Chapter 1, exercise 3

Problem

Compute the relative and absolute speeds (flops) of

* addition, multiplication, division, and vector inner product
* exponentiation (e^x), power (y^x) and logarithm base 10,
* the trigonometric functions: sine, cosine, secant, cosecant, tangent, cotangent

Solution for double precision arithmetic

The following results were computed on an old MacBook Pro. (A little story. I spilled some juice on this machine at ZICE2016. This messed up the keyboard. However, it is still works fine by using an external keyboard and mouse. It's so hard to kill these machines that one feels guilty buying new ones when they come out. Fortunately, the guilty feeling passes quickly and I get new ones.)

NumOps is the number of operations we execute.

```
NumOps = 10^8;
```

We create two lists of NumOps random numbers between 0.1 and 10.0 (we want to avoid bad results involving 0).

```
NumList1 = RandomReal[{0.1, 10.0}, NumOps];
NumList2 = RandomReal[{0.1, 10.0}, NumOps];
```

Some languages can exploit the vector structure of input. For example, adding two lists (a.k.a. vectors) can be done by

```
{1., 2., 3., 4.} + {2., 3., 4., 5.}
```

```
{3., 5., 7., 9.}
```

Similarly for other operations

```
{1., 2., 3., 4.} * {2., 3., 4., 5.}
```

```
{2., 6., 12., 20.}
```

```
{1., 2., 3., 4.} / {2., 3., 4., 5.}
```

```
{0.5, 0.666667, 0.75, 0.8}
```

```
Exp[{1., 2., 3., 4.}]
```

```
{2.71828, 7.38906, 20.0855, 54.5982}
```

$\{1., 2., 3., 4.\}^{\{2., 3., 4., 5.\}}$

```
{1., 8., 81., 1024.}
```

This will generally be faster than a loop. We use this approach for our timing tests.

Timings

Addition

First is addition.

```
time = (NumList1 + NumList2; // AbsoluteTiming) // First
```

```
0.287222
```

```
opsadd = NumOps / time
```

3.48163×10^8

Multiplication.

```
time = (NumList1 * NumList2; // AbsoluteTiming) // First
opsmult = NumOps / time
```

```
0.34178
```

2.92586×10^8

Division:

```
time = (NumList1 / NumList2; // AbsoluteTiming) // First
opsdiv = NumOps / time
```

0.705748

1.41694×10^{8}

The exponential function

In Mathematica, the defined functions are listable. That is, f[list] will produce a list equal to f evaluated at each element of the list. For example,

```
Exp[{1., 2., 3.}]
```

{2.71828, 7.38906, 20.0855}

This is the fastest way to evaluate a function over a list. We shall use this feature in our timings.

```
time = (Exp[NumList1]; // AbsoluteTiming) // First;
opsexp = NumOps / time
```

2.80497×10^{8}

The power function

```
time = (Power[NumList1, NumList2]; // AbsoluteTiming) // First;
opspow = NumOps / time
```

7.06255×10^{7}

The logarithm function

```
time = (Log[NumList1]; // AbsoluteTiming) // First;
opslog = NumOps / time
```

2.33119×10^{8}

The trig functions

```
time = (Sin[NumList1]; // AbsoluteTiming) // First
opssin = NumOps / time
```

0.448358

2.23036×10^{8}

```
time = (Cos[NumList1]; // AbsoluteTiming) // First
opscos = NumOps / time
```

0.449504

2.22467×10^{8}

```
time = (Tan[NumList1]; // AbsoluteTiming) // First
opstan = NumOps / time
```

0.528369

1.89262×10^{8}

```
time = (Cot[NumList1]; // AbsoluteTiming) // First
opscot = NumOps / time
```

0.960282

1.04136×10^{8}

```
time = (Sec[NumList1]; // AbsoluteTiming) // First
opssec = NumOps / time
```

0.978327

1.02215×10^{8}

```
time = (Csc[NumList1]; // AbsoluteTiming) // First
opscsc = NumOps / time
```

