Did It Go Faster?

A short introduction to the BenchmarkTools package

Did It Go Faster?

A short introduction to the BenchmarkTools package

What is BenchmarkTools?

- A Julia package that makes **performance tracking of Julia code easy** by supplying a framework for **writing and running groups of benchmarks** as well as **comparing benchmark results**.

- `julia> Pkg.add("BenchmarkTools")`

- Drives Julia's Base library performance testing CI (~2000 benchmarks)

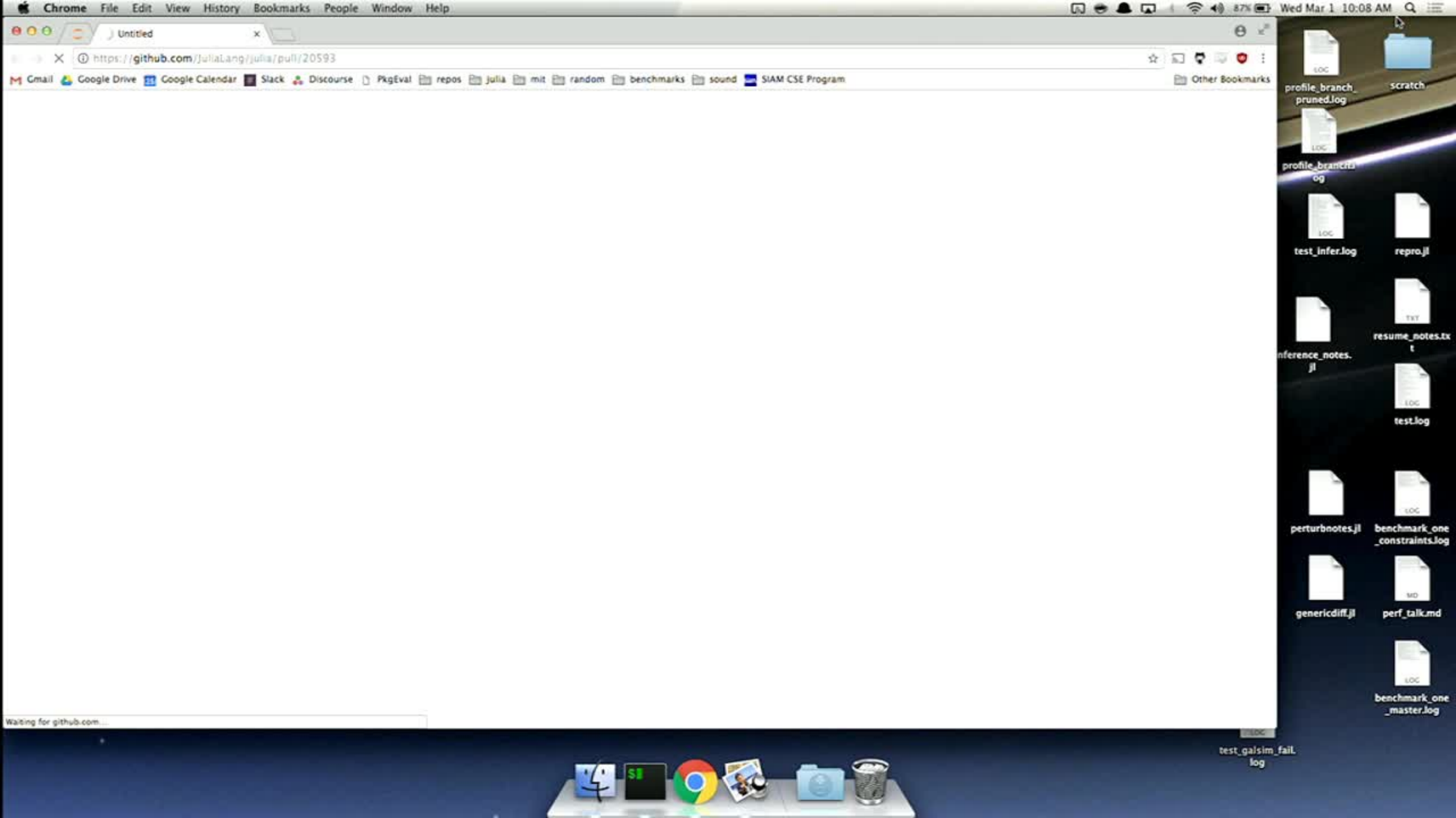
Why BenchmarkTools? (A.K.A. “What about @time?”)

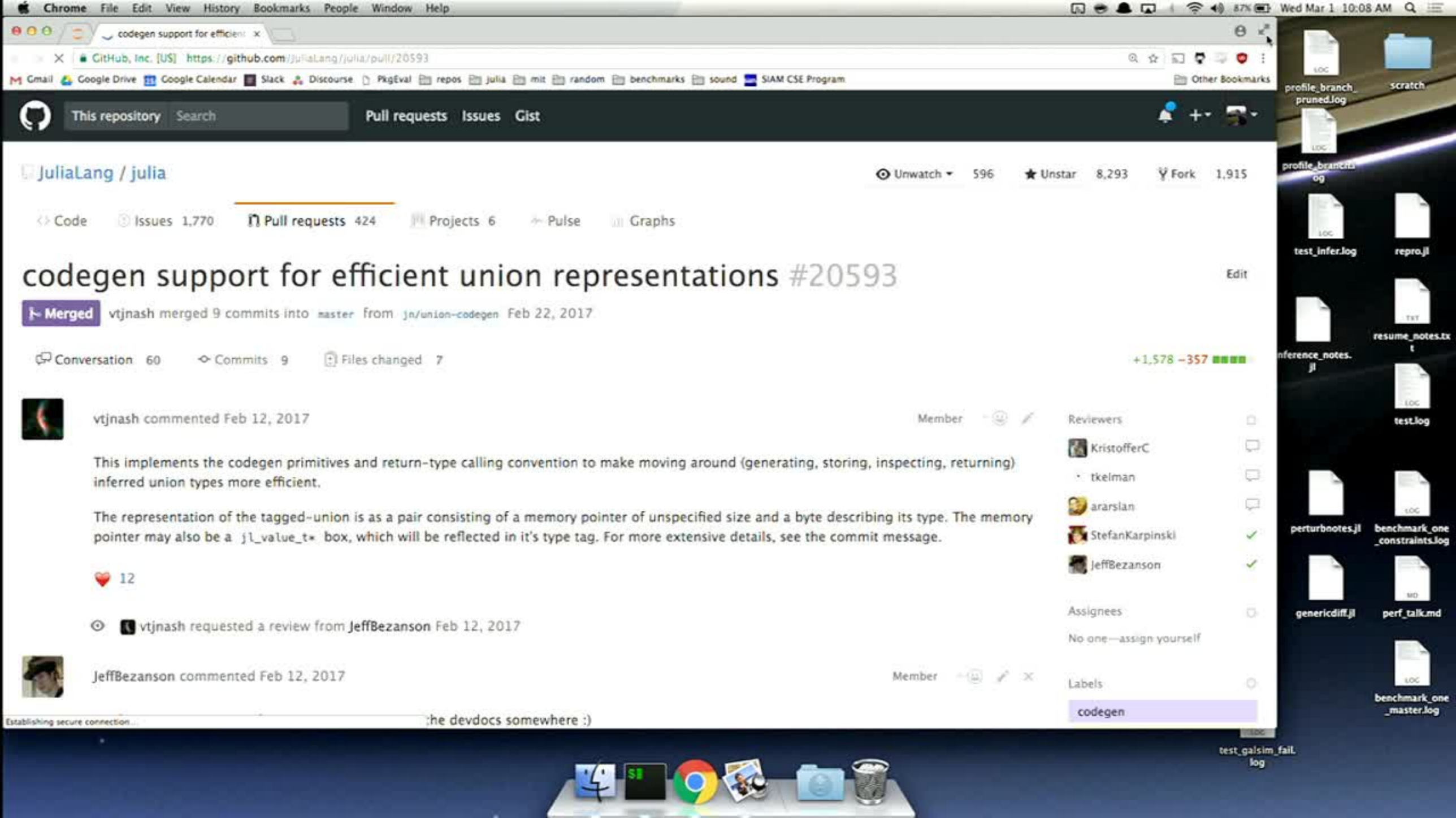
```
julia> @time sum(rand(Bool) for _ in 1:1000);
  0.037500 seconds (22.91 k allocations: 1.072 MiB)


# BenchmarkTools version of `@time`
julia> @btime sum(rand(Bool) for _ in 1:1000);
  6.615 µs (2 allocations: 64 bytes)


julia> f(n) = sum(rand(Bool) for _ in 1:n)
f (generic function with 1 method)


julia> @time f(1000);
  0.000012 seconds (6 allocations: 224 bytes)
```

