Comparison of Numerical Performance of *Mathematica* 8 and Maple 15

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 145 tasks, performed for 20 different problem sizes from small to medium, *Mathematica* was faster in almost every case. The median difference measured found *Mathematica* to be 63 times faster than Maple.

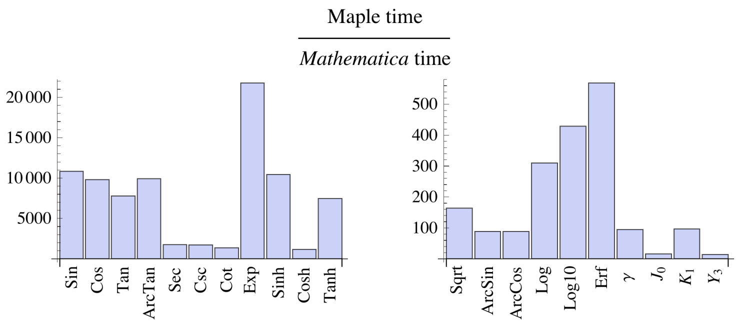
The Maple 15 press release states “Maple 15 is second to none in terms of scalability and performance.” This statement is demonstrably false.

Category	Tests	Median *Mathematica* performance
Elementary & special functions	21	1182 times faster
Complex elementary & special functions	21	2225 times faster
Extended−precision elementary & special functions	21	19 times faster
High−precision function evaluation	8	58 times faster
Real data operations (Default data type)	14	8 times faster
Real data operations (Manual type override)	13	3 times faster
Complex number data operations	12	7 times faster
Complex number data operations (Manual type override)	12	6 times faster
Integer data operations	10	14 times faster
Integer data operations (Manual type override)	9	10 times faster
Sparse real data operations	8	19,899 times faster
Extended−precision data operations	12	10 times faster
Programming	2	65 times faster
GPU use	1	3 times faster

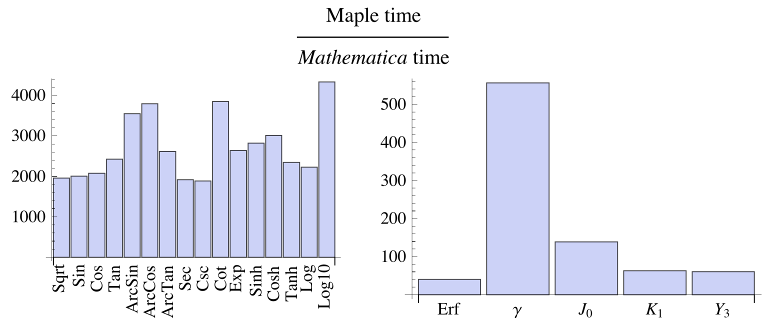
Tests were performed using Windows 7 64-bit with 3.33 GHz quad-core Intel Xeon processors (W3580) with 12 GB of RAM. CUDA tests used a Tesla C2050/C2070 GPU with 448 cores and 2.5 GB of RAM.

Function Evaluation

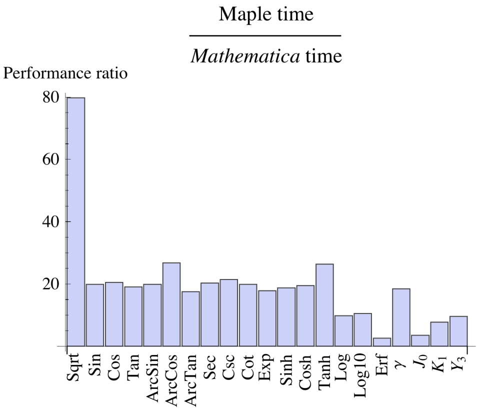
The time taken to evaluate elementary or special functions over a vector of real numbers is massively higher in Maple. In this test, each function was applied to a vector of 10^7 machine-size real numbers. The median relative performance measured *Mathematica* to be over 1000 times faster than Maple. Huge variance is shown, with *Mathematica* only 15 times faster for BesselY but over 20,000 times faster for Exp.



Applying the same functions to a vector of 10^6 machine precision complex numbers measured *Mathematica* to have a median performance advantage of over 2000 times faster.