0.907669

1.10172×10^{8}

Summary tables

```
labels = {"Addition", "Multiplication", "Division", "Exponentiation",
   "Log10", "Power", "Sine", "Cosine", "Tangent", "Secant", "Cosecant", "Cotangent"};
```

```
speeds = {opsadd, opsmult, opsdiv, opsexp, opslog, opspow, opscos, opssin, opstan, opssec, opscsc, opscot};
```

The speeds are

```
{labels, speeds} // Transpose // TableForm
```

Addition	3.48163×10^{8}
Multiplication	2.92586×10^{8}
Inner Product	1.10172×10^{8}
Division	1.41694×10^{8}
Exponentiation	2.80497×10^{8}
Log10	2.33119×10^{8}
Power	7.06255×10^{7}
Sine	2.22467×10^{8}
Cosine	2.23036×10^{8}
Tangent	1.89262×10^{8}
Secant	1.02215×10^{8}
Cosecant	1.10172×10^{8}
Cotangent	1.04136×10^{8}

If we use addition as the norm, the relative speeds are

```
{labels, speeds / opsadd} // Transpose // TableForm
```

Addition	1.
Multiplication	0.840371
Inner Product	0.316439
Division	0.406975
Exponentiation	0.805649
Log10	0.669568
Power	0.202852
Sine	0.638975
Cosine	0.640609
Tangent	0.543601
Secant	0.293585
Cosecant	0.316439
Cotangent	0.299102

and the relative time per operation is the inverse

```
{labels, opsadd / speeds} // Transpose // TableForm
```

Addition	1.
Multiplication	1.18995
Inner Product	3.16017
Division	2.45715
Exponentiation	1.24124
Log10	1.4935
Power	4.9297
Sine	1.56501
Cosine	1.56102
Tangent	1.83958
Secant	3.40617
Cosecant	3.16017
Cotangent	3.34334

Define speeds16 to be speeds because we used double precision arithmetic.

```
speeds16 = speeds;
```

Comments

The relative speeds are consistent with common sense.

Addition is the easiest and therefore the fastest.
Multiplication is a bit slower but not as slow as it would be if the computer did the same steps we do when multiplying numbers. There are special algorithms for doing multiplication.
Division is slower, which is not a suprise.

Exponentiation looks faster than you may expect but recall that Exp[x] is a unary function whereas addition, multiplication and division have two inputs. The more relevant comparison is the Power operation which takes two inputs. That is much slower than the basic arithmetic operations. Exp[x] and Log10[x] use highly developed hard-wired methods. I suspect y^x is a substantially more difficult operation to compute.

The trig functions are all between 1/3 and 2/3 as fast as addition.

In the later lectures on approximation, we will see how complex functions such as e^x and sin[x] can be computed so rapidly.

These results will differ substantially from those you will obtain when you use Matlab, C, or Fortran instead of *Mathematica*. In fact, the Mathematica speed may be different if you use some of the other commands. I have just used the simplest one.

Mathematica contains substantial overhead for any operation. That fixed cost is present in each of the operations above but could be less with Matlab, and much less with C and Fortran.

32-digit precision

We repeat this but now for higher-precision arithmetic.

NumOps is the number of operations we execute.

```
NumOps = 10^6;
```

We create two lists of NumOps random numbers between 0.1 and 10.0 (we want to avoid bad results involving 0) of 32-digit precision.

```
NumList1 = SetPrecision[ RandomReal[{1 / 10, 10}, NumOps], 32];
NumList2 = SetPrecision[RandomReal[{1 / 10, 10}, NumOps], 32];
```

Let’s see what the machine form of the typical 32-digit precision number looks like

```
NumList1[[2]] // FullForm
```

```
9.26995634038053850645155762322247028350830078125`128.
```

Timings

Addition

First is addition.

```
time = (NumList1 + NumList2; // AbsoluteTiming) // First
opsadd = NumOps / time
```

1.7478×10^{6}

Multiplication.

```
time = (NumList1 * NumList2; // AbsoluteTiming) // First
opsmult = NumOps / time
```

```
0.429713
```

2.32713×10^{6}

Division:

```
time = (NumList1 / NumList2; // AbsoluteTiming) // First
opsdiv = NumOps / time
```

```
1.25179
```

```
798 857.
```

The exponential function

```
time = (Exp[NumList1]; // AbsoluteTiming) // First;
opsexp = NumOps / time
```

697 101.