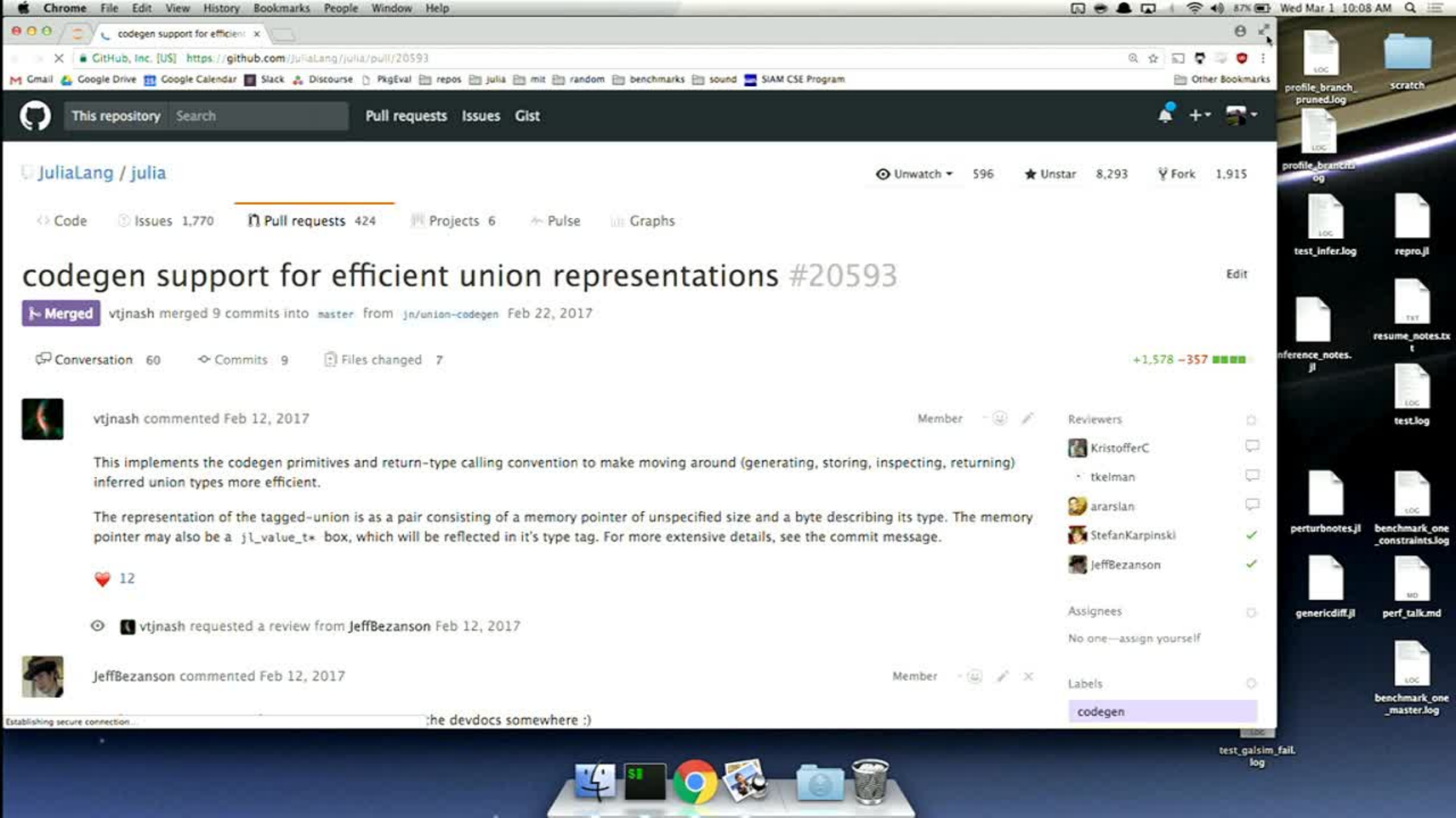
Okay, but what about in the real world?

https://github.com/JuliaLang/julia/pull/20593

https://github.com/JuliaLang/julia/pull/20593


Chrome File Edit View History Bookmarks People Window Help
Wed Mar 1 10:08 AM
Untitled
https://github.com/JuliaLang/julia/pull/20593
Gmail Google Drive Google Calendar Slack Discourse PkgEval repos julia mit random benchmarks sound SIAM CSE Program
Other Bookmarks
Waiting for github.com...
profile_branch_pruned.log
scratch
profile_branch.log
test_infer.log
repro.jl
nference_notes.jl
resume_notes.txt
test.log
perturbnotes.jl
benchmark_one_constraints.log
genericdiff.jl
perf_talk.md
benchmark_one_master.log
test_galsim_fail.log


