Issues with Floating Point

In [1]:
x = 1 - 10^(-17)
Out[1]:
1.0

In [2]:
x - 1
Out[2]:
0.0

In [3]:
x == 1
Out[3]:
true

In [4]:
y = 1.1 * 1.1
Out[4]:
1.2100000000000002

In [5]:
y == 1.21
Out[5]:
false
Timing elementary ops and memory references

You can generate the approx symbol by typing \approx followed by a tab.

The Version of Julia and Architecture of my Laptop

In [7]:
VERSION

Out[7]:
"v"1.3.1"

In [8]:
gethostname()

Out[8]:
"spielmans-MacBook-Pro.local"

In [9]:
Sys.cpu_summary()

Intel(R) Core(TM) i7-6567U CPU @ 3.30GHz:

<table>
<thead>
<tr>
<th>irq</th>
<th>speed</th>
<th>user</th>
<th>nice</th>
<th>sys</th>
<th>idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>3300 MHz</td>
<td>631285 s</td>
<td>0 s</td>
<td>231768 s</td>
<td>1416786 s</td>
</tr>
<tr>
<td>#2</td>
<td>3300 MHz</td>
<td>247144 s</td>
<td>0 s</td>
<td>90263 s</td>
<td>1941905 s</td>
</tr>
<tr>
<td>#3</td>
<td>3300 MHz</td>
<td>616553 s</td>
<td>0 s</td>
<td>202959 s</td>
<td>1459805 s</td>
</tr>
<tr>
<td>#4</td>
<td>3300 MHz</td>
<td>242469 s</td>
<td>0 s</td>
<td>83257 s</td>
<td>1953586 s</td>
</tr>
</tbody>
</table>

So, expect $3 \times 10^9$ ops per second, approximately.
That shows the various sizes of the Caches. Lower depth caches are faster. The linesize is in number of bytes. A typical Int or Float64 in Julia uses 8 bytes.

Timing code is a little delicate. We use a special package to do it.

```
In [12]:
using BenchmarkTools
```

```
In [13]:
n = 100_000_000
```

```
Out[13]:
1000000000
```

**Summation in a loop**
Here is a simple function that sums odd integers (as floats)

In [14]:

```python
function sum_to_n(n)
    s = 0.0
    for i in 1:n
        s += 2*i-1
    end
    return s
end
```

Out[14]:

sum_to_n (generic function with 1 method)

In [15]:

```python
sum_to_n(10)
```

Out[15]:

100.0

In [16]:

```python
@btime sum_to_n($n)
```

```
122.268 ms (0 allocations: 0 bytes)
```

Out[16]:

```
1.0e16
```

The reason we use @btime is that actual times are a little inconsistent.
In [17]:

t0 = time()
sum_to_n(n)
t1 = time()
println("Time was \$\(t1-t0\)")

In [17]:

t0 = time()
sum_to_n(n)
t1 = time()
println("Time was \$\(t1-t0\)")

In [17]:

t0 = time()
sum_to_n(n)
t1 = time()
println("Time was \$\(t1-t0\)")

Time was 0.1391279697418213
Time was 0.12775492668151855
Time was 0.1320970058441162

There are tools that give more information.

In [18]:

@benchmark sum_to_n($n)

Out[18]:

BenchmarkTools.Trial:
  memory estimate:  0 bytes
  allocs estimate:  0
  ---------------
  minimum time:    122.211 ms (0.00% GC)
  median time:     122.596 ms (0.00% GC)
  mean time:       124.331 ms (0.00% GC)
  maximum time:    147.393 ms (0.00% GC)
  ---------------
  samples:         41
  evals/sample:    1

A compiler gotcha

You might wonder why I computed that summation as floats, when all the terms were integers. It is because the sum over integers is much faster. In fact, it is too fast. Let's see.
In [49]:

```python
function sum_to_n_ints(n)
    s = 0
    for i in 1:n
        s += 2*i-1
    end
    return s
end
```

Out[49]:

```
sum_to_n_ints (generic function with 1 method)
```

In [50]:

```python
@btime sum_to_n_ints($n)
```

```
2.347 ns (0 allocations: 0 bytes)
```

Out[50]:

```
100000000
```

That was about 100,000 times faster! How can that be? Let's see what happens if we multiply n by 10.