When evaluated over a vector of 10^5 numbers using 50 digits of extended-precision, *Mathematica*'s median performance was 17 times faster.

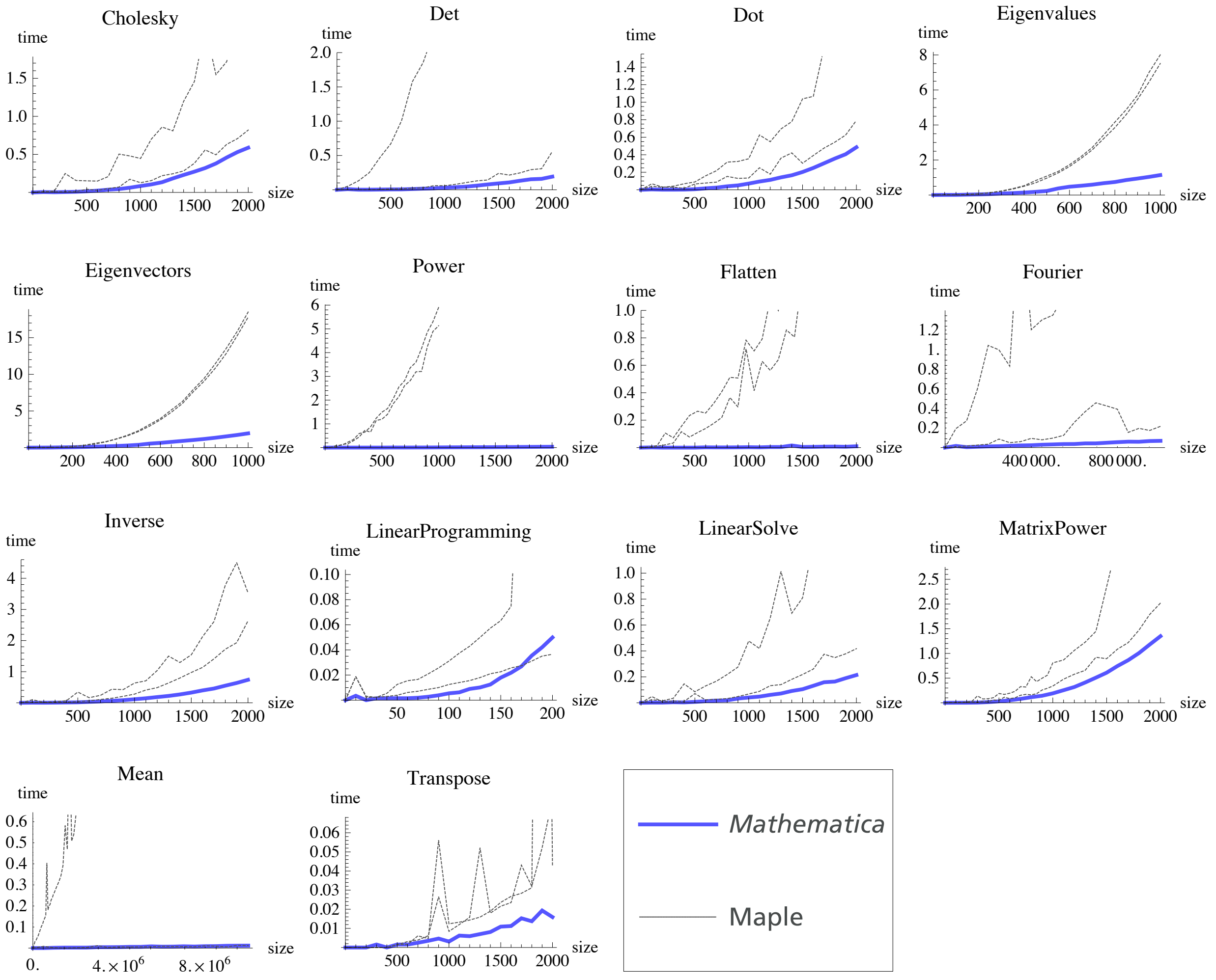


Evaluating all these tests together takes approximately 20 minutes in *Mathematica* and over 13 hours in Maple.

