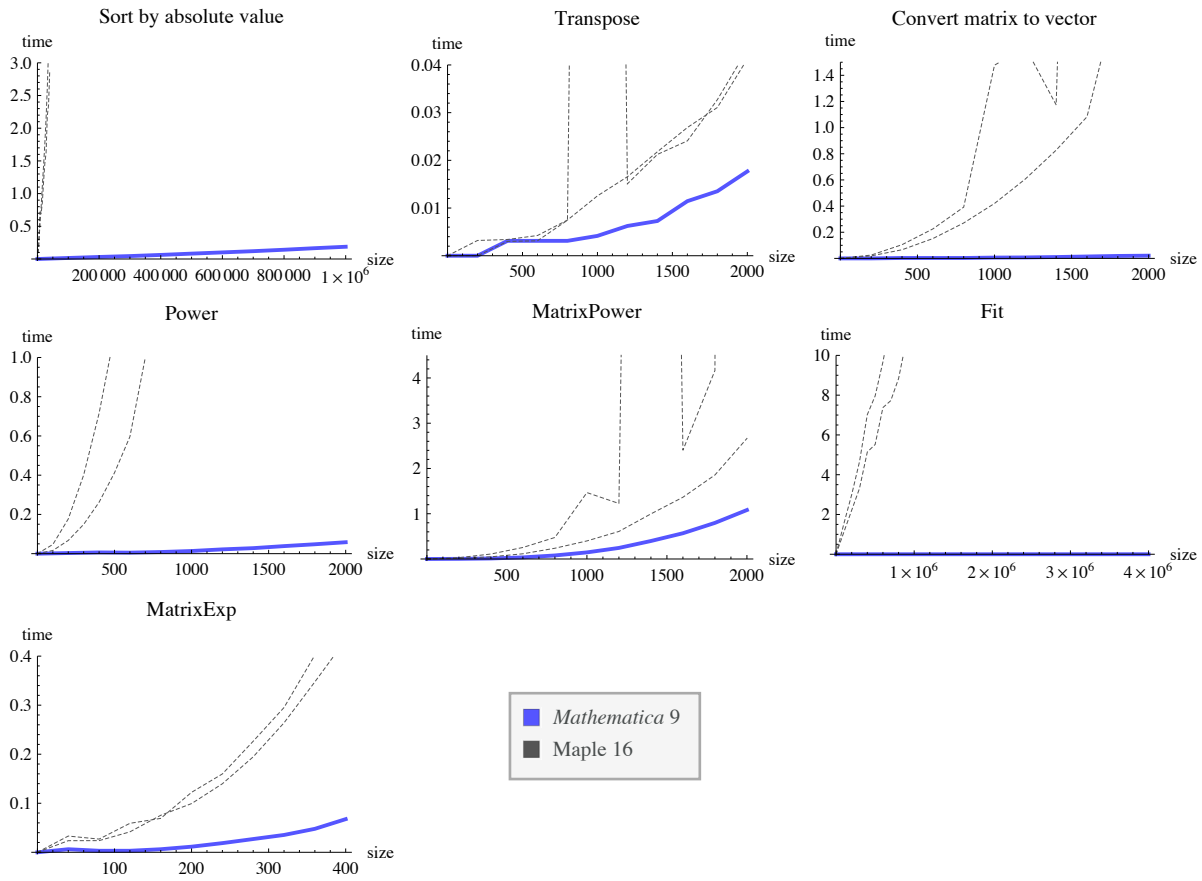
Tests were performed using Windows 7 64-bit with 3.07 GHz quad-core Intel i7 processors (W3580) with 20 GB of RAM. CUDA tests used a Tesla C2050/C2070 GPU with 448 cores and 2.5 GB of RAM.

Machine-Precision Real Linear Algebra

To achieve the very best performance in Maple, you must often use manual type control. This requires declaring the 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 4.5 times faster than Maple's inflexible (float[8]) data type and 26 times faster than its default type.





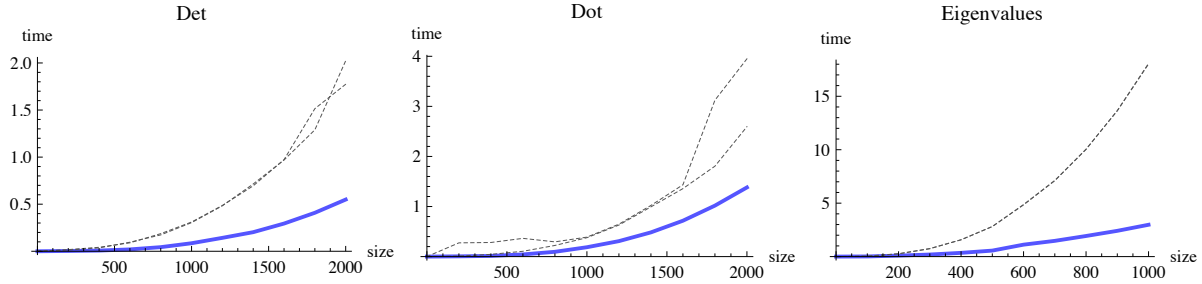
Implementation Notes

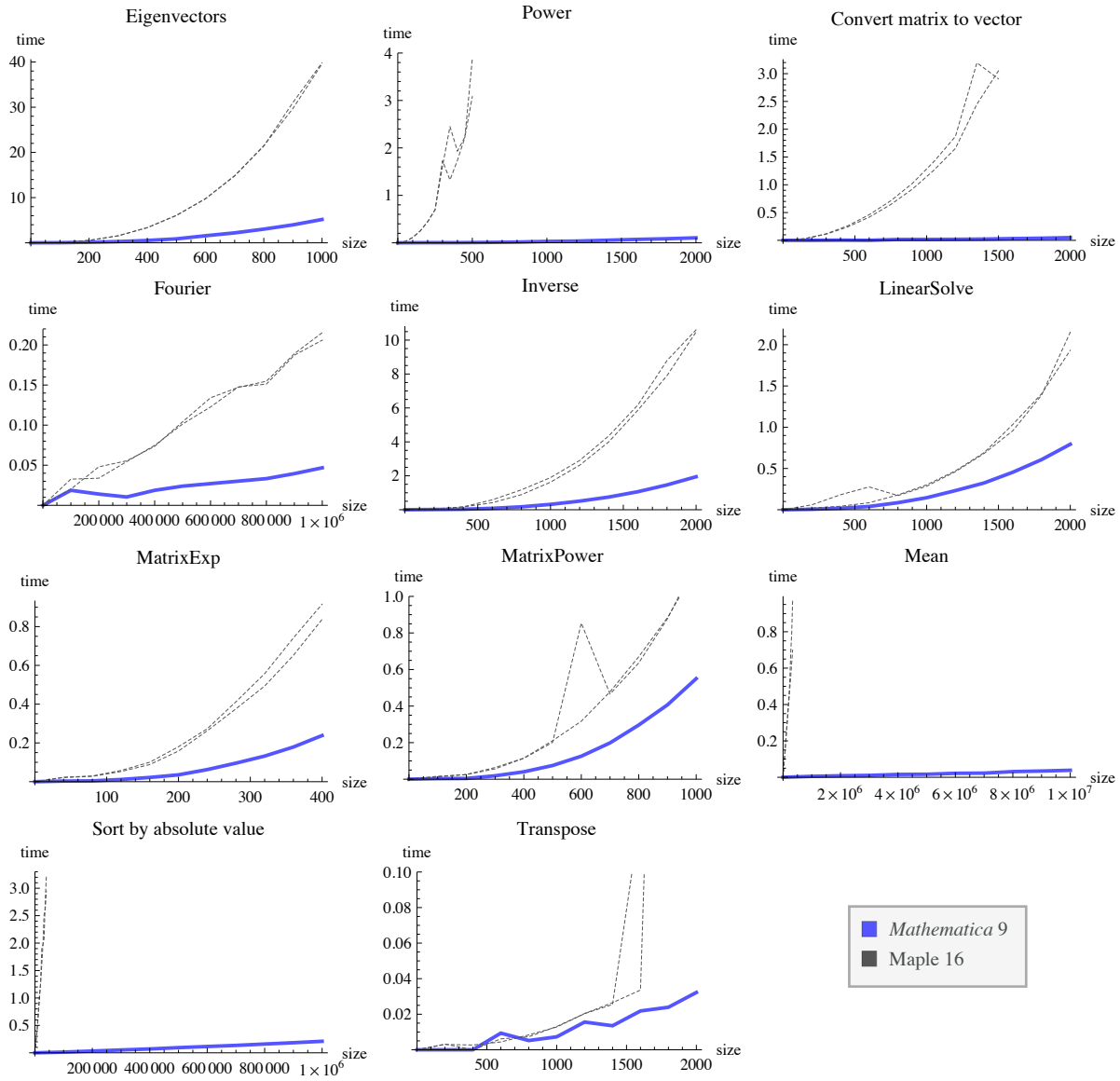
All tests were performed using the documented commands in each system.

MapleSoft feedback has claimed that there are faster methods for converting a matrix to a vector than the documented command "convert(data,Vector)", but they have not provided code or an explanation why the documented command does not use such methods.

