Jupyter Julia Notebooks from first lecture of S&DS 631.

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
```

In [6]:

```
y ≈ 1.21
```

Out[6]:

```
true
```

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

Timing elementary ops and memory references

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:
       speed         user         nice          sys         idle
irq
#1  3300 MHz     631285 s          0 s     231768 s    1416786 s
0 s
#2  3300 MHz     247144 s          0 s      90263 s    1941905 s
0 s
#3  3300 MHz     616553 s          0 s     202959 s    1459805 s
0 s
#4  3300 MHz     242469 s          0 s      83257 s    1953586 s
0 s
```

So, expect 3 * 10^9 ops per second, approximately.

In [10]:

```
using Hwloc
```

In [11]:

```
topology = Hwloc.topology_load()
println("Machine topology:")
print(topology)
```

```
Machine topology:
D0: L0 P0 Machine
    D1: L0 P0 Package
        D2: L0 P-1 L3Cache  Cache{size=4194304,depth=3,linesize=64,a
ssociativity=0,type=Unified}
            D3: L0 P-1 L2Cache  Cache{size=262144,depth=2,linesize=6
4,associativity=4,type=Unified}
                D4: L0 P-1 L1Cache  Cache{size=32768,depth=1,linesiz
e=64,associativity=0,type=Data}
                    D5: L0 P0 Core
                        D6: L0 P0 PU
                        D6: L1 P1 PU
            D3: L1 P-1 L2Cache  Cache{size=262144,depth=2,linesize=6
4,associativity=4,type=Unified}
                D4: L1 P-1 L1Cache  Cache{size=32768,depth=1,linesiz
e=64,associativity=0,type=Data}
                    D5: L1 P1 Core
                        D6: L2 P2 PU
                        D6: L3 P3 PU
```

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]:

```
100000000
```

Summation in a loop

Here is a simple function that sums odd integers (as floats)

In [14]:

```
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]:

```
sum_to_n(10)
```

Out[15]:

```
100.0
```

In [16]:

```
@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)")

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

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]:

```
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]:

```
@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]:

```
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]:

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

```
;  @ In[49]:2 within `sum_to_n_ints'
define i64 @julia_sum_to_n_ints_18616(i64) {
top:
;  @ In[49]:3 within `sum_to_n_ints'
; ┌ @ range.jl:5 within `Colon'
; │┌ @ range.jl:277 within `UnitRange'
; ││┌ @ range.jl:282 within `unitrange_last'
; │││┌ @ operators.jl:341 within `>='
; ││││┌ @ int.jl:424 within `<='
       %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
;  @ In[49]:6 within `sum_to_n_ints'
  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]:

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

```
;  @ In[21]:2 within `sum_to_n'
define double @julia_sum_to_n_18617(i64) {
top:
;  @ In[21]:3 within `sum_to_n'
; ┌ @ simdloop.jl:69 within `macro expansion'
; │┌ @ range.jl:5 within `Colon'
; ││┌ @ range.jl:277 within `UnitRange'
; │││┌ @ range.jl:282 within `unitrange_last'
; ││││┌ @ operators.jl:341 within `>='
; │││││┌ @ int.jl:424 within `<='
        %1 = icmp sgt i64 %0, 0
; ││││└└
      %2 = select i1 %1, i64 %0, i64 0
; │└└└
; │ @ simdloop.jl:71 within `macro expansion'
```

```
; │┌ @ simdloop.jl:51 within `simd_inner_length'
; ││┌ @ range.jl:543 within `length'
; │││┌ @ checked.jl:222 within `checked_sub'
; ││││┌ @ checked.jl:194 within `sub_with_overflow'
       %3 = add nsw i64 %2, -1
; │││└└
; │││┌ @ checked.jl:165 within `checked_add'
; ││││┌ @ checked.jl:132 within `add_with_overflow'
       %4 = call { i64, i1 } @llvm.sadd.with.overflow.i64(i64 %3, i6
4 1)
       %5 = extractvalue { i64, i1 } %4, 1
; ││││└
; ││││ @ checked.jl:166 within `checked_add'
      br i1 %5, label %L16, label %L21

L16:                                              ; preds = %top
      call void @julia_throw_overflowerr_binaryop_15477(%jl_value_t
addrspace(10)* addrspacecast (%jl_value_t* inttoptr (i64 4613719168
to %jl_value_t*) to %jl_value_t addrspace(10)*), i64 %3, i64 1)
      call void @llvm.trap()
      unreachable

L21:                                              ; preds = %top
; ││││ @ checked.jl:165 within `checked_add'
; ││││┌ @ checked.jl:132 within `add_with_overflow'
       %6 = extractvalue { i64, i1 } %4, 0
; │└└└└
; │ @ simdloop.jl:72 within `macro expansion'
; │┌ @ int.jl:49 within `<'
    %7 = icmp slt i64 %6, 1
; │└
   br i1 %7, label %L55, label %L28.lr.ph

L28.lr.ph:                                        ; preds = %L21
; │ @ simdloop.jl:75 within `macro expansion'
   %min.iters.check = icmp ult i64 %2, 16
   br i1 %min.iters.check, label %scalar.ph, label %vector.ph

vector.ph:                                        ; preds = %L28.lr.
ph
   %n.vec = and i64 %2, 9223372036854775792
   br label %vector.body

vector.body:                                      ; preds = %vector.
body, %vector.ph
; │ @ simdloop.jl:78 within `macro expansion'
; │┌ @ int.jl:53 within `+'
    %index = phi i64 [ 0, %vector.ph ], [ %index.next, %vector.body
]
    %vec.ind = phi <4 x i64> [ <i64 0, i64 1, i64 2, i64 3>, %vector
.ph ], [ %vec.ind.next, %vector.body ]
    %vec.phi = phi <4 x double> [ zeroinitializer, %vector.ph ], [ %
20, %vector.body ]
```

```
    %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 `+'
; ││┌ @ promotion.jl:282 within `promote'
; │││┌ @ promotion.jl:259 within `_promote'
; ││││┌ @ number.jl:7 within `convert'
; │││││┌ @ 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>
; ││└└└└
; ││ @ promotion.jl:311 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
; │└
; │ @ simdloop.jl:78 within `macro expansion'
; │┌ @ int.jl:53 within `+'
    %index.next = add i64 %index, 16
    %vec.ind.next = add <4 x i64> %vec.ind, <i64 16, i64 16, i64 16,
i64 16>
    %24 = icmp eq i64 %index.next, %n.vec
    br i1 %24, label %middle.block, label %vector.body

middle.block:                                     ; preds = %vector.
body
```

```
; │└
; │ @ simdloop.jl:77 within `macro expansion' @ In[21]:4
; │┌ @ 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
    %rdx.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, %rdx.shuf
    %rdx.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, %rdx.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, %L
28 ]
   %value_phi14 = phi double [ %bc.merge.rdx, %scalar.ph ], [ %29, %
L28 ]
; │ @ simdloop.jl:77 within `macro expansion' @ In[21]:4
; │┌ @ int.jl:54 within `*'
    %26 = shl nuw i64 %value_phi215, 1
; │└
; │┌ @ int.jl:52 within `-'
    %27 = or i64 %26, 1
; │└
; │┌ @ promotion.jl:311 within `+'
; ││┌ @ promotion.jl:282 within `promote'
; │││┌ @ promotion.jl:259 within `_promote'
; ││││┌ @ number.jl:7 within `convert'
; │││││┌ @ float.jl:60 within `Float64'
        %28 = sitofp i64 %27 to double
; ││└└└└
; ││ @ promotion.jl:311 within `+' @ float.jl:401
    %29 = fadd fast double %value_phi14, %28
; │└
; │ @ simdloop.jl:78 within `macro expansion'
; │┌ @ int.jl:53 within `+'
    %30 = add nuw nsw i64 %value_phi215, 1
; │└
```

```
; │ @ simdloop.jl:75 within `macro expansion'
; │┌ @ int.jl:49 within `<'
    %31 = icmp ult i64 %30, %6
; │└
   br i1 %31, label %L28, label %L55

L55:                                              ; preds = %L28, %m
iddle.block, %L21
   %value_phi5 = phi double [ 0.000000e+00, %L21 ], [ %29, %L28 ], [
%25, %middle.block ]
; └
;  @ In[21]:6 within `sum_to_n'
  ret double %value_phi5
}
```

In [57]:

```
@code_native sum_to_n_ints(n2)
```

```
        .section        __TEXT,__text,regular,pure_instructions
; ┌ @ In[49]:3 within `sum_to_n_ints'
; │┌ @ range.jl:5 within `Colon'
; ││┌ @ range.jl:277 within `UnitRange'
; │││┌ @ range.jl:282 within `unitrange_last'
; ││││┌ @ operators.jl:341 within `>='
; │││││┌ @ In[49]:2 within `<='
        testq   %rdi, %rdi
; │└└└└└
        jle     L33
        leaq    (%rdi,%rdi,2), %rax
        leaq    -1(%rdi), %rcx
        addq    $-2, %rdi
        imulq   %rdi, %rcx
        andq    $-2, %rcx
        addq    %rcx, %rax
        addq    $-2, %rax
; │ @ In[49]:6 within `sum_to_n_ints'
        retq
L33:
        xorl    %eax, %eax
; │ @ In[49]:6 within `sum_to_n_ints'
        retq
        nopw    %cs:(%rax,%rax)
; └
```

Back to the story

Let's see what happens if we sum slightly more complicated expressions.