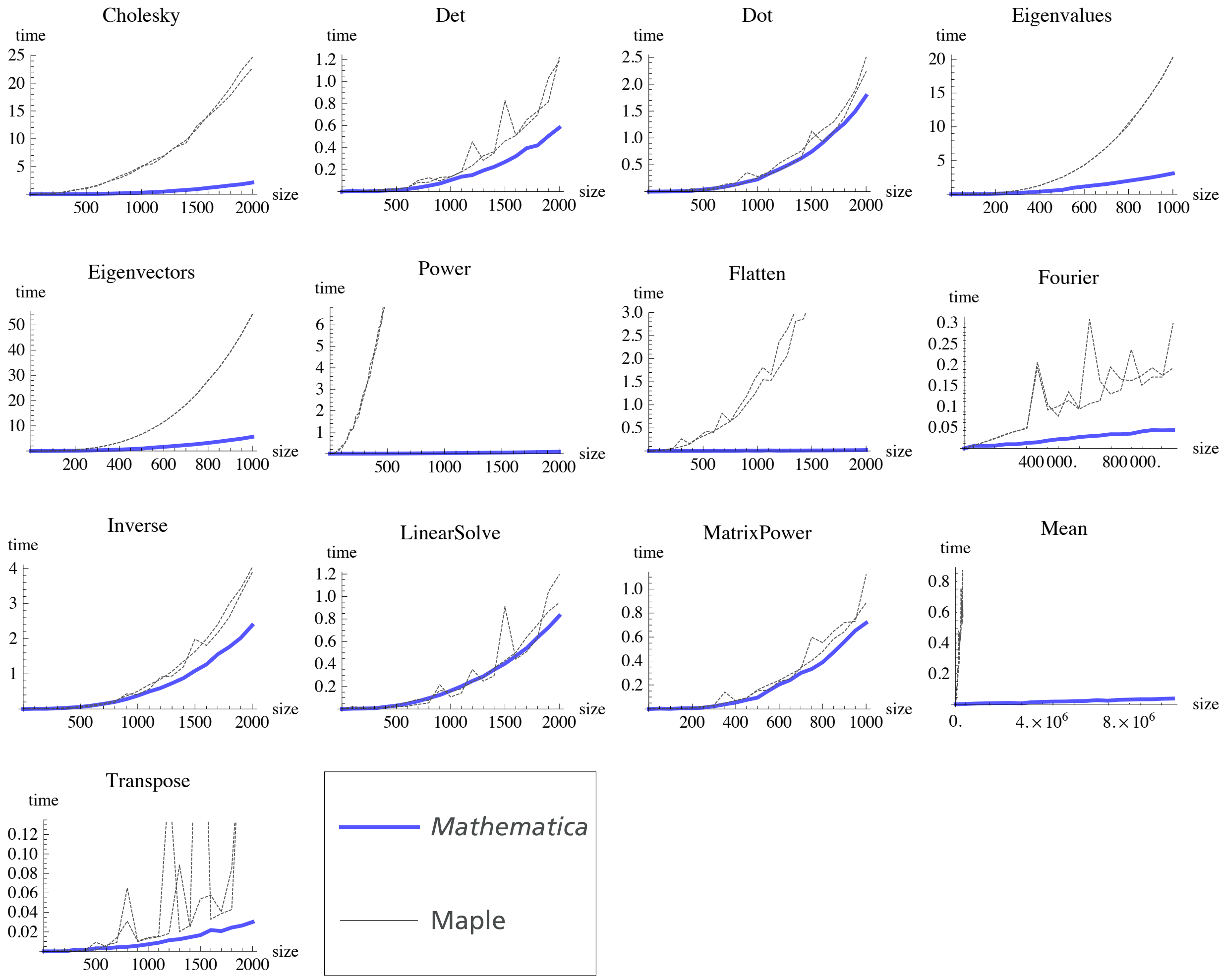
Machine-Precision Linear Algebra

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 3 times faster than Maple's inflexible (float[8]) data type and 13 times faster than its default type.