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.





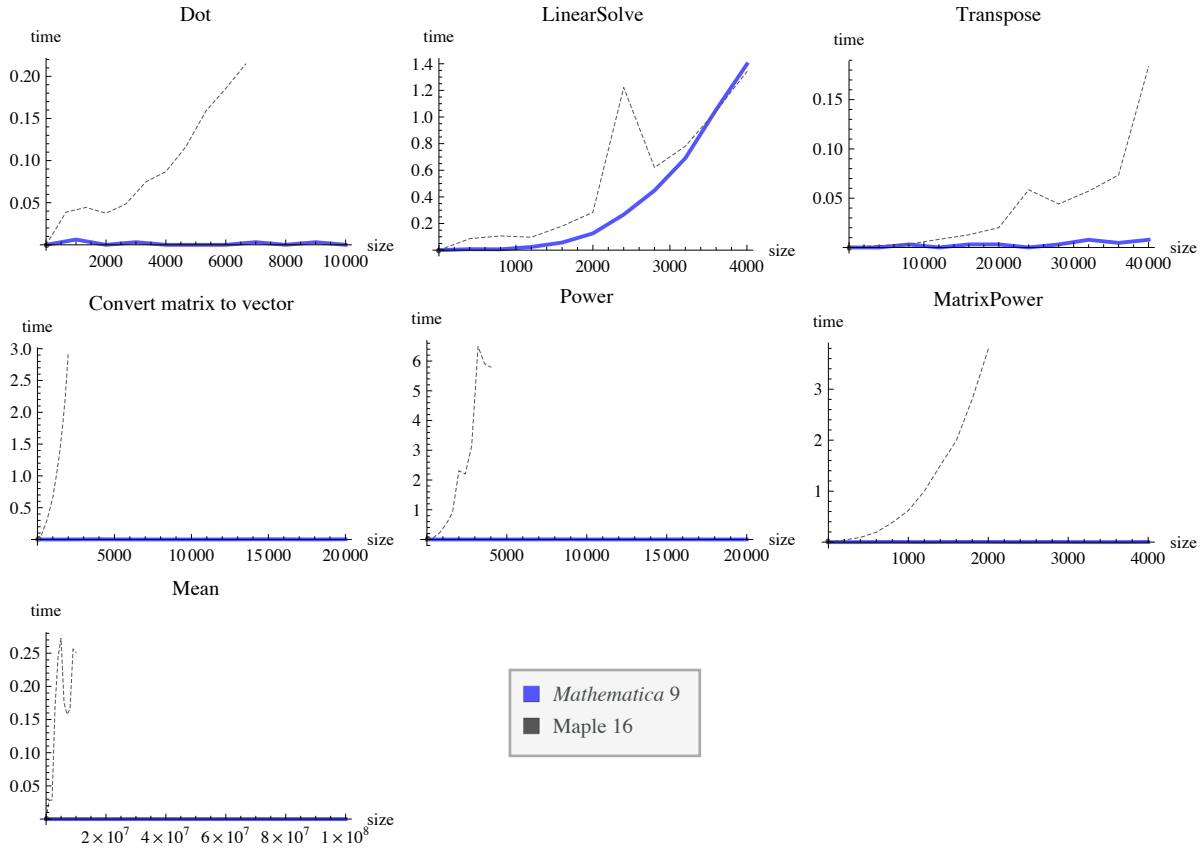
Implementation Notes

Functions in Maple's "Statistics" packages are limited to real numbers only. 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, the Maple implementation of this important concept seems to be limited only to the data representation and not the accompanying sparse algorithms. The difference in performance was so great that Maple was unable to perform many problems to a large enough scale for sensible comparison. In most of these plots, the *Mathematica* timings are too small to be visible on the scale required to plot the Maple timings.



Implementation Notes

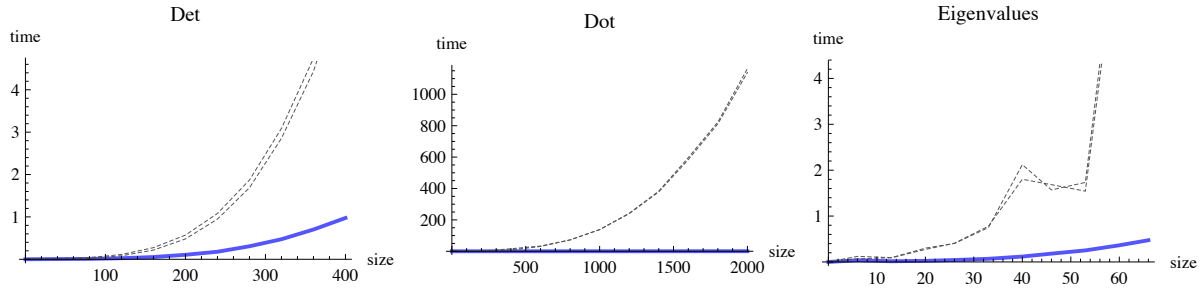
Tests for which neither system has sparse methods have been omitted, since these would be a repeat of the dense tests above. "Sort" and "Sort by absolute value" tests are omitted, as Maple crashed persistently.

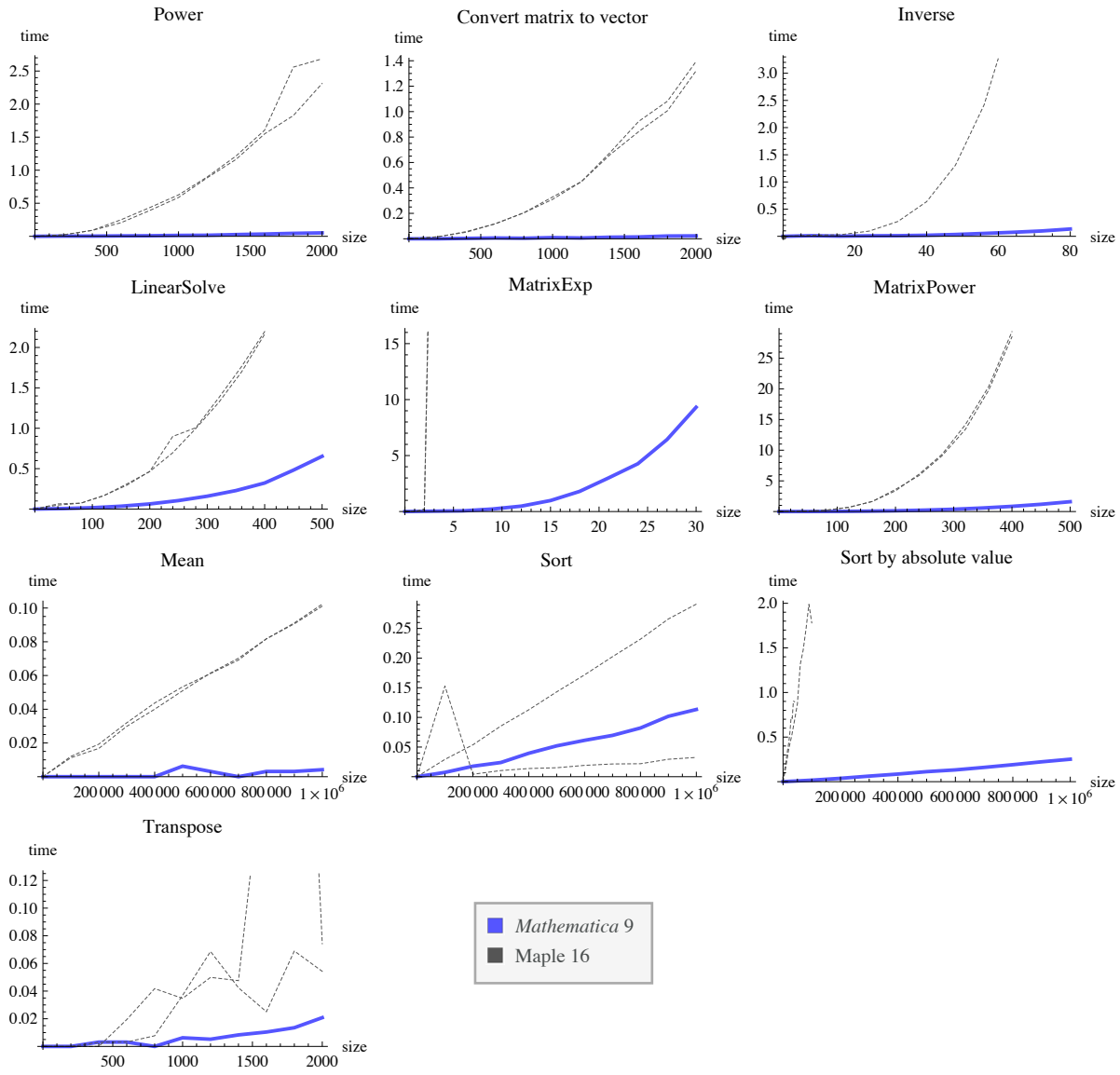
Integer Linear Algebra

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 2800 times faster.

Maple users can choose between a general data type or integer[8], but there appears to be no benefit in using the more limiting data type.