In [19]:

```
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]:

```
@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]:

```
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]:

```
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]:

```
# 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]:

```julia
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]:

```julia
@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
```

In [29]:

```
@benchmark sum_vector($x_float,$p)
```

Out[29]:

```
BenchmarkTools.Trial:
  memory estimate:  0 bytes
  allocs estimate:  0
  --------------
  minimum time:     3.667 s (0.00% GC)
  median time:      3.877 s (0.00% GC)
  mean time:        3.877 s (0.00% GC)
  maximum time:     4.088 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]:

```
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.

In [33]:

```
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, 8]  =  1
```

As you can see, the sparse format just records the nonzero entries. Let's make them vary so we can better distinguish them.

In [36]:

```
S = S .* rand(1:100,8,8)
```

Out[36]:

```
8×8 SparseMatrixCSC{Int64,Int64} with 19 stored entries:
  [3, 1]  =  4
  [4, 1]  =  43
  [7, 1]  =  49
  [8, 1]  =  41
  [7, 2]  =  30
  [6, 3]  =  46
  [1, 4]  =  27
  [3, 4]  =  8
  [4, 4]  =  66
  [6, 4]  =  33
  [3, 5]  =  38
  [6, 5]  =  66
  [3, 6]  =  82
  [4, 6]  =  54
  [5, 6]  =  95
  [7, 6]  =  72
  [1, 7]  =  46
  [3, 7]  =  9
  [5, 8]  =  31
```

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
```

In [39]:

```
S.colptr
```

Out[39]:

```
9-element Array{Int64,1}:
  1
  5
  6
  7
 11
 13
 17
 19
 20
```

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)
```

In [41]:

```
[col_sums(S) row_sums(S)]
```

Out[41]:

```
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
```

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))
```

Out[42]:

```
10000×10000 SparseMatrixCSC{Int64,Int64} with 99943 stored entries:
  [16  ,     1]  =  63
  [827 ,     1]  =  89
  [2010,     1]  =  54
  [2158,     1]  =  7
  [3968,     1]  =  85
  [6204,     1]  =  60
  [8651,     1]  =  45
  [9667,     1]  =  80
  [1111,     2]  =  74
  [2341,     2]  =  66
  [3367,     2]  =  19
  [3518,     2]  =  29
  ⋮
  [7451,  9999]  =  23
  [7546,  9999]  =  7
  [8126,  9999]  =  99
  [8317,  9999]  =  14
  [9049,  9999]  =  19
  [9943,  9999]  =  93
  [9986,  9999]  =  36
  [276 , 10000]  =  31
  [311 , 10000]  =  33
  [3965, 10000]  =  16
  [5973, 10000]  =  74
  [6390, 10000]  =  90
  [7328, 10000]  =  12
```

In [43]:

```
@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
```

In [44]:

```
@benchmark s = row_sums($M)
```

Out[44]:

```
BenchmarkTools.Trial:
  memory estimate:  6.28 MiB
  allocs estimate:  83316
  --------------
  minimum time:     1.636 s (0.00% GC)
  median time:      1.776 s (0.00% GC)
  mean time:        1.781 s (0.00% GC)
  maximum time:     1.931 s (0.00% GC)
  --------------
  samples:          3
  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.

In [46]:

```
@time Mt = Matrix(M');
```

```
  3.288212 seconds (7 allocations: 762.940 MiB, 0.23% gc time)
```

In [48]:

```
s1 = row_sums(M)
s2 = col_sums(Mt)
using LinearAlgebra
norm(s1-s2)
```

Out[48]:

```
0.0
```

The time of multiplying a vector by a matrix is similar either way you do it. Note: we could usually write this as `y = M*x`. We write it in a functional form for timing.

In [60]:

```
x = randn(n)
@btime y = *($M, $x)
;
```

```
  226.428 μs (2 allocations: 78.20 KiB)
```

In [61]:

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

```
  171.752 μs (4 allocations: 78.23 KiB)
```

In [ ]:

```
```