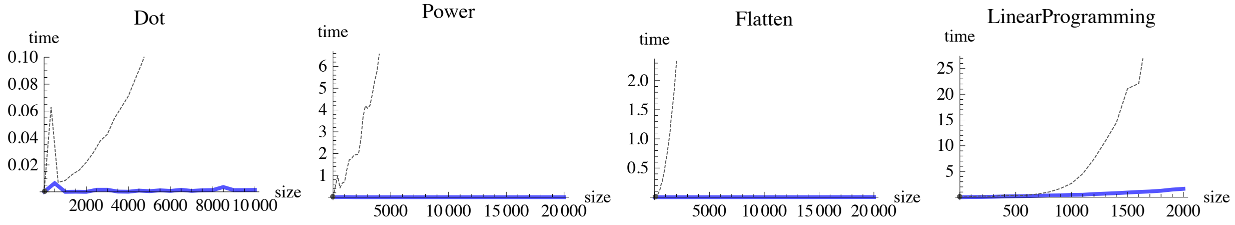


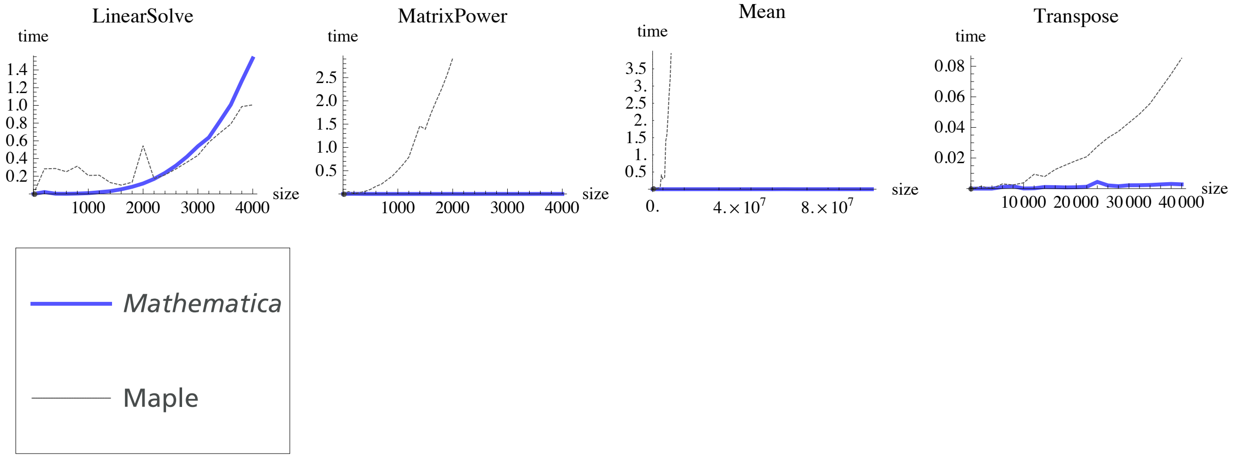
For complex numbers, there is little advantage to manually setting the data type in Maple. *Mathematica* is more than 6 times faster.



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.