In [51]:

```python
n2 = 10*n
@btime sum_to_n_ints($n2)
```

```
2.347 ns (0 allocations: 0 bytes)
```

Out[51]:

```
10000000000
```

Here's what's going on. Julia is a compiled language. It compiles each function for each type of inputs on which it is called. In this case, the compiler recognized that the solution has a closed form, and decided that the loop was unnecessary. (I also tried this in C, and got the same behavior with the -O1 flag).

You can see this by looking at the assembly code. It doesn't have a loop, and is very different from the assembly for the float case.
In [55]:

```c
@code_llvm sum_to_n_ints(n2)
```

define i64 @julia_sum_to_n_ints_18616(i64) {
  top:
    %1 = icmp sgt i64 %0, 0
    br i1 %1, label %L7.L12_crit_edge, label %L29

L7.L12_crit_edge: ; preds = %top
  %2 = mul i64 %0, 3
  %3 = add nsw i64 %0, -1
  %4 = add nsw i64 %0, -2
  %5 = mul i64 %3, %4
  %6 = and i64 %5, -2
  %7 = add i64 %2, %6
  %8 = add i64 %7, -2
    br label %L29

L29: ; preds = %L7.L12_crit_edge, %top
    %value_phi9 = phi i64 [ 0, %top ], [ %8, %L7.L12_crit_edge ]
    ret i64 %value_phi9
}