The median difference put *Mathematica* at 44 times faster.





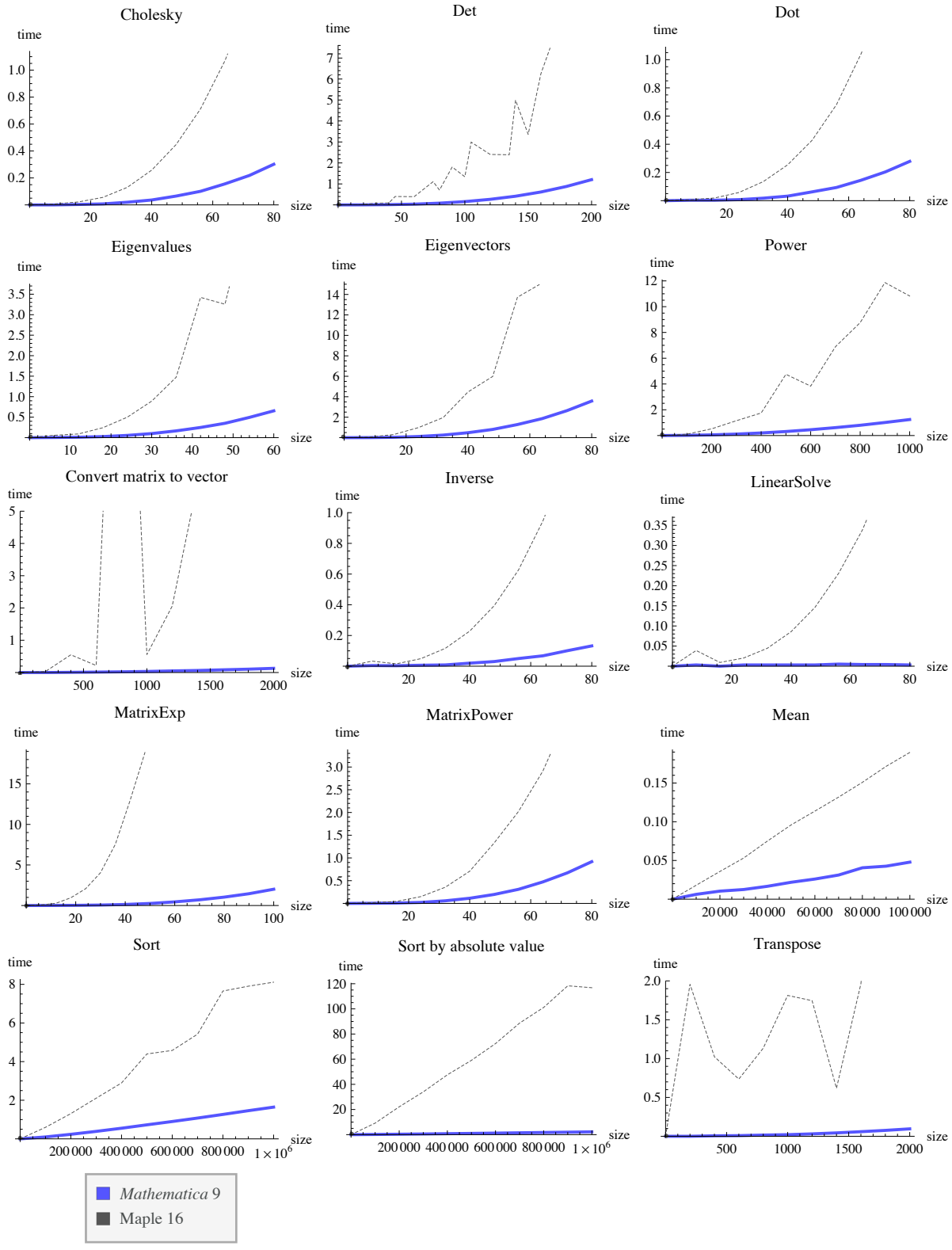
Implementation Notes

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

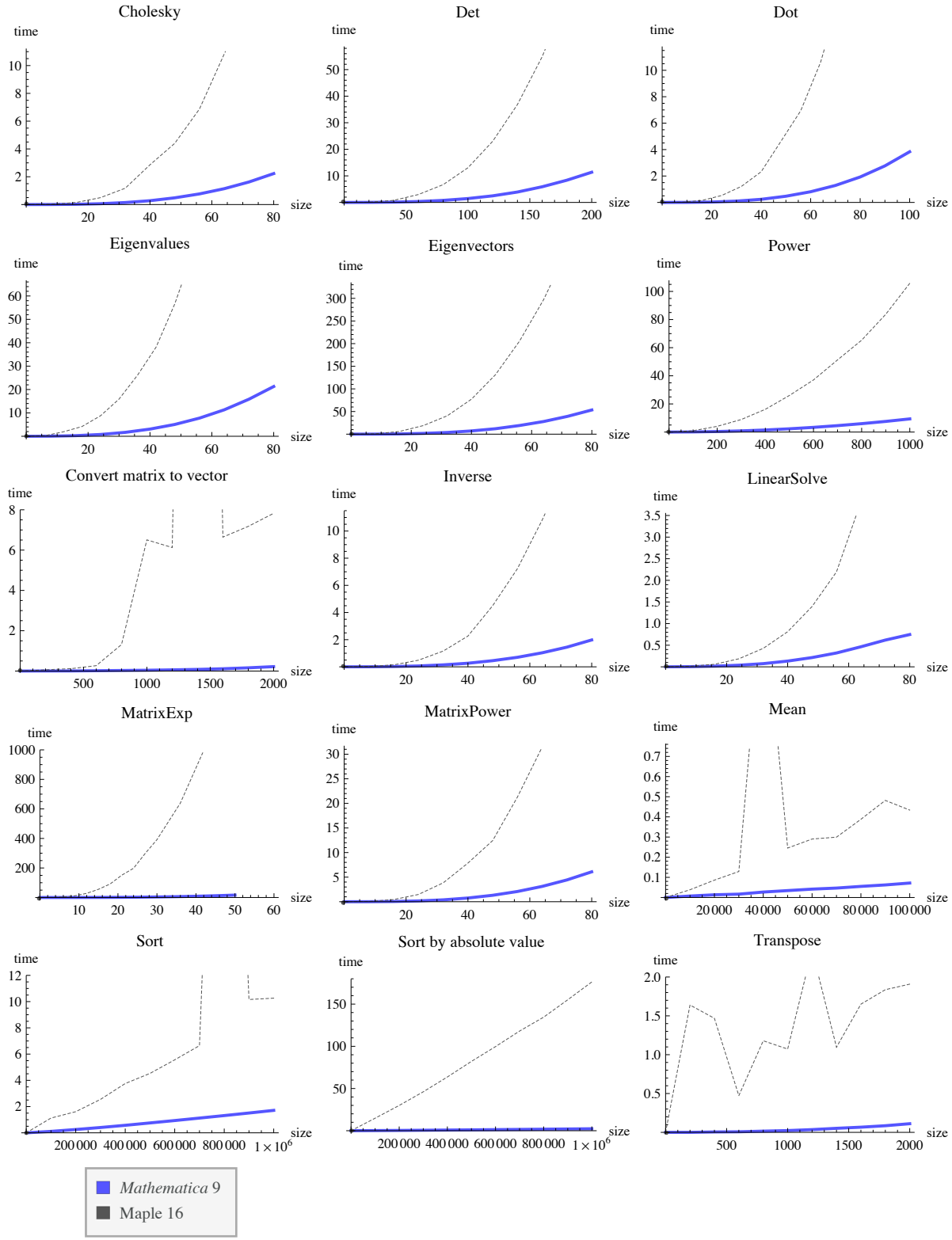
Maple cannot evaluate the Inverse task using the integer[8] type, so only default data type results are shown for this test.

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 10 times faster for 50-decimal-place matrix computations.


Cholesky
Det
Dot
Eigenvalues
Eigenvectors
Power
Convert matrix to vector
Inverse
LinearSolve
MatrixExp
MatrixPower
Mean
Sort
Sort by absolute value
Transpose
time
size
Mathematica 9
Maple 16

Similar performance differences are seen when calculating at 1000-digit precision.