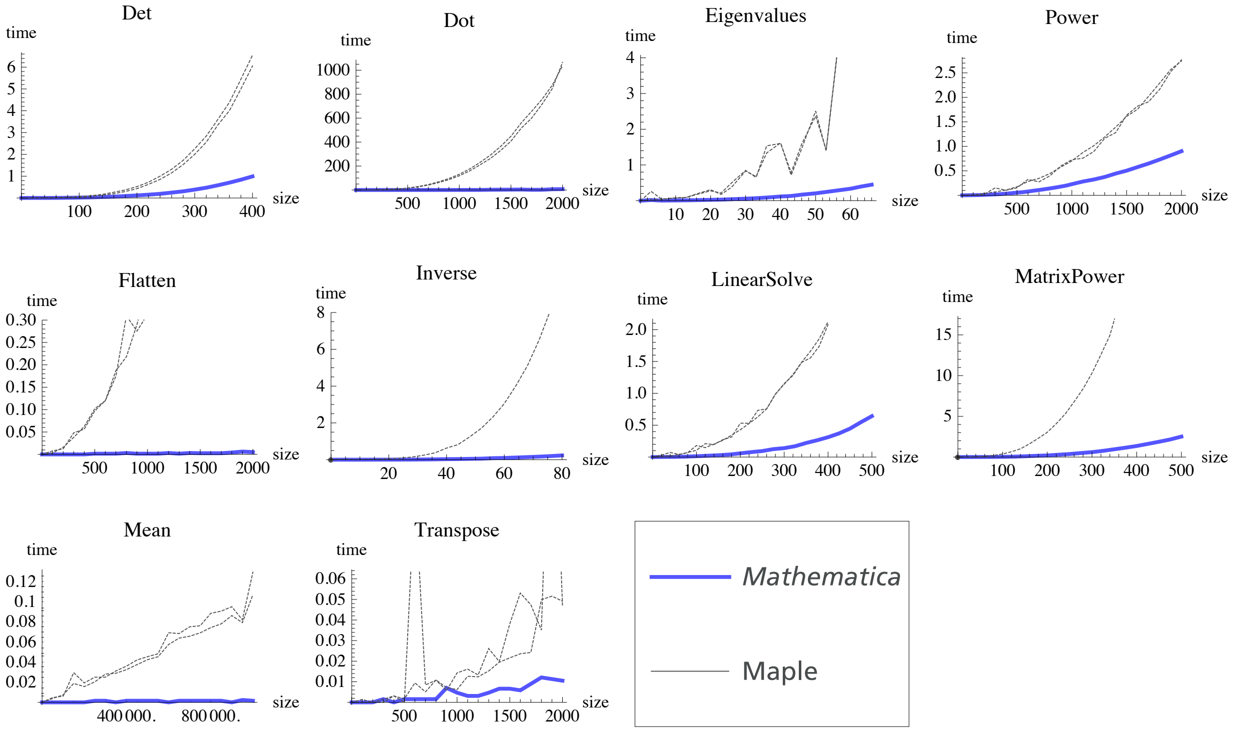




Integer Linear Algebra

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 145 times faster.

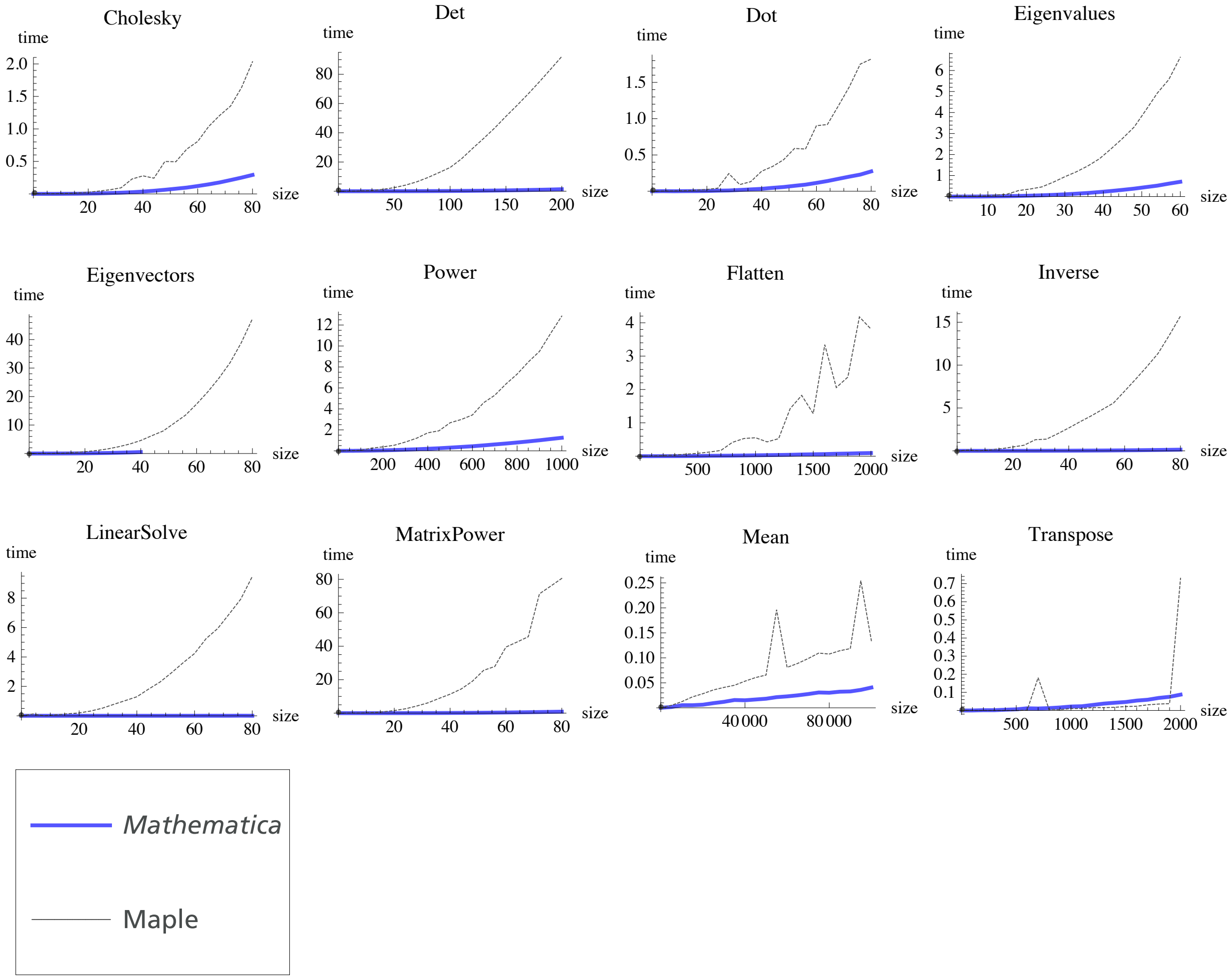
The median difference put *Mathematica* at 15 times faster.