The power function

```
time = (Power[NumList1, NumList2]; // AbsoluteTiming) // First;
opspow = NumOps / time
```

259 568.

The logarithm function

```
time = (Log[NumList1]; // AbsoluteTiming) // First;
opslog = NumOps / time
```

596 518.

The trig functions

```
time = (Sin[NumList1]; // AbsoluteTiming) // First
opssin = NumOps / time
```

4.75244

210 418.

```
time = (Cos[NumList1]; // AbsoluteTiming) // First
opscos = NumOps / time
```

4.72035

211 849.

```
time = (Tan[NumList1]; // AbsoluteTiming) // First
opstan = NumOps / time
```

5.52111

181 123.

```
time = (Cot[NumList1]; // AbsoluteTiming) // First
opscot = NumOps / time
```

6.21565

160 884.

```
time = (Sec[NumList1]; // AbsoluteTiming) // First
opssec = NumOps / time
```

5.42119

184 462.

```
time = (Csc[NumList1]; // AbsoluteTiming) // First
opscsc = NumOps / time
```

5.53123

180 792.

Summary tables

```
labels = {"Addition", "Multiplication", "Division", "Exponentiation",
   "Log10", "Power", "Sine", "Cosine", "Tangent", "Secant", "Cosecant", "Cotangent"};
```

```
speeds32 = {opsadd, opsmult, opsdiv, opsexp, opslog, opspow, opscos, opssin, opstan, opssec, opscsc, opscot};
```

The speeds are

```
{labels, speeds32, speeds16} // Transpose // TableForm
```

Addition	1.7478×10^{6}	3.48163×10^{8}
Multiplication	2.32713×10^{6}	2.92586×10^{8}
Inner Product	2.32713×10^{6}	1.10172×10^{8}
Division	798 857.	1.41694×10^{8}
Exponentiation	697 101.	2.80497×10^{8}
Log10	596 518.	2.33119×10^{8}
Power	259 568.	7.06255×10^{7}
Sine	211 849.	2.22467×10^{8}
Cosine	210 418.	2.23036×10^{8}
Tangent	181 123.	1.89262×10^{8}
Secant	184 462.	1.02215×10^{8}
Cosecant	180 792.	1.10172×10^{8}
Cotangent	160 884.	1.04136×10^{8}

If we use addition as the norm, the relative speeds are

```
speedadd = speeds16[[1]];
{labels, speeds32 / speedadd, speeds16 / speedadd} // Transpose // TableForm
```

Addition	0.00502006	1.
Multiplication	0.00668404	0.840371
Inner Product	0.00668404	0.316439
Division	0.00229449	0.406975
Exponentiation	0.00200223	0.805649
Log10	0.00171333	0.669568
Power	0.000745537	0.202852
Sine	0.000608476	0.638975
Cosine	0.000604368	0.640609
Tangent	0.000520225	0.543601
Secant	0.000529814	0.293585
Cosecant	0.000519274	0.316439
Cotangent	0.000462095	0.299102

Going to 32-digit precision results in operations that are 200 to 2000 times slower. There is always a fixed cost of going beyond machine precision because these operations now involve nontrivial software that must use combinations of 16-digit precision machine arithmetic.

128-digit precision

We repeat this but now for 128-digit precision arithmetic.

NumOps is the number of operations we execute.

```
NumOps = 10^6;
```

We create two lists of NumOps random numbers between 0.1 and 10.0 (we want to avoid bad results involving 0) of 32-digit precision.

```
NumList1 = SetPrecision[ RandomReal[{1 / 10, 10}, NumOps], 128];
NumList2 = SetPrecision[RandomReal[{1 / 10, 10}, NumOps], 128];
```

```
NumList1[[2]] // FullForm
```

```
0.35629716249945531103549001272767782211303710937 5`128.
```

Timings

Addition

First is addition.

```
time = (NumList1 + NumList2; // AbsoluteTiming) // First
opsadd = NumOps / time
```

```
1.62751 × 10^6
```

Multiplication.

```
time = (NumList1 * NumList2; // AbsoluteTiming) // First
opsmult = NumOps / time
```

0.465794

2.14687×10^6

Division:

```
time = (NumList1 / NumList2; // AbsoluteTiming) // First
opsdiv = NumOps / time
```

1.41598

706 223.