Chrome File Edit View History Bookmarks People Window Help
Wed Mar 1 10:08 AM
codegen support for efficient
GitHub, Inc. [US] https://github.com/JuliaLang/julia/pull/20593
Gmail Google Drive Google Calendar Slack Discourse PkgEval repos julia mit random benchmarks sound SIAM CSE Program Other Bookmarks
This repository Search
Pull requests Issues Gist
JuliaLang / julia
Unwatch 596 Unstar 8,293 Fork 1,915
Code Issues 1,770 Pull requests 424 Projects 6 Pulse Graphs
codegen support for efficient union representations #20593
Edit
Merged vtjnash merged 9 commits into master from jn/union-codegen Feb 22, 2017
Conversation 60 Commits 9 Files changed 7
+1,578 −357
vtjnash commented Feb 12, 2017
Member
This implements the codegen primitives and return-type calling convention to make moving around (generating, storing, inspecting, returning) inferred union types more efficient.
The representation of the tagged-union is as a pair consisting of a memory pointer of unspecified size and a byte describing its type. The memory pointer may also be a jl_value_t* box, which will be reflected in it's type tag. For more extensive details, see the commit message.
12
vtjnash requested a review from JeffBezanson Feb 12, 2017
JeffBezanson commented Feb 12, 2017
Member
:he devdocs somewhere :)
Establishing secure connection...
Reviewers
KristofferC
tkelman
ararslan
StefanKarpinski
JeffBezanson
Assignees
No one—assign yourself
Labels
codegen
profile_branch_pruned.log
scratch
profile_branch.log
test_infer.log
repro.jl
inference_notes.jl
resume_notes.txt
test.log
perturbnotes.jl
benchmark_one_constraints.log
genericdiff.jl
perf_talk.md
benchmark_one_master.log
test_galsim_fail.log


Chrome File Edit View History Bookmarks People Window Help
Wed Mar 1 10:08 AM
codegen support for efficient
GitHub, Inc. [US] https://github.com/JuliaLang/julia/pull/20593
Gmail Google Drive Google Calendar Slack Discourse PkgEval repos julia mit random benchmarks sound SIAM CSE Program
Other Bookmarks
This repository Search
Pull requests Issues Gist
JuliaLang / julia
Unwatch 596 Unstar 8,293 Fork 1,915
Code Issues 1,770 Pull requests 424 Projects 6 Pulse Graphs
codegen support for efficient union representations #20593
Edit
Merged vtjnash merged 9 commits into master from jn/union-codegen Feb 22, 2017
Conversation 60 Commits 9 Files changed 7
+1,578 −357
vtjnash commented Feb 12, 2017
Member
This implements the codegen primitives and return-type calling convention to make moving around (generating, storing, inspecting, returning) inferred union types more efficient.
The representation of the tagged-union is as a pair consisting of a memory pointer of unspecified size and a byte describing its type. The memory pointer may also be a jl_value_t* box, which will be reflected in it's type tag. For more extensive details, see the commit message.
12
vtjnash requested a review from JeffBezanson Feb 12, 2017
JeffBezanson commented Feb 12, 2017
Member
:he devdocs somewhere :)
Establishing secure connection...
Reviewers
KristofferC
tkelman
ararslan
StefanKarpinski
JeffBezanson
Assignees
No one—assign yourself
Labels
codegen
profile_branch_pruned.log
scratch
profile_branch.log
test_infer.log
repro.jl
inference_notes.jl
resume_notes.txt
test.log
perturbnotes.jl
benchmark_one_constraints.log
genericdiff.jl
perf_talk.md
benchmark_one_master.log
test_galsim_fail.log

JuliaLang / julia

Unwatch 596 Unstar 8,293 Fork 1,915

Code Issues 1,770 Pull requests 424 Projects 6 Pulse Graphs

codegen support for efficient union representations #20593

Edit

Merged vtjnash merged 9 commits into `master` from `jn/union-codegen` 7 days ago

Conversation 60 Commits 9 Files changed 7 +1,578 −357

vtjnash commented 16 days ago Member

This implements the codegen primitives and return-type calling convention to make moving around (generating, storing, inspecting, returning) inferred union types more efficient.

The representation of the tagged-union is as a pair consisting of a memory pointer of unspecified size and a byte describing its type. The memory pointer may also be a `jl_value_t*` box, which will be reflected in it's type tag. For more extensive details, see the commit message.

12

vtjnash requested a review from JeffBezanson 16 days ago

JeffBezanson commented 16 days ago Member

Cool! First request: copy that commit message into the devdocs somewhere :)

1 1

johnmyleswhite commented 16 days ago Member

So excited to see this being worked on. Am I right to think there are still performance optimizations that can be made? For now, I'm using the following snippet to check the performance of union types:

Reviewers

KristofferC

tkelman

ararslan

StefanKarpinski

JeffBezanson

Assignees

No one—assign yourself

Labels

codegen

types and dispatch

Projects

None yet

Milestone

0.6.0

codegen support for efficient union representations #20593

Edit

Merged vtjnash merged 9 commits into master from jn/union-codegen 7 days ago

Conversation 60 Commits 9 Files changed 7 +1,578 −357

vtjnash commented 16 days ago Member

This implements the codegen primitives and return-type calling convention to make moving around (generating, storing, inspecting, returning) inferred union types more efficient.

The representation of the tagged-union is as a pair consisting of a memory pointer of unspecified size and a byte describing its type. The memory pointer may also be a `jl_value_t*` box, which will be reflected in it's type tag. For more extensive details, see the commit message.

❤ 12

vtjnash requested a review from JeffBezanson 16 days ago

JeffBezanson commented 16 days ago Member

Cool! First request: copy that commit message into the devdocs somewhere :)

👍 1 😄 1

johnmyleswhite commented 16 days ago Member

So excited to see this being worked on. Am I right to think there are still performance optimizations that can be made? For now, I'm using the following snippet to check the performance of union types:

```
function stable(n)
    s = 0.0
    for i in 1:n
```

Reviewers

KristofferC

tkelman

ararslan

StefanKarpinski ✓

JeffBezanson ✓

Assignees

No one—assign yourself

Labels

codegen

types and dispatch

Projects

None yet

Milestone

0.6.0

Notifications

Unsubscribe

This implements the codegen primitives and return-type calling convention to make moving around (generating, storing, inspecting, returning) inferred union types more efficient.

The representation of the tagged-union is as a pair consisting of a memory pointer of unspecified size and a byte describing its type. The memory pointer may also be a `jl_value_t*` box, which will be reflected in it's type tag. For more extensive details, see the commit message.

❤️ 12 +😀

vtjnash requested a review from JeffBezanson 16 days ago

JeffBezanson commented 16 days ago Member

Cool! First request: copy that commit message into the devdocs somewhere :)

👍 1 😀 1

johnmyleswhite commented 16 days ago Member

So excited to see this being worked on. Am I right to think there are still performance optimizations that can be made? For now, I'm using the following snippet to check the performance of union types:

```
function stable(n)
    s = 0.0
    for i in 1:n
        s += sin(1.0)
    end
    return s
end

function unstable(n)
    s = 0
    for i in 1:n
        s += sin(1.0)
    end
    return s
end

@time stable(10_000_000)
@time unstable(10_000_000)
```

Currently this branch produces very similar behavior as Julia 0.5 did: the union type of `s` in `unstable` causes about a 2x latency degradation and allocates a lot of memory.

KristofferC
tkelman
ararslan
StefanKarpinski ✓
JeffBezanson ✓

Assignees
No one—assign yourself

Labels
codegen
types and dispatch

Projects
None yet

Milestone
0.6.0

Notifications
Unsubscribe
You're receiving notifications because you were mentioned.