Note: Maple cannot evaluate the Inverse and MatrixPower tasks using the integer[8] type, so only default data type results are shown.

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.

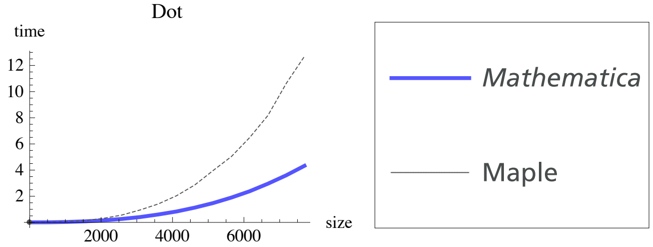


Fourier and Linear programming tests are excluded because Maple returns only machine-precision results.

GPU Performance

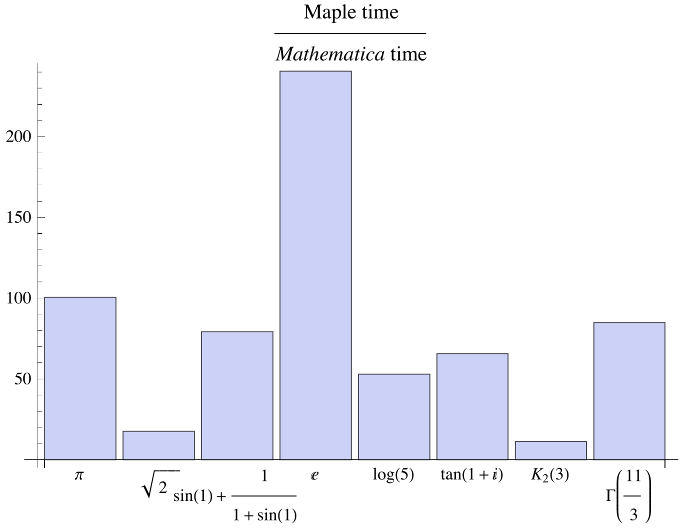
Maple's CUDA support is extremely limited, with only a single function, Dot, implemented, and only for double-precision CUDA hardware. It has no OpenCL or single precision support. *Mathematica* can run arbitrary CUDA or OpenCL code, and has many built-in CUDA accelerated functions.

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



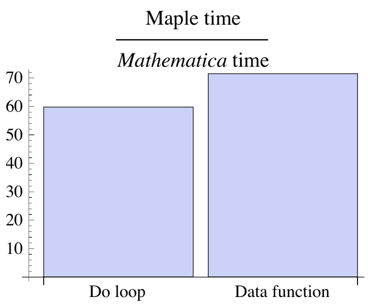
High-Precision Evaluation

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. The examples in this test were too simple for this to matter, but despite this extra verification work, *Mathematica* was between 10 and 240 times faster than Maple at evaluating expressions to 1,000,000 decimal places. (BesselK and Gamma were evaluated to only 5000 digits.)