In [56]:

```c
@code_llvm sum_to_n(n2)
```

define double @julia_sum_to_n_18617(i64) {
  top:
    %1 = icmp sgt i64 %0, 0
    %2 = select i1 %1, i64 %0, i64 0
    %3 = add nsw i64 %0, -1
    %4 = add nsw i64 %0, -2
    %5 = mul i64 %3, %4
    %6 = and i64 %5, -2
    %7 = add i64 %2, %6
    %8 = add i64 %7, -2
    %9 = select i1 %1, double %8, double 0
    ret double %9
```
%vec.phi19 = phi <4 x double> [ zeroinitializer, %vector.ph], [
%21, %vector.body]
%vec.phi20 = phi <4 x double> [ zeroinitializer, %vector.ph], [
%22, %vector.body]
%vec.phi21 = phi <4 x double> [ zeroinitializer, %vector.ph], [
%23, %vector.body]

; | @ simdloop.jl:77 within `macro expansion' @ In[21]:4 |
; | @ int.jl:54 within `*'
%8 = shl nuw <4 x i64> %vec.ind, <i64 1, i64 1, i64 1, i64 1>
%step.add = shl <4 x i64> %vec.ind, <i64 1, i64 1, i64 1, i64 1>
%9 = add <4 x i64> %step.add, <i64 8, i64 8, i64 8, i64 8>
%step.add16 = shl <4 x i64> %vec.ind, <i64 1, i64 1, i64 1, i64 1>
%10 = add <4 x i64> %step.add16, <i64 16, i64 16, i64 16, i64 16>
%step.add17 = shl <4 x i64> %vec.ind, <i64 1, i64 1, i64 1, i64 1>
%11 = add <4 x i64> %step.add17, <i64 24, i64 24, i64 24, i64 24>

; | @ int.jl:52 within `-'
%12 = or <4 x i64> %8, <i64 1, i64 1, i64 1, i64 1>
%13 = or <4 x i64> %9, <i64 1, i64 1, i64 1, i64 1>
%14 = or <4 x i64> %10, <i64 1, i64 1, i64 1, i64 1>
%15 = or <4 x i64> %11, <i64 1, i64 1, i64 1, i64 1>

; | @ promotion.jl:311 within `+' @ int.jl:53 within `+' @ float.jl:60 within `Float64'
%16 = sitofp <4 x i64> %12 to <4 x double>
%17 = sitofp <4 x i64> %13 to <4 x double>
%18 = sitofp <4 x i64> %14 to <4 x double>
%19 = sitofp <4 x i64> %15 to <4 x double>

; | @ simdloop.jl:78 within `macro expansion' @ int.jl:53 within `+' @ float.jl:401
%20 = fadd fast <4 x double> %vec.phi, %16
%21 = fadd fast <4 x double> %vec.phi19, %17
%22 = fadd fast <4 x double> %vec.phi20, %18
%23 = fadd fast <4 x double> %vec.phi21, %19

middle.block: ; preds = %vector.body
body
@ simdloop.jl:77 within `macro expansion' @ In[21]:4
; | r @ promotion.jl:311 within `+' @ float.jl:401
  %bin.rdx = fadd fast <4 x double> %21, %20
  %bin.rdx22 = fadd fast <4 x double> %22, %bin.rdx
  %bin.rdx23 = fadd fast <4 x double> %23, %bin.rdx22
  %dx.shuf = shufflevector <4 x double> %bin.rdx23, <4 x double>
  undef, <4 x i32> <i32 2, i32 3, i32 undef, i32 undef>
  %bin.rdx24 = fadd fast <4 x double> %bin.rdx23, %dx.shuf
  %dx.shuf25 = shufflevector <4 x double> %bin.rdx24, <4 x double>
  undef, <4 x i32> <i32 1, i32 undef, i32 undef, i32 undef>
  %bin.rdx26 = fadd fast <4 x double> %bin.rdx24, %dx.shuf25
  %25 = extractelement <4 x double> %bin.rdx26, i32 0
  %cmp.n = icmp eq i64 %2, %n.vec
; | @ simdloop.jl:75 within `macro expansion'
  br i1 %cmp.n, label %L55, label %scalar.ph

scalar.ph:                                        ; preds = %middle.
  block, %L28.lr.ph
    %bc.resume.val = phi i64 [ %n.vec, %middle.block ], [ 0, %L28.lr.ph ]
    %bc.merge.rdx = phi double [ %25, %middle.block ], [ 0.000000e+00, %L28.lr.ph ]
    br label %L28

L28:                                              ; preds = %scalar.ph, %L28
    %value_phi215 = phi i64 [ %bc.resume.val, %scalar.ph ], [ %30, %L28 ]
    %value_phi14 = phi double [ %bc.merge.rdx, %scalar.ph ], [ %29, %L28 ]
    ; | @ simdloop.jl:77 within `macro expansion' @ In[21]:4
    ; | r @ int.jl:54 within `*'
      %26 = shl nuw i64 %value_phi215, 1
    ; | L
    ; | r @ int.jl:52 within `-'
      %27 = or i64 %26, 1
    ; | L
    ; | r @ promotion.jl:311 within `+'
    ; | r @ promotion.jl:282 within `promote'
    ; | r @ promotion.jl:259 within `_promote'
    ; | r @ number.jl:7 within `convert'
    ; | r @ float.jl:60 within `Float64'
      %28 = sitofp i64 %27 to double
    ; | L L L L
    ; | @ promotion.jl:311 within `+' @ float.jl:401
      %29 = fadd fast double %value_phi14, %28
    ; | L
    ; | @ simdloop.jl:78 within `macro expansion'
    ; | r @ int.jl:53 within `+'
      %30 = add nuw nsw i64 %value_phi215, 1
Let's see what happens if we sum slightly more complicated expressions.
In [19]:

```python
f(i) = (i+10)*(i+9)*(i+6) / ((i)*(i+1)*(i+3))

function sum_f(n)
    s = 0.0
    for i in 1:n
        s += f(i)
    end
    return s
end
```

Out[19]:

```
sum_f (generic function with 1 method)
```

In [20]:

```python
@btime sum_f($n)

204.939 ms (0 allocations: 0 bytes)
```

Out[20]:

```
9.930874690005353e7
```

It takes a little bit longer, but not as much as you would expect. Note that we can speed the simple loop a little. This trick does not help the more complicated one.

In [21]:

```python
function sum_to_n(n)
    s = 0.0
    @simd for i in 1:n
        s += 2*i-1
    end
    return s
end

@btime sum_to_n($n)

76.570 ms (0 allocations: 0 bytes)
```

Out[21]:

```
1.0e16
```

Let's time summing n random floats and n random integers.
In [22]:

```python
x_float = rand(n)
x_int = rand(1:1000,n)
```

Out[22]:

```
100000000-element Array{Int64,1}:
  877
  176
  41
  619
  47
  839
  209
  636
  941
  390
  781
  39
  190
  ·
  171
  749
  158
  416
  516
  497
  538
  852
  628
  563
  707
  892
```

In [23]:

```python
# slightly fancy: returns same data type as input
function sum_vector(x)
    s = zero(x[1])
    for xi in x
        s += xi
    end
    return s
end
```

Out[23]:

```
sum_vector (generic function with 1 method)
```
In [24]:

@@btime sum_vector($x\_int)$

   41.052 ms (0 allocations: 0 bytes)

Out[24]:

50043005967

In [25]:

@@btime sum_vector($x\_float)$

   127.808 ms (0 allocations: 0 bytes)

Out[25]:

4.999616772670455e7

We see that adding ints is a little faster than adding floats. And, the memory access costs almost nothing. It’s like I lied. There are two reasons:

- The cache lines each hold 8 numbers. So, only the first of every 8 is a cache miss.
- The cache notices that we are fetching in order, and starts sending data before it is requested (probably).

So, let’s compute the sums in a random order. This should cause a lot more cache misses.

Note that the @@ things are optimizations. You could remove them and get good code.

In [26]:

function sum_vector(x, order)
    @assert length(x) == length(order)
    s = zero(x[1])
    @inbounds for i in 1:length(x)
        s += x[order[i]]
    end
    return s
end

Out[26]:

sum_vector (generic function with 2 methods)
In [27]:

```plaintext
using Random
Random.seed!(0)  # Not necessary, but makes results reproducible
p = randperm(n)
```

Out[27]:

```
100000000-element Array{Int64,1}:
  49597440
  79027566
  88211541
  1797603
  97478646
  15077832
  76931792
  33247206
  90403623
  53797768
  75267254
  5756903
  96704086
  ...
  48614943
  24796104
  96173030
  3356869
  26704510
  47711596
  13455416
  82239290
  32275046
  80546305
  55059727
  58635446
```

In [28]:

```plaintext
@benchmark sum_vector($x_int,$p)
```

Out[28]:

```
BenchmarkTools.Trial:
  memory estimate:  0 bytes
  allocs estimate:  0
  ---------------
  minimum time:    4.627 s (0.00% GC)
  median time:     4.727 s (0.00% GC)
  mean time:       4.727 s (0.00% GC)
  maximum time:    4.827 s (0.00% GC)
  ---------------
  samples:         2
  evals/sample:    1
```
I can't explain why those took such different amounts of time. I do know that if you are going to spend that much time in memory access, then you can fit in a lot of computation with the data that you do retrieve without taking much longer.

In [30]:

```python
function sum_vector_f(x, order)
    @assert length(x) == length(order)
    s = zero(x[1])
    @inbounds for i in 1:length(x)
        s += f(x[order[i]])
    end
    return s
end
```

Out[30]:

```
sum_vector_f (generic function with 1 method)
```
In [31]:
@benchmark sum_vector_f($x_int,$p)

Out[31]:
BenchmarkTools.Trial:
  memory estimate:  0 bytes
  allocs estimate:  0
  ---------------
  minimum time:    5.334 s (0.00% GC)
  median time:     5.334 s (0.00% GC)
  mean time:       5.334 s (0.00% GC)
  maximum time:    5.334 s (0.00% GC)
  ---------------
  samples:         1
  evals/sample:    1

In [32]:
@benchmark sum_vector_f($x_float,$p)

Out[32]:
BenchmarkTools.Trial:
  memory estimate:  0 bytes
  allocs estimate:  0
  ---------------
  minimum time:    3.680 s (0.00% GC)
  median time:     3.680 s (0.00% GC)
  mean time:       3.680 s (0.00% GC)
  maximum time:    3.680 s (0.00% GC)
  ---------------
  samples:         2
  evals/sample:    1

**Sparse Matrices**

If you want to write fast code involving sparse matrices, then you have to pay attention to how they are stored.

The standard in Julia and Matlab is Compressed Column Format.

This essentially means that the locations of the nonzero entries are stored. Here's an example of a sparse matrix, but we first create it dense so you can see it.
Random.seed!(0)
M = rand(8,8) .< 0.2

Out[33]:

8×8 BitArray{2}:
0 0 0 1 0 0 1 0
0 0 0 0 0 0 0 0
1 0 0 1 1 1 1 0
1 0 0 1 0 1 0 0
0 0 0 0 0 1 0 1
0 0 1 1 1 0 0 0
1 1 0 0 0 1 0 0
1 0 0 0 0 0 0 0

In [35]:
S = sparse(M)

Out[35]:

8×8 SparseMatrixCSC{Bool,Int64} with 19 stored entries:
[3, 1]  =  1
[4, 1]  =  1
[7, 1]  =  1
[8, 1]  =  1
[7, 2]  =  1
[6, 3]  =  1
[1, 4]  =  1
[3, 4]  =  1
[4, 4]  =  1
[6, 4]  =  1
[3, 5]  =  1
[6, 5]  =  1
[3, 6]  =  1
[4, 6]  =  1
[5, 6]  =  1
[7, 6]  =  1
[1, 7]  =  1
[3, 7]  =  1
[5, 7]  =  1

As you can see, the sparse format just records the nonzero entries. Let's make them vary so we can better distinguish them.
S is stored in three arrays. I suggest reading about the CSC format to understand them. For now, just know that one contains the indices of the rows with nonzeros in each column, and another stores the nonzero entries.
In [37]:
S.nzval

Out[37]:
19-element Array{Int64,1}:
   4
   43
   49
   41
   30
   46
   27
   8
   66
   33
   38
   66
   82
   54
   95
   72
   46
    9
   31

In [38]:
[S.rowval S.nzval]

Out[38]:
19×2 Array{Int64,2}:
  3  4
  4 43
  7 49
  8 41
  7 30
  6 46
  1 27
  3  8
  4 66
  6 33
  3 38
  6 66
  3 82
  4 54
  5 95
  7 72
  1 46
  3  9
  5 31
The moral is that one should, whenever possible, perform operations on columns instead of rows.

```
In [40]:

function col_sums(S)
    n = size(S, 2)
    s = zeros(n)
    for i in 1:n
        s[i] = sum(S[:,i])  # the ith column
    end
    return s
end

function row_sums(S)
    n = size(S, 1)
    s = zeros(n)
    for i in 1:n
        s[i] = sum(S[i,:])  # the ith column
    end
    return s
end
```

Out[40]:

row_sums (generic function with 1 method)
To see this, let’s create a large sparse matrix. It will be $n$ by $n$ with $m$ entries. Entries will be random numbers between 1 and 100.

```
In [42]:
n = 10_000
m = 100_000
M = sparse(rand(1:n,m), rand(1:n,m), rand(1:100, m))
```

```
8×2 Array{Float64,2}:
 137.0  73.0
 30.0   0.0
 46.0  141.0
134.0  163.0
104.0  126.0
303.0  145.0
 55.0  151.0
 31.0  41.0
```
In [43]:

```python
@benchmark s = col_sums(M)
```

Out[43]:

```
BenchmarkTools.Trial:
  memory estimate:  3.51 MiB
  allocs estimate: 30002
  ---------------
  minimum time:   1.213 ms (0.00% GC)
  median time:    1.440 ms (0.00% GC)
  mean time:      2.172 ms (30.16% GC)
  maximum time:   232.741 ms (99.32% GC)
  ---------------
  samples:        2307
  evals/sample:   1
```
That is a 1000-fold difference. If you really need the row-sums, it is easier to compute column sums of the matrix transpose; although, computing the transpose is not all that fast.

The time of multiplying a vector by a matrix is similar either way you do it. Note: we could usually write this as \( y = Mx \). We write it in a functional form for timing.
In [61]:

xt = x'
@btime yt = *($xt, $M)
;

171.752 μs (4 allocations: 78.23 KiB)