11 participants

Lock conversation

Cool! First request: copy that commit message into the devdocs somewhere :)

1 1

johnmyleswhite commented 16 days ago — Member

So excited to see this being worked on. Am I right to think there are still performance optimizations that can be made? For now, I'm using the following snippet to check the performance of union types:

```
function stable(n)
    s = 0.0
    for i in 1:n
        s += sin(1.0)
    end
    return s
end

function unstable(n)
    s = 0
    for i in 1:n
        s += sin(1.0)
    end
    return s
end

@time stable(10_000_000)
@time unstable(10_000_000)
```

Currently this branch produces very similar behavior as Julia 0.5 did: the union type of `s` in `unstable` causes about a 2x latency degradation and allocates a lot of memory.

andyferris commented 16 days ago — Contributor

Very cool, Jameson. I'm guessing from the OP and John's comment that the remaining half of work is to make unions of two isbits types isbits themselves? Would these be laid out be like a variant/ `Switch` , or overlapped like a C union?

vtjnash commented 16 days ago — Member

That's a boring example. I can cut down the allocation count, but they don't really cost any time relative to `sin` . The performance variation seen here is entirely due to a processor bug / feature. If you `@profile` both, you'll see that >99% of time in all versions is spent in the `sin` kernel:

codegen

types and dispatch

Projects

None yet

Milestone

0.6.0

Notifications

Unsubscribe

You're receiving notifications because you were mentioned.

11 participants

Lock conversation

Cool! First request: copy that commit message into the devdocs somewhere :)

1 1

johnmyleswhite commented 16 days ago — Member

So excited to see this being worked on. Am I right to think there are still performance optimizations that can be made? For now, I'm using the following snippet to check the performance of union types:

```
function stable(n)
    s = 0.0
    for i in 1:n
        s += sin(1.0)
    end
    return s
end

function unstable(n)
    s = 0
    for i in 1:n
        s += sin(1.0)
    end
    return s
end

@time stable(10_000_000)
@time unstable(10_000_000)
```

Currently this branch produces very similar behavior as Julia 0.5 did: the union type of `s` in `unstable` causes about a 2x latency degradation and allocates a lot of memory.

andyferris commented 16 days ago — Contributor

Very cool, Jameson. I'm guessing from the OP and John's comment that the remaining half of work is to make unions of two isbits types isbits themselves? Would these be laid out be like a variant/ `Switch` , or overlapped like a C union?

vtjnash commented 16 days ago — Member

That's a boring example. I can cut down the allocation count, but they don't really cost any time relative to `sin` . The performance variation seen here is entirely due to a processor bug / feature. If you `@profile` both, you'll see that >99% of time in all versions is spent in the `sin` kernel:

codegen

types and dispatch

Projects

None yet

Milestone

0.6.0

Notifications

Unsubscribe

You're receiving notifications because you were mentioned.

11 participants

Lock conversation

performance of union types:

```
function stable(n)
    s = 0.0
    for i in 1:n
        s += sin(1.0)
    end
    return s
end

function unstable(n)
    s = 0
    for i in 1:n
        s += sin(1.0)
    end
    return s
end

@time stable(10_000_000)
@time unstable(10_000_000)
```

Currently this branch produces very similar behavior as Julia 0.5 did: the union type of `s` in `unstable` causes about a 2x latency degradation and allocates a lot of memory.

andyferris commented 16 days ago — Contributor

Very cool, Jameson. I'm guessing from the OP and John's comment that the remaining half of work is to make unions of two isbits types isbits themselves? Would these be laid out be like a variant/ `Switch` , or overlapped like a C union?

vtjnash commented 16 days ago — Member

That's a boring example. I can cut down the allocation count, but they don't really cost any time relative to `sin` . The performance variation seen here is entirely due to a processor bug / feature. If you `@profile` both, you'll see that >99% of time in all versions is spent in the `sin` kernel:

```
julia> @time stable(10_000_000) # comparison
  0.153794 seconds (5 allocations: 176 bytes)
8.414709847530957e6

julia> @time unstable(10_000_000) # altered version of PR
  0.318522 seconds (5 allocations: 176 bytes)
8.414709847530957e6

julia> @time unstable(10_000_000) # master last week
  0.365068 seconds (30.00 M allocations: 457.764 MiB, 3.54% gc time)
```

0.6.0

Notifications

Unsubscribe

You're receiving notifications because you were mentioned.

11 participants

Lock conversation

johnmyleswhite commented 16 days ago — Member

So excited to see this being worked on. Am I right to think there are still performance optimizations that can be made? For now, I'm using the following snippet to check the performance of union types:

```
function stable(n)
    s = 0.0
    for i in 1:n
        s += sin(1.0)
    end
    return s
end

function unstable(n)
    s = 0
    for i in 1:n
        s += sin(1.0)
    end
    return s
end

@time stable(10_000_000)
@time unstable(10_000_000)
```

Currently this branch produces very similar behavior as Julia 0.5 did: the union type of `s` in `unstable` causes about a 2x latency degradation and allocates a lot of memory.

andyferris commented 16 days ago — Contributor

Very cool, Jameson. I'm guessing from the OP and John's comment that the remaining half of work is to make unions of two isbits types isbits themselves? Would these be laid out be like a variant/ `Switch` , or overlapped like a C union?

vtjnash commented 16 days ago — Member

That's a boring example. I can cut down the allocation count, but they don't really cost any time relative to `sin` . The performance variation seen here is entirely due to a processor bug / feature. If you `@profile` both, you'll see that >99% of time in all versions is spent in the `sin` kernel:

```
julia> @time stable(10_000_000) # comparison
  0.153794 seconds (5 allocations: 176 bytes)
8.414709847538957e6

julia> @time unstable(10_000_000) # altered version of PR
```

None yet

Milestone

0.6.0

Notifications

Unsubscribe

You're receiving notifications because you were mentioned.

11 participants

Lock conversation

```
function stable(n)
    s = 0.0
    for i in 1:n
        s += sin(1.0)
    end
    return s
end

function unstable(n)
    s = 0
    for i in 1:n
        s += sin(1.0)
    end
    return s
end

@time stable(10_000_000)
@time unstable(10_000_000)
```

Currently this branch produces very similar behavior as Julia 0.5 did: the union type of `s` in `unstable` causes about a 2x latency degradation and allocates a lot of memory.

andyferris commented 16 days ago — Contributor

Very cool, Jameson. I'm guessing from the OP and John's comment that the remaining half of work is to make unions of two isbits types isbits themselves? Would these be laid out be like a variant/ `Switch` , or overlapped like a C union?

vtjnash commented 16 days ago — Member

That's a boring example. I can cut down the allocation count, but they don't really cost any time relative to `sin` . The performance variation seen here is entirely due to a processor bug / feature. If you `@profile` both, you'll see that >99% of time in all versions is spent in the `sin` kernel:

```
julia> @time stable(10_000_000) # comparison
  0.153794 seconds (5 allocations: 176 bytes)
8.4147098475309576e6

julia> @time unstable(10_000_000) # altered version of PR
  0.318522 seconds (5 allocations: 176 bytes)
8.4147098475309576e6

julia> @time unstable(10_000_000) # master last week
  0.365068 seconds (30.00 M allocations: 457.764 MiB, 3.54% gc time)
8.4147098475309576e6
```

Notifications

Unsubscribe

You're receiving notifications because you were mentioned.