Programming Performance

Mathematica supports internal compilation of its programs for fast execution. Maple does not have this capability. The result is that two low-level loop- and test-based programs run 60 and 71 times faster in *Mathematica*.



Implementation Notes

Tests were performed using Windows 7 64-bit with 3.33 GHz quad-core Intel Xeon processors (W3580) and 12 GB of RAM. CUDA tests used a Tesla C2050/C2070 GPU with 448 cores and 2.5 GB of RAM.

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 alternative Maple implementations created. See source code for details.

For most functions, the test suite finds the average time for 10 evaluations for each size of problem, but high-precision evaluations were performed only once, since both systems cache the results for later reuse.

Because Maple results vary dramatically between different sessions, the entire test suite was run more than once and the times averaged between different sessions.

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*.

The entire test, including random data generated, takes approximately 90 minutes in *Mathematica* and several days in Maple.

Mathematica **program**

Test program

```
$HistoryLength = 0;

SetDirectory["C:\\Users\\jonm\\Desktop\\MapleBenchmarks"];

steps = 20;
repeats = 10;
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],
  "BigVector", RandomReal[1, {i}, WorkingPrecision → 50],
  "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" <> 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" <> path <> ".dat", Map[AbsoluteTiming[N[#, n]][[1]] &, fns]]
```

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

Real Matrix Operations

```
timedReport["FourierReal", Fourier, maxVector, "RealVector"];
```

```
timedReport["MeanReal", Mean, 10 maxVector, "RealVector"];
```

```
timedReport["DotReal", #.# &, maxMatrix , "RealMatrix"];
```

```
timedReport["DetReal", Det, maxMatrix, "RealMatrix"];
```

```
timedReport["InverseReal", Inverse, maxMatrix, "RealMatrix"];
```

```
timedReport["CholeskyReal",
  CholeskyDecomposition[Transpose[#].#] &, maxMatrix, "RealMatrix"];
```

```
timedReport["EigenvaluesReal", Eigenvalues, maxMatrix / 2, "RealMatrix"];
```

```
timedReport["EigenvectorsReal", Eigenvectors, maxMatrix / 2, "RealMatrix"];
```

```
timedReport["ElementPowerReal", #^5 &, maxMatrix, "RealMatrix"];
```

```
timedReport["LinearSolveReal", LinearSolve[#, #[[1]]] &, maxMatrix, "RealMatrix"];
```

```
timedReport["LinearProgrammingReal",
  LinearProgramming[#[[1]], #, #[[2]], Method → "Simplex"] &, maxMatrix / 10, "RealMatrix"];
```

```
timedReport["TransposeReal", Transpose, maxMatrix, "RealMatrix"];
```

```
timedReport["FlattenReal", Flatten, maxMatrix, "RealMatrix"];
```

```
timedReport["MatrixPowerReal", MatrixPower[#, 5] &, maxMatrix, "RealMatrix"];
```

Complex Matrices Operations

```
timedReport["FourierComplex", Fourier, maxVector, "ComplexVector"];

timedReport["MeanComplex", Mean, 10 maxVector, "ComplexVector"];

timedReport["DetComplex", Det, maxMatrix, "ComplexMatrix"];

timedReport["DotComplex", #.# &, maxMatrix , "ComplexMatrix"];

timedReport["InverseComplex", Inverse, maxMatrix, "ComplexMatrix"];

timedReport["EigenvaluesComplex", Eigenvalues, maxMatrix / 2, "ComplexMatrix"];

timedReport["EigenvectorsComplex", Eigenvectors, maxMatrix / 2, "ComplexMatrix"];

timedReport["ElementPowerComplex", #^5 &, maxMatrix, "ComplexMatrix"];

timedReport["LinearSolveComplex", LinearSolve[#, #[[1]]] &, maxMatrix, "ComplexMatrix"];

timedReport["TransposeComplex", Transpose, maxMatrix, "ComplexMatrix"];

timedReport["FlattenComplex", Flatten, maxMatrix, "ComplexMatrix"];

timedReport["CholeskyComplex",
  CholeskyDecomposition[ConjugateTranspose[#].#] &, maxMatrix, "ComplexMatrix"];

timedReport["MatrixPowerComplex", MatrixPower[#, 5] &, maxMatrix / 2, "ComplexMatrix"];
```