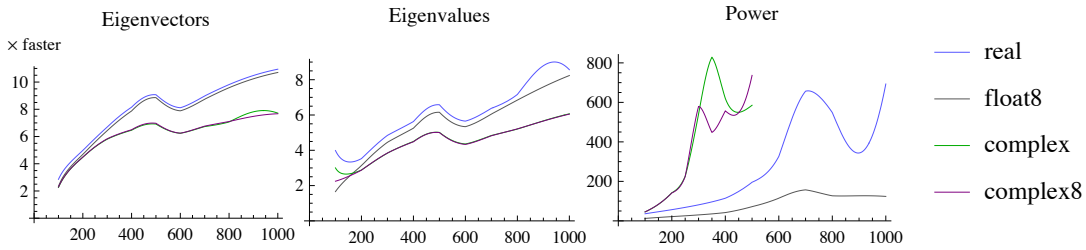


Implementation Notes

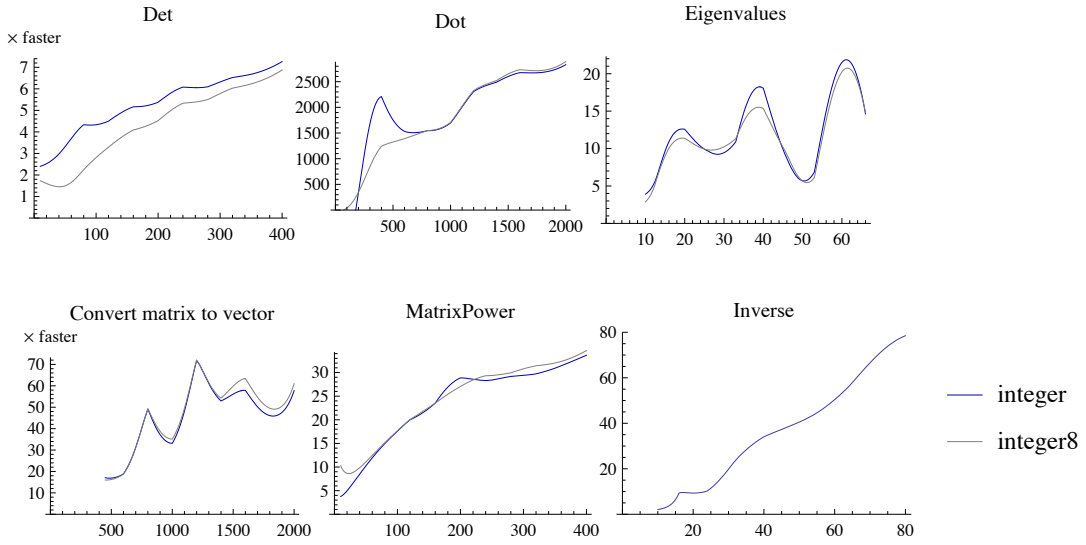
Many Maple functions are limited to machine-precision numbers. For this reason, tests for Fit and Fourier have been omitted, and Mean has been implemented using "add".

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.



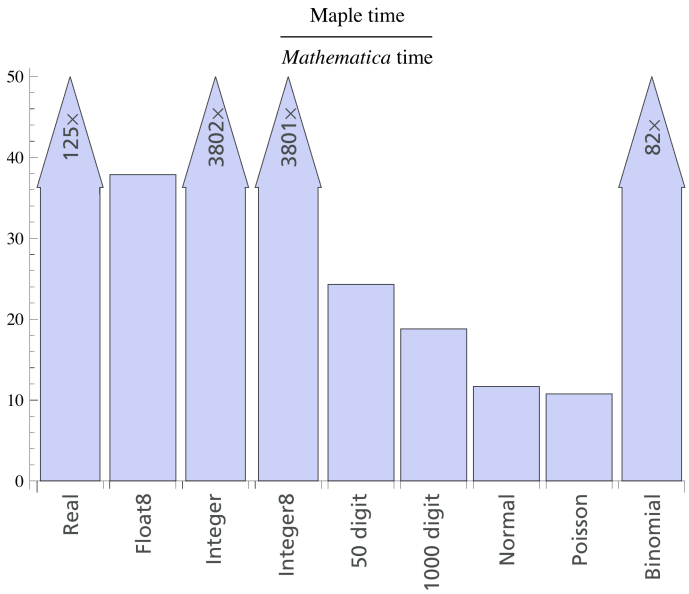
Poor algorithm scaling seems to be a particular problem for Maple's handling of integer matrices. More operations show a trend in favor of *Mathematica* as problem size increases.



Random Numbers

Median performance for random number generation shows *Mathematica* to be 38 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

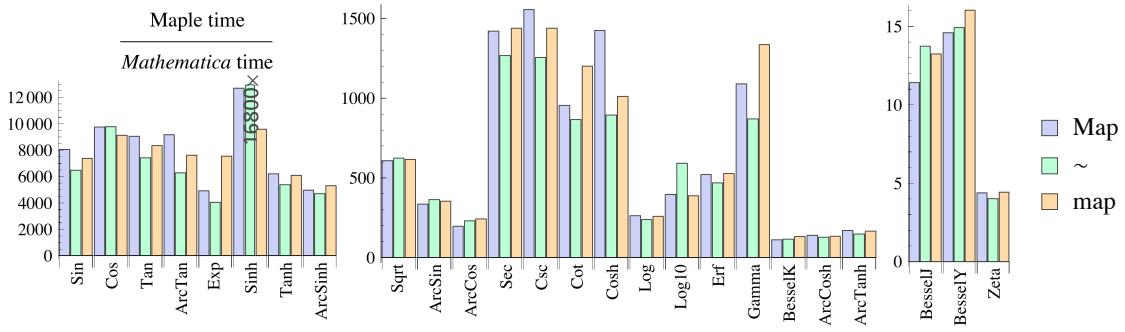
Documented Maple commands were used in all cases. Maple's default type for reals and its special "Float8" data generation were both compared to the default data type in *Mathematica*. Maple's default type for integers and its special "Integer8" data generation were both compared to the default data type 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 functions over data and a choice of data types. Different circumstances require different choices and often come at the cost of 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 default data structure and Map command.

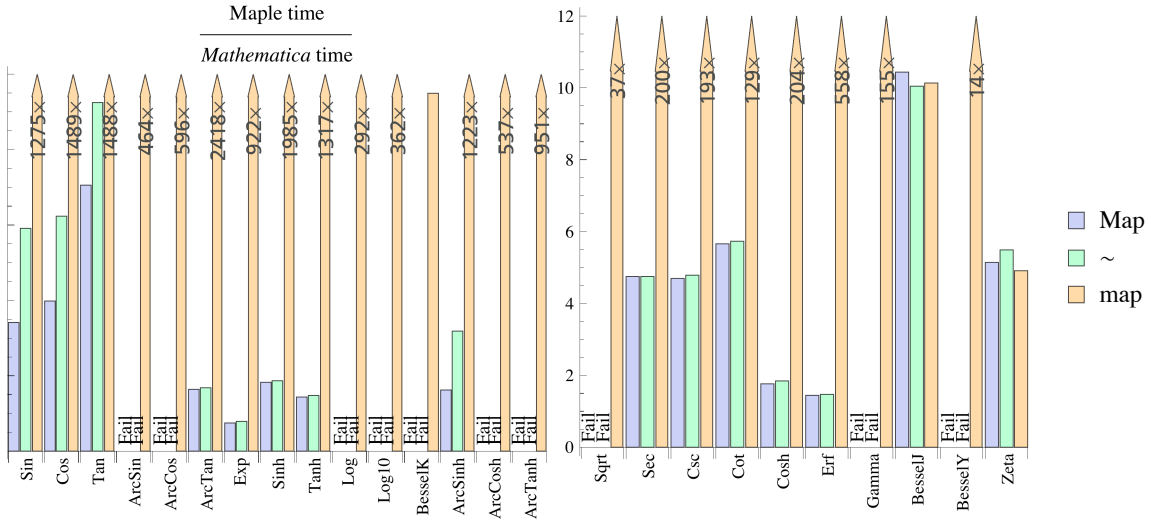
Using the default data type to represent data (as most users would), there is no appreciable difference between the Maple commands "map", "Map" and "~", (though "Map" is a destructive operation, so extra effort is needed to copy the data, if the original data is to be preserved).

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

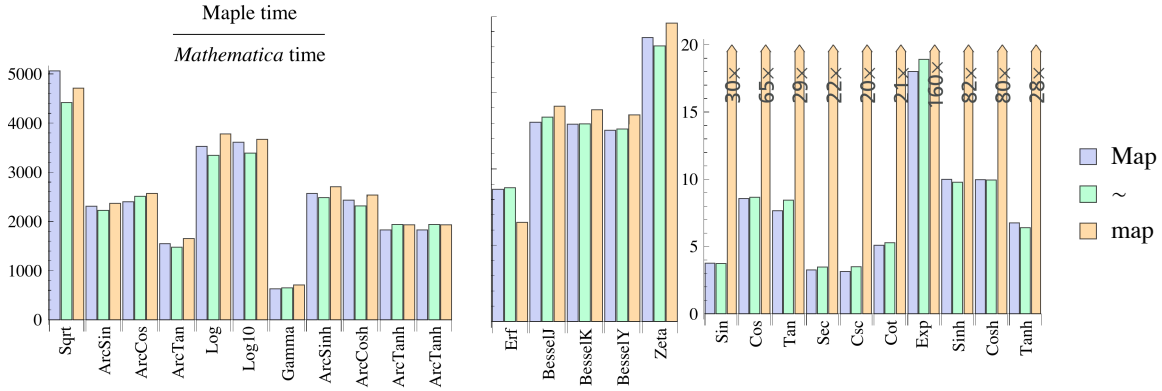


Better performance can be achieved in Maple by manually specifying (or converting) data to the "float 8" data type. However, this comes at considerable cost to flexibility. First you must ensure that all the input data is machine reals. Code will fail if you attempt to store a value that is complex,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