11 participants

Lock conversation

```
        s += sin(1.0)
    end
    return s
end

function unstable(n)
    s = 0
    for i in 1:n
        s += sin(1.0)
    end
    return s
end

@time stable(10_000_000)
@time unstable(10_000_000)
```

Currently this branch produces very similar behavior as Julia 0.5 did: the union type of `s` in `unstable` causes about a 2x latency degradation and allocates a lot of memory.

andyferris commented 16 days ago — Contributor

Very cool, Jameson. I'm guessing from the OP and John's comment that the remaining half of work is to make unions of two isbits types isbits themselves? Would these be laid out be like a variant/ `Switch` , or overlapped like a C union?

vtjnash commented 16 days ago — Member

That's a boring example. I can cut down the allocation count, but they don't really cost any time relative to `sin` . The performance variation seen here is entirely due to a processor bug / feature. If you `@profile` both, you'll see that >99% of time in all versions is spent in the `sin` kernel:

```
julia> @time stable(10_000_000) # comparison
  0.153794 seconds (5 allocations: 176 bytes)
8.414709847530957e6

julia> @time unstable(10_000_000) # altered version of PR
  0.318522 seconds (5 allocations: 176 bytes)
8.414709847530957e6

julia> @time unstable(10_000_000) # master last week
  0.365068 seconds (30.00 M allocations: 457.764 MiB, 3.54% gc time)
8.414709847530957e6

julia> @time unstable(10_000_000) # master today
  0.323274 seconds (10.00 M allocations: 152.588 MiB, 2.94% gc time)
8.414709847530957e6
```

```
@time stable(10_000_000)
@time unstable(10_000_000)
```

Currently this branch produces very similar behavior as Julia 0.5 did: the union type of `s` in `unstable` causes about a 2x latency degradation and allocates a lot of memory.

andyferris commented 16 days ago — Contributor

Very cool, Jameson. I'm guessing from the OP and John's comment that the remaining half of work is to make unions of two isbits types isbits themselves? Would these be laid out be like a variant/ `Switch` , or overlapped like a C union?

vtjnash commented 16 days ago — Member

That's a boring example. I can cut down the allocation count, but they don't really cost any time relative to `sin` . The performance variation seen here is entirely due to a processor bug / feature. If you `@profile` both, you'll see that >99% of time in all versions is spent in the `sin` kernel:

```
julia> @time stable(10_000_000) # comparison
  0.153794 seconds (5 allocations: 176 bytes)
8.414709847530957e6

julia> @time unstable(10_000_000) # altered version of PR
  0.318522 seconds (5 allocations: 176 bytes)
8.414709847530957e6

julia> @time unstable(10_000_000) # master last week
  0.365068 seconds (30.00 M allocations: 457.764 MiB, 3.54% gc time)
8.414709847530957e6

julia> @time unstable(10_000_000) # master today
  0.323274 seconds (10.00 M allocations: 152.588 MiB, 2.94% gc time)
8.414709847530957e6
```

A slightly more interesting version is:

```
julia> @noinline sumup(x::Int) = iseven(x) ? x / 2 : x + 3
sumup (generic function with 1 method)

julia> function f(c)
         x::Int = c
         while x != 1
           x = sumup(x)
         end
```

```
    s = 0
    for i in 1:n
        s += sin(1.0)
    end
    return s
end

@time stable(10_000_000)
@time unstable(10_000_000)
```

Currently this branch produces very similar behavior as Julia 0.5 did: the union type of `s` in `unstable` causes about a 2x latency degradation and allocates a lot of memory.

andyferris commented 16 days ago — Contributor

Very cool, Jameson. I'm guessing from the OP and John's comment that the remaining half of work is to make unions of two isbits types isbits themselves? Would these be laid out be like a variant/ `Switch` , or overlapped like a C union?

vtjnash commented 16 days ago — Member

That's a boring example. I can cut down the allocation count, but they don't really cost any time relative to `sin` . The performance variation seen here is entirely due to a processor bug / feature. If you `@profile` both, you'll see that >99% of time in all versions is spent in the `sin` kernel:

```
julia> @time stable(10_000_000) # comparison
  0.153794 seconds (5 allocations: 176 bytes)
8.414709847530957e6

julia> @time unstable(10_000_000) # altered version of PR
  0.318522 seconds (5 allocations: 176 bytes)
8.414709847530957e6

julia> @time unstable(10_000_000) # master last week
  0.365068 seconds (30.00 M allocations: 457.764 MiB, 3.54% gc time)
8.414709847530957e6

julia> @time unstable(10_000_000) # master today
  0.323274 seconds (10.00 M allocations: 152.588 MiB, 2.94% gc time)
8.414709847530957e6
```

A slightly more interesting version is:

```
julia> @noinline sumup(x::Int) = iseven(x) ? x / 2 : x + 3
```

Lock conversation

out be like a variant/ `Switch` , or overlapped like a C union?

vtjnash commented 16 days ago Member

That's a boring example. I can cut down the allocation count, but they don't really cost any time relative to `sin` . The performance variation seen here is entirely due to a processor bug / feature. If you `@profile` both, you'll see that >99% of time in all versions is spent in the `sin` kernel:

```
julia> @time stable(10_000_000) # comparison
  0.153794 seconds (5 allocations: 176 bytes)
8.414709847530957e6

julia> @time unstable(10_000_000) # altered version of PR
  0.318522 seconds (5 allocations: 176 bytes)
8.414709847530957e6

julia> @time unstable(10_000_000) # master last week
  0.365068 seconds (30.00 M allocations: 457.764 MiB, 3.54% gc time)
8.414709847530957e6

julia> @time unstable(10_000_000) # master today
  0.323274 seconds (10.00 M allocations: 152.588 MiB, 2.94% gc time)
8.414709847530957e6
```

A slightly more interesting version is:

```
julia> @noinline sumup(x::Int) = iseven(x) ? x / 2 : x + 3
sumup (generic function with 1 method)

julia> function f(c)
         x::Int = c
         while x != 1
           x = sumup(x)
         end
       end
f (generic function with 1 method)

julia> @time for i = 1:10^7; f(166138751); end # altered version of PR
  3.253064 seconds

julia> @time for i = 1:10^7; f(166138751); end # master
 11.707397 seconds (650.00 M allocations: 9.686 GiB, 2.39% gc time)
```

Although even here it is very difficult to craft a meaningful benchmark since most of the cost is in using a floating point representation, and the cost of allocation is relatively just noise:

That's a boring example. I can cut down the allocation count, but they don't really cost any time relative to `sin`. The performance variation seen here is entirely due to a processor bug / feature. If you `@profile` both, you'll see that >99% of time in all versions is spent in the `sin` kernel:

```
julia> @time stable(10_000_000) # comparison
  0.153794 seconds (5 allocations: 176 bytes)
8.4147098475309576e6

julia> @time unstable(10_000_000) # altered version of PR
  0.318522 seconds (5 allocations: 176 bytes)
8.4147098475309576e6

julia> @time unstable(10_000_000) # master last week
  0.365068 seconds (30.00 M allocations: 457.764 MiB, 3.54% gc time)
8.4147098475309576e6

julia> @time unstable(10_000_000) # master today
  0.323274 seconds (10.00 M allocations: 152.588 MiB, 2.94% gc time)
8.4147098475309576e6
```

A slightly more interesting version is:

```
julia> @noinline sumup(x::Int) = iseven(x) ? x / 2 : x + 3
sumup (generic function with 1 method)

julia> function f(c)
         x::Int = c
         while x != 1
           x = sumup(x)
         end
       end
f (generic function with 1 method)

julia> @time for i = 1:10^7; f(166138751); end # altered version of PR
  3.253064 seconds

julia> @time for i = 1:10^7; f(166138751); end # master
 11.707397 seconds (650.00 M allocations: 9.686 GiB, 2.39% gc time)
```

Although even here it is very difficult to craft a meaningful benchmark since most of the cost is in using a floating point representation, and the cost of allocation is relatively just noise:

```
julia> @noinline sumup(x) = (xi = unsafe_trunc(Int, x); iseven(xi) ? div(x, 2) : x + 3)
sumup (generic function with 1 method)

julia> function f(c)
```

```
end
v1 = Vector{Nullable{Int}}(n)
v2 = Vector{Nullable{Int}}(n)
v3 = Vector{Nullable{Int}}(n)
for i = 1:n
    if rand() < 0.6666
        v1[i] = Nullable(rand(1:100))
    else
        v1[i] = Nullable{Int}()
    end
    if rand() < 0.6666
        v2[i] = Nullable(rand(1:100))
    else
        v2[i] = Nullable{Int}()
    end
end
```

yields

```
julia> @time map!(+, v3, v1, v2);
  0.132412 seconds (4 allocations: 160 bytes)
```

which is about 6 times slower than the `Int` case and about 3 times faster than this PR currently (I didn't attempt to optimize that kernel for `+`).

andreasnoack commented 15 days ago — Member

> Maybe DataArrays would be a good example of the kind of things this PR will solve, but it seems to be broken on v0.6.

You can try this PR JuliaStats/DataArrays.jl#235

vtjnash commented 15 days ago — Member

@vchuravy I already looked at it - the assertion is wrong. I just need to push a fix to avoid calling it there.

@andyferris This PR is only for the implementation of the codegen support. It doesn't implement support for representing unions efficiently in vectors.

andyferris commented 15 days ago — Contributor

> This PR is only for the implementation of the codegen support. It doesn't implement support for representing unions efficiently in vectors.

@andyferris This PR is only for the implementation of the codegen support. It doesn't implement support for representing unions efficiently in vectors.

andyferris commented 15 days ago — Contributor

> This PR is only for the implementation of the codegen support. It doesn't implement support for representing unions efficiently in vectors.

Right – I'm getting ahead of you (because this is exciting 😊)

ararslan added this to the 0.6.0 milestone 13 days ago

ararslan commented 13 days ago — Member

@nanosoldier `runbenchmarks(ALL, vs = ":master")`

tkelman added the needs docs label 13 days ago

nanosoldier commented 13 days ago — Member

Your benchmark job has completed - possible performance regressions were detected. A full report can be found here. cc @jrevels

vtjnash commented 13 days ago — Member

Wow cool, I've never created a 14552.04% slowdown before. I didn't even have to allocate anything!

😊 6

tkelman commented 12 days ago — Member

@nanosoldier `runbenchmarks(ALL, vs = ":master")`

nanosoldier commented 12 days ago — Member

andyferris commented 15 days ago — Contributor

> This PR is only for the implementation of the codegen support. It doesn't implement support for representing unions efficiently in vectors.

Right - I'm getting ahead of you (because this is exciting 😊)

ararslan added this to the 0.6.0 milestone 13 days ago

ararslan commented 13 days ago — Member

@nanosoldier `runbenchmarks(ALL, vs = ":master")`

tkelman added the needs docs label 13 days ago

nanosoldier commented 13 days ago — Member

Your benchmark job has completed - possible performance regressions were detected. A full report can be found here. cc @jrevels

vtjnash commented 13 days ago — Member

Wow cool, I've never created a 14552.04% slowdown before. I didn't even have to allocate anything!

😊 6

tkelman commented 12 days ago — Member

@nanosoldier `runbenchmarks(ALL, vs = ":master")`

nanosoldier commented 12 days ago — Member

Your benchmark job has completed - possible performance regressions were detected. A full report can be found here. cc @jrevels

🎉 1

ararslan added this to the 0.6.0 milestone 13 days ago

ararslan commented 13 days ago — Member

@nanosoldier `runbenchmarks(ALL, vs = ":master")`

tkelman added the needs docs label 13 days ago

nanosoldier commented 13 days ago — Member

Your benchmark job has completed - possible performance regressions were detected. A full report can be found here. cc @jrevels

vtjnash commented 13 days ago — Member

Wow cool, I've never created a 14552.04% slowdown before. I didn't even have to allocate anything!

6

tkelman commented 12 days ago — Member

@nanosoldier `runbenchmarks(ALL, vs = ":master")`

nanosoldier commented 12 days ago — Member

Your benchmark job has completed - possible performance regressions were detected. A full report can be found here. cc @jrevels

1

vtjnash reviewed 9 days ago — View changes

src/codegen.cpp — Show outdated

Benchmark Report

Job Properties

Commit(s): JuliaLang/julia@c9132d2af2cb28888e8e74c3d8406978b8be461e vs JuliaLang/julia@7ab6428050466fa20cde4e06869d9a926919ff2c

Triggered By: link

Tag Predicate: `ALL`

Results

Note: If Chrome is your browser, I strongly recommend installing the Wide GitHub extension, which makes the result table easier to read.

Below is a table of this job's results, obtained by running the benchmarks found in JuliaCI/BaseBenchmarks.jl. The values listed in the `ID` column have the structure `[parent_group, child_group, ..., key]`, and can be used to index into the BaseBenchmarks suite to retrieve the corresponding benchmarks.

The percentages accompanying time and memory values in the below table are noise tolerances. The "true" time/memory value for a given benchmark is expected to fall within this percentage of the reported value.

Benchmark Report

Job Properties

Commit(s): JuliaLang/julia@c9132d2af2cb28888e8e74c3d8406978b8be461e vs JuliaLang/julia@7ab6428050466fa20cde4e06869d9a926919ff2c

Triggered By: link

Tag Predicate: `ALL`

Results

Note: If Chrome is your browser, I strongly recommend installing the Wide GitHub extension, which makes the result table easier to read.

Below is a table of this job's results, obtained by running the benchmarks found in JuliaCI/BaseBenchmarks.jl. The values listed in the `ID` column have the structure `[parent_group, child_group, ..., key]`, and can be used to index into the BaseBenchmarks suite to retrieve the corresponding benchmarks.