The exponential function

```
time = (Exp[NumList1]; // AbsoluteTiming) // First;
opsexp = NumOps / time
```

415 528.

The power function

```
time = (Power[NumList1, NumList2]; // AbsoluteTiming) // First;
opspow = NumOps / time
```

167 959.

The logarithm function

```
time = (Log[NumList1]; // AbsoluteTiming) // First;
opslog = NumOps / time
```

378 204.

The trig functions

```
time = (Sin[NumList1]; // AbsoluteTiming) // First
opssin = NumOps / time
```

6.25862

159 780.

```
time = (Cos[NumList1]; // AbsoluteTiming) // First
opscos = NumOps / time
```

6.39561

156 357.

```
time = (Tan[NumList1]; // AbsoluteTiming) // First
opstan = NumOps / time
```

7.19025

139 077.

```
time = (Cot[NumList1]; // AbsoluteTiming) // First
opscot = NumOps / time
```

7.90729

126 466.

```
time = (Sec[NumList1]; // AbsoluteTiming) // First
opssec = NumOps / time
```

```
7.00951
```

```
142 663.
```

```
time = (Csc[NumList1]; // AbsoluteTiming) // First
opscsc = NumOps / time
```

```
7.06288
```

```
141 585.
```

Summary tables

```
labels = {"Addition", "Multiplication", "Division", "Exponentiation",
   "Log10", "Power", "Sine", "Cosine", "Tangent", "Secant", "Cosecant", "Cotangent"};
```

```
speeds128 = {opsadd, opsmult, opsdiv, opsexp, opslog, opspow, opscos, opssin, opstan, opssec, opscsc, opscot};
```

The speeds are

```
{labels, speeds128, speeds32, speeds16} // Transpose // TableForm
```

Addition	1.62751×10^{6}	1.7478×10^{6}	3.48163×10^{8}
Multiplication	2.14687×10^{6}	2.32713×10^{6}	2.92586×10^{8}
Inner Product	2.14687×10^{6}	2.32713×10^{6}	1.10172×10^{8}
Division	706 223.	798 857.	1.41694×10^{8}
Exponentiation	415 528.	697 101.	2.80497×10^{8}
Log10	378 204.	596 518.	2.33119×10^{8}
Power	167 959.	259 568.	7.06255×10^{7}
Sine	156 357.	211 849.	2.22467×10^{8}
Cosine	159 780.	210 418.	2.23036×10^{8}
Tangent	139 077.	181 123.	1.89262×10^{8}
Secant	142 663.	184 462.	1.02215×10^{8}
Cosecant	141 585.	180 792.	1.10172×10^{8}
Cotangent	126 466.	160 884.	1.04136×10^{8}

If we use double precision addition as the norm, the relative speeds are

```
speedadd = speeds16[[1]];
{labels, speeds128 / speedadd, speeds32 / speedadd, speeds16 / speedadd} // Transpose // TableForm
```

Addition	0.00467457	0.00502006	1.
Multiplication	0.00616629	0.00668404	0.840371
Inner Product	0.00616629	0.00668404	0.316439
Division	0.00202843	0.00229449	0.406975
Exponentiation	0.00119349	0.00200223	0.805649
Log10	0.00108629	0.00171333	0.669568
Power	0.000482415	0.000745537	0.202852
Sine	0.000449093	0.000608476	0.638975
Cosine	0.000458922	0.000604368	0.640609
Tangent	0.00039946	0.000520225	0.543601
Secant	0.000409761	0.000529814	0.293585
Cosecant	0.000406664	0.000519274	0.316439
Cotangent	0.000363237	0.000462095	0.299102

Note, that going from 32-digit arithmetic to 128-digit precision arithmetic resulted in only a small loss of speed.

This is true in Mathematica, but may not be true for other high-precision software.