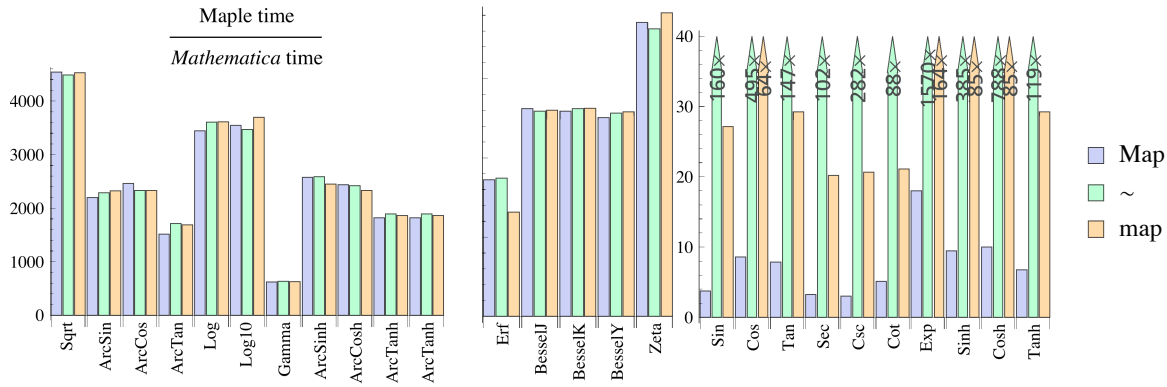
Using the input dataset in the interval [-10,10], Maple's "Map" and "~" commands fail, for obvious reasons, for 11 of the tests, and "map", while more robust, is typically more than 100 times slower than *Mathematica*.



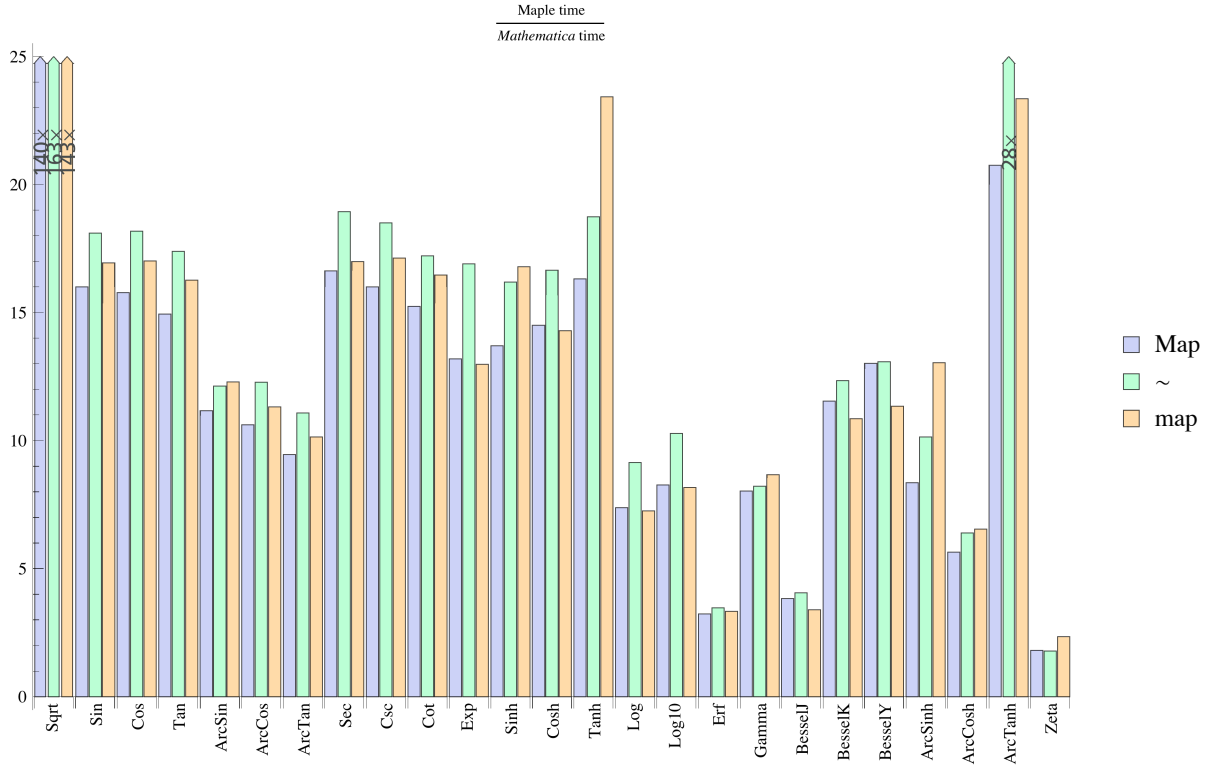
For most computations over complex numbers, using Maple's default data type, there is no significant performance difference whether you use "map", "Map", or "~" . For a few of the simplest functions, there is a significant penalty for using "map".



Surprisingly, there appears to be no significant gain by enforcing the "complex8" data type (despite the loss of flexibility and the risk of failure should any computations return values under "Map" or "~" that cannot be stored in "complex8"). And for a small set of the simplest functions, there is a significant performance penalty when combined with "~".



When working in 50-decimal-place extended arithmetic, it makes little difference how you apply functions in Maple: they are fairly consistently 10–15 times slower than *Mathematica*.



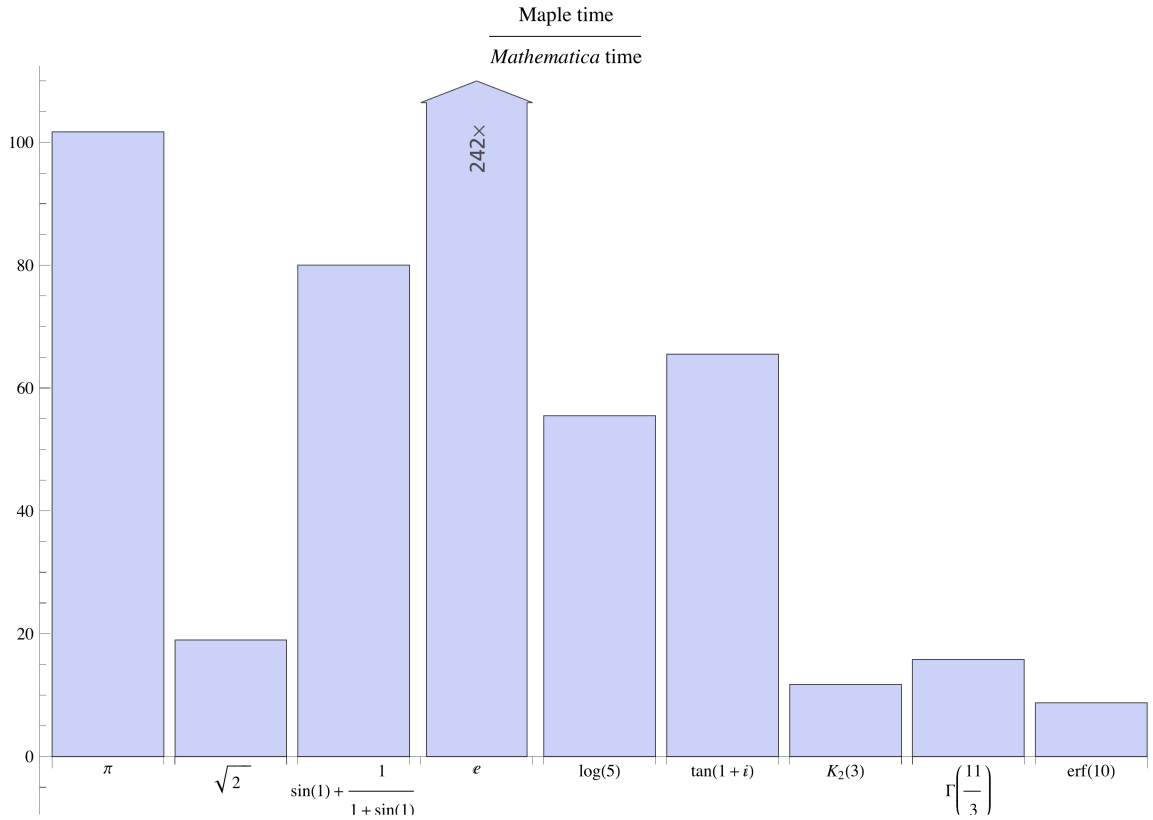
Getting the best performance out of Maple requires significant system knowledge and carefully chosen manual intervention. In contrast, all *Mathematica* computations were performed using the same function and the same default data type and still outperformed Maple in every single test.

Implementation Notes

Maple 16 was unable to perform many of the tests in this section (even where Maple 15 had managed in less memory in Version 1 of the test suite), but reducing the test size often resulted in *Mathematica* computations that were too fast to measure. For those tests, the Maple test has been scaled back to 50% or 10% of the size of the *Mathematica* test, and linear time scaling has been assumed.