The percentages accompanying time and memory values in the below table are noise tolerances. The "true" time/memory value for a given benchmark is expected to fall within this percentage of the reported value.

A ratio greater than `1.0` denotes a possible regression (marked with ❌), while a ratio less than `1.0` denotes a possible improvement (marked with ✅). Only significant results - results that indicate possible regressions or improvements - are shown below (thus, an empty table means that all benchmark results remained invariant between builds).

ID	time ratio	memory ratio
`["array","comprehension",("collect","Array{Float64,1}")]`	0.36 (15%) ✅	1.00 (1%)
`["array","comprehension",("collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]`	0.66 (15%) ✅	1.00 (1%)

Commit(s): JuliaLang/julia@c9132d2af2cb28888e8e74c3d8406978b8be461e vs JuliaLang/julia@7ab6428050466fa20cde4e06869d9a926919ff2c

Triggered By: link

Tag Predicate: `ALL`

Results

Note: If Chrome is your browser, I strongly recommend installing the Wide GitHub extension, which makes the result table easier to read.

Below is a table of this job's results, obtained by running the benchmarks found in JuliaCI/BaseBenchmarks.jl. The values listed in the `ID` column have the structure `[parent_group, child_group, ..., key]`, and can be used to index into the BaseBenchmarks suite to retrieve the corresponding benchmarks.

The percentages accompanying time and memory values in the below table are noise tolerances. The "true" time/memory value for a given benchmark is expected to fall within this percentage of the reported value.

A ratio greater than `1.0` denotes a possible regression (marked with ❌), while a ratio less than `1.0` denotes a possible improvement (marked with ✅). Only significant results - results that indicate possible regressions or improvements - are shown below (thus, an empty table means that all benchmark results remained invariant between builds).

ID	time ratio	memory ratio
`["array","comprehension",("collect","Array{Float64,1}")]`	0.36 (15%) ✅	1.00 (1%)
`["array","comprehension",("collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]`	0.66 (15%) ✅	1.00 (1%)
`["array","comprehension",("comprehension_collect","Array{Float64,1}")]`	0.54 (15%) ✅	1.00 (1%)
`["array","comprehension", ("comprehension_collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]`	0.66 (15%) ✅	1.00 (1%)
`["array","comprehension",("comprehension_iteration","Array{Float64,1}")]`	0.84 (15%) ✅	1.00 (1%)
`["array","convert",("Int","Float64")]`	1.65 (15%) ❌	1.00 (1%)
`["array","index",("sumelt","Array{Int32,2}")]`	4.34 (50%) ❌	1.00 (1%)
`["array","index",("sumelt_boundscheck","Array{Int32,2}")]`	4.34 (50%) ❌	1.00 (1%)
`["broadcast","sparse",((1000,1000),2)]`	0.83 (15%) ✅	1.00 (1%)
`["dates","parse","DateTime"]`	1.26 (15%) ❌	1.00 (1%)

Triggered By: link

Tag Predicate: `ALL`

Results

Note: If Chrome is your browser, I strongly recommend installing the Wide GitHub extension, which makes the result table easier to read.

Below is a table of this job's results, obtained by running the benchmarks found in JuliaCI/BaseBenchmarks.jl. The values listed in the `ID` column have the structure `[parent_group, child_group, ..., key]`, and can be used to index into the BaseBenchmarks suite to retrieve the corresponding benchmarks.

The percentages accompanying time and memory values in the below table are noise tolerances. The "true" time/memory value for a given benchmark is expected to fall within this percentage of the reported value.

A ratio greater than `1.0` denotes a possible regression (marked with ❌), while a ratio less than `1.0` denotes a possible improvement (marked with ✅). Only significant results - results that indicate possible regressions or improvements - are shown below (thus, an empty table means that all benchmark results remained invariant between builds).

ID	time ratio	memory ratio
`["array","comprehension",("collect","Array{Float64,1}")]`	0.36 (15%) ✅	1.00 (1%)
`["array","comprehension",("collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]`	0.66 (15%) ✅	1.00 (1%)
`["array","comprehension",("comprehension_collect","Array{Float64,1}")]`	0.54 (15%) ✅	1.00 (1%)
`["array","comprehension", ("comprehension_collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]`	0.66 (15%) ✅	1.00 (1%)
`["array","comprehension",("comprehension_iteration","Array{Float64,1}")]`	0.84 (15%) ✅	1.00 (1%)
`["array","convert",("Int","Float64")]`	1.65 (15%) ❌	1.00 (1%)
`["array","index",("sumelt","Array{Int32,2}")]`	4.34 (50%) ❌	1.00 (1%)
`["array","index",("sumelt_boundscheck","Array{Int32,2}")]`	4.34 (50%) ❌	1.00 (1%)
`["broadcast","sparse",((1000,1000),2)]`	0.83 (15%) ✅	1.00 (1%)
`["dates","parse","DateTime"]`	1.26 (15%) ❌	1.00 (1%)
`["dates","parse",("DateTime","ISODateTimeFormat")]`	1.23 (15%) ❌	1.00 (1%)

Triggered By: link

Tag Predicate: `ALL`

Results

Note: If Chrome is your browser, I strongly recommend installing the Wide GitHub extension, which makes the result table easier to read.

Below is a table of this job's results, obtained by running the benchmarks found in JuliaCI/BaseBenchmarks.jl. The values listed in the `ID` column have the structure `[parent_group, child_group, ..., key]`, and can be used to index into the BaseBenchmarks suite to retrieve the corresponding benchmarks.

The percentages accompanying time and memory values in the below table are noise tolerances. The "true" time/memory value for a given benchmark is expected to fall within this percentage of the reported value.

A ratio greater than `1.0` denotes a possible regression (marked with ❌), while a ratio less than `1.0` denotes a possible improvement (marked with ✅). Only significant results - results that indicate possible regressions or improvements - are shown below (thus, an empty table means that all benchmark results remained invariant between builds).

ID	time ratio	memory ratio
`["array","comprehension",("collect","Array{Float64,1}")]`	0.36 (15%) ✅	1.00 (1%)
`["array","comprehension",("collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]`	0.66 (15%) ✅	1.00 (1%)
`["array","comprehension",("comprehension_collect","Array{Float64,1}")]`	0.54 (15%) ✅	1.00 (1%)
`["array","comprehension", ("comprehension_collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]`	0.66 (15%) ✅	1.00 (1%)
`["array","comprehension",("comprehension_iteration","Array{Float64,1}")]`	0.84 (15%) ✅	1.00 (1%)
`["array","convert",("Int","Float64")]`	1.65 (15%) ❌	1.00 (1%)
`["array","index",("sumelt","Array{Int32,2}")]`	4.34 (50%) ❌	1.00 (1%)
`["array","index",("sumelt_boundscheck","Array{Int32,2}")]`	4.34 (50%) ❌	1.00 (1%)
`["broadcast","sparse",((1000,1000),2)]`	0.83 (15%) ✅	1.00 (1%)
`["dates","parse","DateTime"]`	1.26 (15%) ❌	1.00 (1%)
`["dates","parse",("DateTime","ISODateTimeFormat")]`	1.23 (15%) ❌	1.00 (1%)

Tag Predicate: `ALL`

Results

Note: If Chrome is your browser, I strongly recommend installing the Wide GitHub extension, which makes the result table easier to read.