Sparse Operations

```
timedReport["DotSparse", #.# &, maxSparseMatrix / 2, "SparseMatrix"];

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"];

Quiet@timedReport["LinearSolveSparse",
   LinearSolve[#, #[[1]]] &, maxSparseMatrix / 5, "MediumSparseMatrix"];

Quiet@timedReport["LinearProgrammingSparse",
   LinearProgramming[#[[1]], #, #[[2]], ConstantArray[0, Length[#[[1]]]],
     Method → "RevisedSimplex"] &, maxSparseMatrix / 10 , "MediumSparseMatrix"];
```

Integer Operations

```
timedReport["DetInteger", Det, maxMatrix  / 5 , "IntegerMatrix"];

timedReport["DotInteger", #.# &, maxMatrix , "IntegerMatrix"];

timedReport["InverseInteger", Inverse, maxMatrix / 25, "IntegerMatrix"];

timedReport["LinearSolveInteger", LinearSolve[#, #[[1]]] &, maxMatrix / 4, "IntegerMatrix"];

timedReport["ElementPowerInteger", #^5 &, maxMatrix, "IntegerMatrix"];

timedReport["TransposeInteger", Transpose, maxMatrix, "IntegerMatrix"];

timedReport["FlattenInteger", Flatten, maxMatrix, "IntegerMatrix"];

timedReport["MeanInteger", Mean, maxVector, "IntegerVector"];

timedReport["SortInteger", Sort, maxVector, "IntegerVector"];

timedReport["EigenvaluesInteger", Eigenvalues, maxMatrix / 30, "IntegerMatrix"];
```

```
timedReport["MatrixPowerInteger", MatrixPower[#, 5] &, maxMatrix / 4, "IntegerMatrix"];

(*timedReport["LinearProgrammingInteger",
  LinearProgramming[#[[1]],#,#[[2]],Method→"Simplex"]&,maxMatrix/10,"IntegerMatrix"];*)
```

Maple can’t do this in its matrix notation. It converts to real.

Extended-Precision Matrix Ops

Same as matrix ops except done at precision 50

```
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, "BigVector"];

timedReport["MeanExtended", Mean, maxVector / 10, "BigVector"];

timedReport["SortExtended", Sort, maxVector / 10, "BigVector"];

timedReport["EigenvectorsExtended", Eigenvectors, maxMatrix / 50, "ExtendedMatrix"];

timedReport["MatrixPowerExtended", MatrixPower[#, 5] &, maxMatrix / 25, "ExtendedMatrix"];
```

High Precision

```
highPrecisionReport["ManyDigits",
                                  1
  {Pi, Sqrt[2], Sin[1] + ──────────, Exp[1], Log[5], Tan[1 + I]}, 1 000 000];
                         1 + Sin[1]
highPrecisionReport["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, BesselJ[0, #] &, BesselK[1, #] &, BesselY[3, #] &};
specialfnlist = {Erf, Gamma, BesselJ[0, #] &, BesselK[1, #] &, BesselY[3, #] &};
NumericTest["ElementaryFunctions", fnList, 10 maxVector , "RealVector"];

NumericTest["ElementaryFunctionsComplex", fnList, maxVector , "ComplexVector"];

NumericTest["ElementaryFunctionsExtended", fnList, maxVector / 10, "BigVector"];
```

Programming

```
dotest = Compile[{{n, _Integer}},
   Block[{a = 1.}, Do[a = a + If[x > y, x, y], {x, 1, n}, {y, 1, n}]; a]];

datafntest = Compile[{{x, _Real, 1}}, If[# > 0, #^2, #^4] & /@ x];

Export["MathematicaProgramming.dat", {
   AbsoluteTiming[dotest[5. maxMatrix]][[1]],
   AbsoluteTiming[datafntest[makeData["RealVector", maxVector * 10.]]][[1]]}];
```

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”):

#
```

For more comparisons with Maple, visit
http://www.wolfram.com/mathematica/compare-mathematica/compare-mathematica-and-maple.html