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, between 9 and 242 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	Median *Mathematica* speed
BernoulliB	6000	2.7 times faster
Fibonacci	500 000	12 times faster
HarmonicNumber	10 000	Maple fails for n > 50
Zeta	100 000	21 times faster. Maple crashes for n ≥ 60,000
Binomial	10^{14}	99 times faster

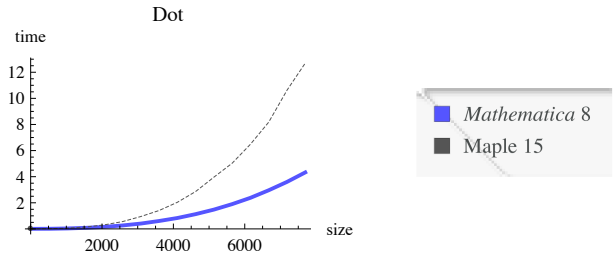
Implementation Notes

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 (W3580) with 20 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 16.01 and *Mathematica* 9.0.0 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, 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 failed with "Unable to allocate memory" errors 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.

Tests where Maple is unable to perform the calculation have not been used for calculating median values.

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 $a\,x^2 + b\,x$, 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, including random data generated, takes approximately 300 minutes in *Mathematica* and several days in Maple.

Revision Notes

Significant changes since Version 1 of this benchmark:

- Increased size of test suite from 164 tests to 506
- Tested Maple "Map" and "In place" (~) commands, 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
- Removed faulty LinearProgramming tests (test was not the same task in each system)
- Removed programming tests that did not relate to numeric computation
- Used more modern test computer with more memory

Test Code

Mathematica test suite source code

Test Code

```
$HistoryLength = 0;
SetDirectory["C:\\BenchmarkData"];
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 <> ".dat", Table[{Floor[t],
      Mean[Table[
        data = makeData[type, Floor[t]];
        AbsoluteTiming[fn[data]][[1]], {repeats}]]}
     , {t, hi/steps, hi, hi/steps}]]];

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

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

evaluateTest[path_, expr_Hold] :=
 Export["Mathematica" <> ToString[$VersionNumber] <> path <> ".dat",
  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};
NumericTest["ElementaryFunctions", fnList, 10 maxVector , "RealVector"];
NumericTest["ElementaryFunctionsComplex", fnList, maxVector , "ComplexVector"];
NumericTest["ElementaryFunctionsExtended",
  fnList, maxVector / 10, "ExtendedVector"];
```

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["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"];
```

Complex Matrices 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 / 5, "ComplexMatrix"];
```

Sparse Operations

```
timedReport["DotSparse", #.# &, maxSparseMatrix / 2, "SparseMatrix"];
Quiet@timedReport["LinearSolveSparse",
   LinearSolve[#, #[[1]]] &, maxSparseMatrix / 5, "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] &, maxSparseMatrix / 5, "SparseMatrix"];
(*timedReport["MatrixExpSparse", MatrixExp, maxSparseMatrix, "SparseMatrix"];*)
timedReport["MovingAverageSparse",
  MovingAverage[#, 10] &, 10 maxVector, "SparseVector"];
(*timedReport["SortSparse",Sort,10 maxVector,"SparseVector"];
timedReport["SortCustomSparse",SortBy[#,Abs]&,10 maxVector,"SparseVector"];*)
(*Crashes Maple*)
```

Integer 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, maxMatrix / 5 , "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, 30, "IntegerMatrix"];
(*timedReport["MovingAverageInteger", MovingAverage[#, 10] &,
  maxVector, "IntegerVector"];*)(*Maple can't do this in integers*)
```

50-Digit-Precision Operations

```
timedReport["DotExtended", #.# &, maxMatrix / 25 , "ExtendedMatrix"];
timedReport["InverseExtended", Inverse, maxMatrix / 25, "ExtendedMatrix"];
timedReport["CholeskyExtended",
  CholeskyDecomposition[Transpose[#].#] &, maxMatrix / 25, "ExtendedMatrix"];
timedReport["EigenvaluesExtended", Eigenvalues, 60, "ExtendedMatrix"];
timedReport["EigenvectorsExtended", Eigenvectors, 80, "ExtendedMatrix"];
timedReport["ElementPowerExtended", #^5 &, maxMatrix / 2, "ExtendedMatrix"];
timedReport["LinearSolveExtended",
  LinearSolve[#, #[[1]]] &, maxMatrix / 25, "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, maxVector, "ExtendedVector"];
timedReport["SortCustomExtended", SortBy[#, Abs] &, maxVector, "ExtendedVector"];
timedReport["EigenvectorsExtended", Eigenvectors, maxMatrix / 25, "ExtendedMatrix"];
timedReport["MatrixPowerExtended",
  MatrixPower[#, 5] &, maxMatrix / 25, "ExtendedMatrix"];
timedReport["MatrixExpExtended", MatrixExp, 100, "ExtendedMatrix"];
```

1000-Digit-Precision Operations

```
timedReport["DotExtended1000", #.# &, maxMatrix / 20 , "ExtendedMatrix1000"];
timedReport["InverseExtended1000", Inverse, maxMatrix / 25, "ExtendedMatrix1000"];
timedReport["CholeskyExtended1000",
  CholeskyDecomposition[Transpose[#].#] &, maxMatrix / 25, "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["FourierExtended1000",
  Fourier,maxVector/100,"ExtendedVector1000"];*)
(*Maple converts to floats*)
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, 50, "ExtendedMatrix1000"];
(*timedReport["MovingAverageExtended1000",MovingAverage[#,10]&,
  maxVector,"ExtendedVector1000"];*)(*Maple converts to floats*)
```

Integer 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,0,100000,10000}], Maple crashes around 60,000*)
   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 Source Code

```
# Initialize packages
with(LinearAlgebra):
with(DiscreteTransforms):
with(combinat, fibonacci):
with(Statistics):
with(RandomTools[MersenneTwister]):
with(GraphTheory):
with(RandomGraphs):
with(Optimization):
with(RandomTools):
with(stats):
currentdir("C:\\MapleBenchmarks"):

steps := 20:
repeats := 10:
maxVector := 10^6:
maxMatrix := 2000:
maxSparseMatrix := 20000:

# Tools
AbsoluteTiming := proc (expr) local temp:
temp := time[real]():
eval(expr):
time[real]()-temp end proc:
makeData := proc (type, size) local dat, i:
if type = "Vector" then dat := RandomVector[row](size, generator = 0. ..
```

```
1.0)

elif type = "Matrix" then dat := RandomMatrix(size, size, generator = 0.
.. 1.0)
elif type = "VectorFloat8" then dat := RandomVector[row](size, generator
= 0. .. 1.0, datatype = float[8])

elif type = "MatrixFloat8" then dat := RandomMatrix(size, size,
generator = 0. .. 1.0, 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))] := GenerateFloat() 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))] := GenerateFloat() end do
elif type = "SparseVector" then dat := Vector(size, storage = sparse):
for i to floor((1/10000)*size) do
dat[RandomTools[Generate](integer(range = 1 .. size))] :=
GenerateFloat() end do

elif type = "IntegerMatrix" then dat := RandomMatrix(size, size,
generator = RandomInteger)
elif type = "IntegerVector" then dat := RandomVector[row](size,
generator = RandomInteger)

elif type = "IntegerMatrix8" then dat := RandomMatrix(size, size,
generator = RandomInteger, datatype = integer[8])
elif type = "IntegerVector8" then dat := RandomVector[row](size,
generator = RandomInteger, datatype = integer[8])

elif type = "BigNumberMatrix" then dat := RandomMatrix(size, size,
generator = (proc (x) GenerateFloat(digits = 50) end proc))
elif type = "BigNumberVector" then dat := RandomVector[row](size,
generator = (proc (x) GenerateFloat(digits = 50) end proc))
elif type = "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) end if:
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+AbsoluteTiming('expr(data)') end do:
totaltime/repeats end proc:

timedReport := proc (filename, expr, hi, type) ExportMatrix(cat("Maple",
filename, ".dat"), 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, time:
data := makeData(type, n):
AbsoluteTiming('map(fn, data)') end proc:

numerictest := proc (file, fns, n, type) ExportMatrix(cat("Maple", file,
".dat"), Matrix([seq(numericfntest(i, n, type), `in`(i, fns))])) end proc:
numericfntestinplace := proc (fn, n, type) local data, time:
data := makeData(type, n):
```

```
AbsoluteTiming('Map(fn, data)') end proc:

highPrecisionTest := proc (file, fns, n) ExportMatrix(cat("Maple", file,
".dat"), Matrix([seq(AbsoluteTiming('evalf(i, n)'), `in`(i, fns))])) end
proc:
RandomInteger := proc (x, y) RandomTools[Generate](integer(range = 1 ..
100)) end proc:

# Sparse ops
timedReport("DotSparse", 'proc (x) x.x end proc',
(1/3)*maxSparseMatrix, "SparseMatrix"):

timedReport("LinearSolveSparse", 'proc (x) LinearSolve(x, Column(x, 1))
end proc', (1/5)*maxSparseMatrix, "MediumSparseMatrix"):

timedReport("LinearProgrammingSparse", 'proc (x) LPSolve(Column(x, 1),
[x, Column(x, 2)], [0, infinity]) end proc', (1/10)*maxSparseMatrix,
"MediumSparseMatrix"):
timedReport("TransposeSparse", 'Transpose', 2*maxSparseMatrix,
"SparseMatrix"):
timedReport("FlattenSparse", 'proc (m) convert(m, Vector[row]) end
proc', maxMatrix, "SparseMatrix"):

timedReport("MatrixPowerSparse", 'proc (x) x^5 end proc', maxMatrix,
"SparseMatrix"):

timedReport("MeanSparse", 'Mean', 10*maxVector, "SparseVector"):
timedReport("ElementPowerSparse", 'proc (m) map(proc (x) x^5 end proc,
m) end proc', (1/5)*maxSparseMatrix, "SparseMatrix"):

# Real (Default) Matrix ops
timedReport("FourierReal", 'FourierTransform', maxVector, "Vector"):
timedReport("SortReal", 'sort', (1/10)*maxVector, "Vector"):
timedReport("MeanReal", 'Mean', 2*maxVector, "Vector"):
timedReport("DotReal", 'proc (x) x.x end 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^5 end proc', maxMatrix,
"Matrix"):
timedReport("DetReal", 'Determinant', maxMatrix, "Matrix"):
timedReport("TransposeReal", 'Transpose', maxMatrix, "Matrix"):
timedReport("FlattenReal", 'proc (m) convert(m, Vector[row]) end proc',
3*maxMatrix*(1/4), "Matrix"):
timedReport("EigenvaluesReal", 'Eigenvalues', (1/2)*maxMatrix, "Matrix"):
timedReport("EigenvectorsReal", 'Eigenvectors', (1/2)*maxMatrix, "Matrix"):
timedReport("LinearProgrammingReal", 'proc (x) LPSolve(Column(x, 1),
[x, Column(x, 2)], [0, infinity]) end proc', (1/10)*maxMatrix, "Matrix"):
timedReport("ElementPowerReal", 'proc (m) map(proc (x) x^5 end proc,
m) end proc', (1/2)*maxMatrix, "Matrix"):

# Float[8] Matrix ops
timedReport("FourierFloat8", 'FourierTransform', maxVector, "VectorFloat8"):
timedReport("SortFloat8", 'sort', maxVector, "VectorFloat8"):
timedReport("MeanFloat8", 'Mean', 10*maxVector, "VectorFloat8"):
timedReport("DotFloat8", 'proc (x) x.x end 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^5 end proc', maxMatrix,
"MatrixFloat8"):
timedReport("DetFloat8", 'Determinant', maxMatrix, "MatrixFloat8"):
timedReport("TransposeFloat8", 'Transpose', maxMatrix, "MatrixFloat8"):
timedReport("FlattenFloat8", 'proc (m) convert(m, Vector[row]) end
proc', 3*maxMatrix*(1/4), "MatrixFloat8"):
timedReport("EigenvaluesFloat8", 'Eigenvalues', (1/2)*maxMatrix,
"MatrixFloat8"):
timedReport("EigenvectorsFloat8", 'Eigenvectors', (1/2)*maxMatrix,
"MatrixFloat8"):
timedReport("LinearProgrammingFloat8", 'proc (x) LPSolve(Column(x, 1),
[x, Column(x, 2)], [0, infinity]) end proc', (1/10)*maxMatrix,
"MatrixFloat8"):
timedReport("ElementPowerFloat8", 'proc (m) map(proc (x) x^5 end proc,
m) end proc', (1/2)*maxMatrix, "MatrixFloat8"):

# Complex
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.x end 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^5 end 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', 3*maxMatrix*(1/4), "ComplexMatrix"):
timedReport("ElementPowerComplex", 'proc (m) map(proc (x) x^5 end
proc, m) end proc', (1/4)*maxMatrix, "ComplexMatrix"):

# Complex 8
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.x end 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^5 end 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^5 end
proc, m) end proc', (1/4)*maxMatrix, "ComplexMatrix8"):

# Integer matrix ops
# type=Anything
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', (1/5)*maxMatrix, "IntegerMatrix"):
timedReport("MatrixPowerInteger", 'proc (x) x^5 end proc',
(1/5)*maxMatrix, "IntegerMatrix"):


timedReport("DetInteger", 'Determinant', (1/5)*maxMatrix, "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"):

# type=integer[8]
timedReport("DotInteger8", 'proc (x) x.x end proc', maxMatrix,
"IntegerMatrix8"):
timedReport("LinearSolveInteger8", 'proc (x) LinearSolve(x, Column(x,
1)) end proc', (1/5)*maxMatrix, "IntegerMatrix8"):


timedReport("DetInteger8", 'Determinant', (1/5)*maxMatrix,
"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^5 end
proc, m) end proc', maxMatrix, "IntegerMatrix8"):

# Programming
dotest := proc (n) local a, x, y:
a := 1.:
for x to n do for y to n do a := a+`if`(y < x, x, y) end do end do:
a end proc:
```

```
datafntest := proc (x) map(proc (val) `if`(0 < val, val^2, val^4) end
proc, x) end proc:

ExportMatrix("MapleProgramming.dat",
Matrix([AbsoluteTiming('dotest(5*maxMatrix)'),
AbsoluteTiming('datafntest(makeData("Vector", 10*maxVector, float[8]))')])):

# 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)'], 5000):


# Function evaluation
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]:

numerictest("ElementaryFunctions", fnList, 10*maxVector, "Vector"):
numerictest("ElementaryFunctionsComplex", fnList, maxVector,
"ComplexVector"):

# High precision matrix ops
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',
(1/25)*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("FourierExtended", 'FourierTransform', (1/100)*maxVector,
"BigNumberVector"):
timedReport("SortExtended", 'sort', (1/10)*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"):

numerictest("ElementaryFunctionsExtended", fnList, (1/10)*maxVector,
"BigNumberVector"):
timedReport("ElementPowerExtended", 'proc (m) map(proc (x) x^5 end
proc, m) end proc', (1/2)*maxMatrix, "BigNumberMatrix"):

#

with (LinearAlgebra);
with (DiscreteTransforms);
with (combinat, fibonacci);
```

```
with (Statistics);
with (RandomTools[MersenneTwister]);
with (GraphTheory);
with (RandomGraphs);
with (Optimization);
with (RandomTools);
with (stats);
with (combinat);

currentdir ("C:/BenchmarkData");
steps := 10;
repeats := 5;
maxVector := 10^6;
maxMatrix := 2000;
maxSparseMatrix := 20000;
AbsoluteTiming := proc (expr) local temp, dat;
temp := time[real] ();
eval (expr);
time[real] () - temp end proc;

makeData := proc (type, size) local dat, i;
if type = "Vector" then dat := RandomVector[row] (size, generator = -10.0 .. 10.0)
elif type = "Matrix" then dat := RandomMatrix (size, size, generator = -10.0 .. 10.0)
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 elif type = "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 := RandomMatrix (size, size, generator = RandomInteger) elif type = "IntegerVector" then dat := RandomVector[row] (size, generator = RandomInteger)
elif type = "IntegerMatrix8" then dat := RandomMatrix (size, size, generator = RandomInteger, datatype = integer[8]) elif type = "IntegerVector8" then dat := RandomVector[row] (size, generator = RandomInteger, 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)
elif type = "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) end if;
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 + AbsoluteTiming (' expr (data)') end do;
totaltime/repeats end proc;

timedReport := proc (filename, expr, hi, type) ExportMatrix (cat ("Maple", filename, ".dat"), 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, time;
data := makeData (type, n);
AbsoluteTiming (' map (fn, data)') end proc;

numerictest := proc (file, fns, n, type) ExportMatrix (cat ("Maple", file, ".dat"), Matrix ([seq (numericfntest (i, n, type), `in` (i,
```

```
fns))])) end proc;

numerictestinplace := proc (file, fns, n, type) ExportMatrix (cat ("Maple", file, ".dat"), Matrix ([seq (numericfntestinplace (i, n, type), `in` (i, fns))])) end proc;

numerictestelementwise := proc (file, fns, n, type) ExportMatrix (cat ("Maple", file, ".dat"), Matrix ([seq (elementwisefntest (i, n, type), `in` (i, fns))])) end proc;

numericfntestinplace := proc (fn, n, type) local data, time;
data := makeData (type, n);
AbsoluteTiming (' Map (fn, data)') end proc;

elementwisefntest := proc (fn, n, type) local data, time;
data := makeData (type, n);
AbsoluteTiming (' fn (data)') end proc;

highPrecisionTest := proc (file, fns, n) ExportMatrix (cat ("Maple", file, ".dat"), Matrix ([seq (AbsoluteTiming (' evalf (i, n)'), `in` (i, fns))])) end proc;

RandomInteger := proc (x, y) RandomTools[Generate] (integer (range = 1 .. 100)) end proc;



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

timedReport ("FourierReal", ' FourierTransform', maxVector, "Vector");
timedReport ("SortReal", ' sort', maxVector, "Vector");
timedReport ("MeanReal", ' Mean', 2*maxVector, "Vector");
timedReport ("DotReal", ' proc (x) x.x end 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^5 end 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 ("ElementPowerReal", ' proc (m) map (proc (x) x^5 end proc, m) end proc', (1/2)*maxMatrix, "Matrix");
timedReport ("FitReal", ' proc (m) Fit (a + b*x, m, m, x) end proc', maxVector, "Vector");
timedReport ("MatrixExpReal", ' MatrixExponential', (1/5)*maxMatrix, "Matrix");
timedReport ("CovarianceReal", ' CovarianceMatrix', maxMatrix, "Matrix");
timedReport (“SortCustomReal”, ‘ proc (data) sort (data, proc (x, y) abs (y) < abs (x) end proc) end proc’, (1/25)*maxVector, “Vector”);

timedReport ("FourierFloat8", ' FourierTransform', maxVector, "VectorFloat8");
timedReport ("SortFloat8", ' sort', maxVector, "VectorFloat8");
timedReport ("MeanFloat8", ' Mean', 10*maxVector, "VectorFloat8");
timedReport ("DotFloat8", ' proc (x) x.x end 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^5 end 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 ("ElementPowerFloat8", ' proc (m) map (proc (x) x^5 end proc, m) end proc', (1/2)*maxMatrix, "MatrixFloat8");
timedReport ("FitFloat8", ' proc (m) Fit (a + b*x, m, m, x) end proc', maxVector, "VectorFloat8");
timedReport ("MatrixExpFloat8", ' MatrixExponential', (1/5)*maxMatrix, "MatrixFloat8");
timedReport ("CovarianceFloat8", ' CovarianceMatrix', maxMatrix, "MatrixFloat8");
timedReport ("SortCustomFloat8", ' proc (data) sort (data, proc (x, y) abs (y) < abs (x) end proc) end proc', (1/25)*maxVector,
"VectorFloat8");

timedReport ("FourierComplex", ' FourierTransform', maxVector, "ComplexVector");
timedReport ("MeanComplex", ' proc (x) add (i, i = x)/Dimension (x) end proc', (1/3)*maxVector, "ComplexVector");
timedReport ("DotComplex", ' proc (x) x.x end 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^5 end 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', 3*maxMatrix*(1/4), "ComplexMatrix");
timedReport ("ElementPowerComplex", ' proc (m) map (proc (x) x^5 end proc, m) end proc', (1/4)*maxMatrix,
"ComplexMatrix");
timedReport ("MatrixExpComplex", ' MatrixExponential', (1/5)*maxMatrix, "ComplexMatrix");
timedReport ("SortCustomComplex", ' proc (data) sort (data, proc (x, y) abs (y) < abs (x) end proc) end proc',
(1/25)*maxVector, "ComplexVector");

timedReport ("FourierComplex8", ' FourierTransform', maxVector, "ComplexVector8");
timedReport ("MeanComplex8", ' proc (x) add (i, i = x)/Dimension (x) end proc', (1/3)*maxVector, "ComplexVector8");
timedReport ("DotComplex8", ' proc (x) x.x end 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^5 end 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^5 end proc, m) end proc', (1/4)*maxMatrix,
"ComplexMatrix8");
timedReport ("MatrixExpComplex8", ' MatrixExponential', (1/5)*maxMatrix, "ComplexMatrix8");
timedReport ("SortCustomComplex8", ' proc (data) sort (data, proc (x, y) abs (y) < abs (x) end proc) end proc',
(1/25)*maxVector, "ComplexVector8");

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', (1/5)*maxMatrix, "IntegerMatrix");
timedReport ("MatrixPowerInteger", ' proc (x) x^5 end proc', (1/5)*maxMatrix, "IntegerMatrix");
timedReport ("DetInteger", ' Determinant', (1/5)*maxMatrix, "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 ("SortCustomInteger", ' proc (data) sort (data, proc (x, y) abs (y) < abs (x) end proc) end proc', (1/25)*maxVector,
"IntegerVector");

timedReport ("DotInteger8", ' proc (x) x.x end proc', maxMatrix, "IntegerMatrix8");
timedReport ("LinearSolveInteger8", ' proc (x) LinearSolve (x, Column (x, 1)) end proc', (1/5)*maxMatrix, "IntegerMatrix8");
timedReport ("MatrixPowerInteger8", ' proc (x) x^5 end proc', (1/5)*maxMatrix, "IntegerMatrix8");
```

```
timedReport ("DetInteger8", ' Determinant', (1/5)*maxMatrix, "IntegerMatrix8");
tiedReport ("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^5 end proc, m) end proc', maxMatrix, "IntegerMatrix8");
steps := 3;
timedReport ("MatrixExpInteger8", ' MatrixExponential', 3, "IntegerMatrix8");
timedReport ("MatrixExpInteger", ' MatrixExponential', 3, "IntegerMatrix");
steps := 10;
timedReport ("SortCustomInteger8", ' proc (data) sort (data, proc (x, y) abs (y) < abs (x) end proc) end proc', 100000,
"IntegerVector8");

ExportMatrix ("MapleExactFunctions.dat", Matrix ([[
AbsoluteTiming (' seq (bernoulli (i), i = 1 .. 6000)'),
AbsoluteTiming (' seq (fibonacci (i), i = 1 .. 500000, 1000)'),
AbsoluteTiming (' seq (binomial (i^2, i), i = 1 .. 10000)')]]));

ExportMatrix ("MapleRandomNumbers.dat", Matrix ([[
AbsoluteTiming (' makeData ("Matrix", 2000)'),
AbsoluteTiming (' makeData ("MatrixFloat8", 2000)'),
AbsoluteTiming (' makeData ("IntegerMatrix", 2000)'),
AbsoluteTiming (' makeData ("IntegerMatrix8", 2000)'),
AbsoluteTiming (' makeData ("BigNumberMatrix", 2000)'),
AbsoluteTiming (' makeData ("BigNumberMatrix1000", 2000)'),
 AbsoluteTiming (' Sample (RandomVariable (Normal (0, 1)), 10^7)'),
 AbsoluteTiming (' Sample (RandomVariable (Poisson (4)), 10^7)'),
 AbsoluteTiming (' Sample (RandomVariable (BinomialDistribution (10, .2)), 10^7)')]]));

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];

numerictest ("ElementaryFunctionsComplexSmall", fnList, (1/10)*maxVector, "ComplexVector");
numerictest ("ElementaryFunctionsSmall", fnList, maxVector, "Vector");
numerictest ("ElementaryFunctionsFloat8Small", fnList, maxVector, "VectorFloat8");
numerictest ("ElementaryFunctionsComplex8Small", fnList, (1/10)*maxVector, "ComplexVector8");
numerictest ("ElementaryFunctionsComplex8Medium", fnList, (1/2)*maxVector, "ComplexVector8");
numerictest ("ElementaryFunctionsFloat8Medium", fnList, 5*maxVector, "VectorFloat8");
numerictest ("ElementaryFunctionsMedium", fnList, 10*maxVector*(1/2), "Vector");

fnListSafe := [sin, cos, tan, arctan, sec, csc, cot, exp, sinh, cosh, tanh, erf, proc (x) BesselJ (0, x) end proc, arcsinh, Zeta];
numerictestinplace ("ElementaryFunctionsInPlaceFloat8", fnListSafe, 10*maxVector, "VectorFloat8");
numerictestinplace ("ElementaryFunctionsInPlaceComplex", fnList, maxVector, "ComplexVector");
numerictestinplace ("ElementaryFunctionsInPlaceComplex8", fnList, maxVector, "ComplexVector8");
numerictestinplace ("ElementaryFunctionsInPlaceSmall", fnList, maxVector, "Vector");
numerictestinplace ("ElementaryFunctionsInPlaceMedium", fnList, 5*maxVector, "Vector");

elementwisefnListSafe := [`~`[sin], `~`[cos], `~`[tan], `~`[arctan], `~`[sec], `~`[csc], `~`[cot], `~`[exp], `~`[sinh], `~`[cosh], `~`[tanh],
`~`[erf], `~`[proc (x) BesselJ (0, x) end proc], `~`[arcsinh], `~`[Zeta]];

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]];

numerictestelementwise ("ElementaryFunctionsElementwiseFloat8", elementwisefnListSafe, 10*maxVector, "VectorFloat8");
numerictestelementwise ("ElementaryFunctionsElementwiseComplexSmall", elementwisefnList, (1/10)*maxVector,
"ComplexVector");
numerictestelementwise ("ElementaryFunctionsElementwiseComplex8Small", elementwisefnList, (1/10)*maxVector,
"ComplexVector8");
numerictest ("ElementaryFunctionsComplex", fnList, maxVector, "ComplexVector");

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);
```

```
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',
(1/25)*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");
numerictest ("ElementaryFunctionsExtended", fnList, (1/10)*maxVector, "BigNumberVector");
numerictestelementwise ("ElementaryFunctionsElementwiseExtended", elementwisefnList, (1/10)*maxVector,
"BigNumberVector");
numerictestinplace ("ElementaryFunctionsInPlaceExtended", fnList, (1/10)*maxVector, "BigNumberVector");

Digits := 1000;
timedReport ("DotExtended1000", ' proc (x) x.x end proc', (1/25)*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',
(1/25)*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");
```