Below is a table of this job's results, obtained by running the benchmarks found in JuliaCI/BaseBenchmarks.jl. The values listed in the `ID` column have the structure `[parent_group, child_group, ..., key]`, and can be used to index into the BaseBenchmarks suite to retrieve the corresponding benchmarks.

The percentages accompanying time and memory values in the below table are noise tolerances. The "true" time/memory value for a given benchmark is expected to fall within this percentage of the reported value.

A ratio greater than `1.0` denotes a possible regression (marked with ❌), while a ratio less than `1.0` denotes a possible improvement (marked with ✅). Only significant results - results that indicate possible regressions or improvements - are shown below (thus, an empty table means that all benchmark results remained invariant between builds).

ID	time ratio	memory ratio
`["array","comprehension",("collect","Array{Float64,1}")]`	0.36 (15%) ✅	1.00 (1%)
`["array","comprehension",("collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]`	0.66 (15%) ✅	1.00 (1%)
`["array","comprehension",("comprehension_collect","Array{Float64,1}")]`	0.54 (15%) ✅	1.00 (1%)
`["array","comprehension", ("comprehension_collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]`	0.66 (15%) ✅	1.00 (1%)
`["array","comprehension",("comprehension_iteration","Array{Float64,1}")]`	0.84 (15%) ✅	1.00 (1%)
`["array","convert",("Int","Float64")]`	1.65 (15%) ❌	1.00 (1%)
`["array","index",("sumelt","Array{Int32,2}")]`	4.34 (50%) ❌	1.00 (1%)
`["array","index",("sumelt_boundscheck","Array{Int32,2}")]`	4.34 (50%) ❌	1.00 (1%)
`["broadcast","sparse",((1000,1000),2)]`	0.83 (15%) ✅	1.00 (1%)
`["dates","parse","DateTime"]`	1.26 (15%) ❌	1.00 (1%)
`["dates","parse",("DateTime","ISODateTimeFormat")]`	1.23 (15%) ❌	1.00 (1%)
`["linalg","arithmetic",("sqrtm","Base.LinAlg.UnitUpperTriangular",1024)]`	2.41 (45%) ❌	1.00 (1%)

Results

Note: If Chrome is your browser, I strongly recommend installing the Wide GitHub extension, which makes the result table easier to read.

Below is a table of this job's results, obtained by running the benchmarks found in JuliaCI/BaseBenchmarks.jl. The values listed in the `ID` column have the structure `[parent_group, child_group, ..., key]`, and can be used to index into the BaseBenchmarks suite to retrieve the corresponding benchmarks.

The percentages accompanying time and memory values in the below table are noise tolerances. The "true" time/memory value for a given benchmark is expected to fall within this percentage of the reported value.

A ratio greater than `1.0` denotes a possible regression (marked with ❌), while a ratio less than `1.0` denotes a possible improvement (marked with ✅). Only significant results - results that indicate possible regressions or improvements - are shown below (thus, an empty table means that all benchmark results remained invariant between builds).

ID	time ratio	memory ratio
["array","comprehension",("collect","Array{Float64,1}")]	0.36 (15%) ✅	1.00 (1%)
["array","comprehension",("collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]	0.66 (15%) ✅	1.00 (1%)
["array","comprehension",("comprehension_collect","Array{Float64,1}")]	0.54 (15%) ✅	1.00 (1%)
["array","comprehension", ("comprehension_collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]	0.66 (15%) ✅	1.00 (1%)
["array","comprehension",("comprehension_iteration","Array{Float64,1}")]	0.84 (15%) ✅	1.00 (1%)
["array","convert",("Int","Float64")]	1.65 (15%) ❌	1.00 (1%)
["array","index",("sumelt","Array{Int32,2}")]	4.34 (50%) ❌	1.00 (1%)
["array","index",("sumelt_boundscheck","Array{Int32,2}")]	4.34 (50%) ❌	1.00 (1%)
["broadcast","sparse",((1000,1000),2)]	0.83 (15%) ✅	1.00 (1%)
["dates","parse","DateTime"]	1.26 (15%) ❌	1.00 (1%)
["dates","parse",("DateTime","ISODateTimeFormat")]	1.23 (15%) ❌	1.00 (1%)
["linalg","arithmetic",("sqrtm","Base.LinAlg.UnitUpperTriangular",1024)]	2.41 (45%) ❌	1.00 (1%)
["linalg","arithmetic",("sqrtm","UpperTriangular",1024)]	2.40 (45%) ❌	1.00 (1%)
["micro","pisum"]	0.52 (15%) ✅	1.00 (1%)

Note: If Chrome is your browser, I strongly recommend installing the Wide GitHub extension, which makes the result table easier to read.

Below is a table of this job's results, obtained by running the benchmarks found in JuliaCI/BaseBenchmarks.jl. The values listed in the `ID` column have the structure `[parent_group, child_group, ..., key]`, and can be used to index into the BaseBenchmarks suite to retrieve the corresponding benchmarks.

The percentages accompanying time and memory values in the below table are noise tolerances. The "true" time/memory value for a given benchmark is expected to fall within this percentage of the reported value.

A ratio greater than `1.0` denotes a possible regression (marked with ❌), while a ratio less than `1.0` denotes a possible improvement (marked with ✅). Only significant results - results that indicate possible regressions or improvements - are shown below (thus, an empty table means that all benchmark results remained invariant between builds).

ID	time ratio	memory ratio
`["array","comprehension",("collect","Array{Float64,1}")]`	0.36 (15%) ✅	1.00 (1%)
`["array","comprehension",("collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]`	0.66 (15%) ✅	1.00 (1%)
`["array","comprehension",("comprehension_collect","Array{Float64,1}")]`	0.54 (15%) ✅	1.00 (1%)
`["array","comprehension", ("comprehension_collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]`	0.66 (15%) ✅	1.00 (1%)
`["array","comprehension",("comprehension_iteration","Array{Float64,1}")]`	0.84 (15%) ✅	1.00 (1%)
`["array","convert",("Int","Float64")]`	1.65 (15%) ❌	1.00 (1%)
`["array","index",("sumelt","Array{Int32,2}")]`	4.34 (50%) ❌	1.00 (1%)
`["array","index",("sumelt_boundscheck","Array{Int32,2}")]`	4.34 (50%) ❌	1.00 (1%)
`["broadcast","sparse",((1000,1000),2)]`	0.83 (15%) ✅	1.00 (1%)
`["dates","parse","DateTime"]`	1.26 (15%) ❌	1.00 (1%)
`["dates","parse",("DateTime","ISODateTimeFormat")]`	1.23 (15%) ❌	1.00 (1%)
`["linalg","arithmetic",("sqrtm","Base.LinAlg.UnitUpperTriangular",1024)]`	2.41 (45%) ❌	1.00 (1%)
`["linalg","arithmetic",("sqrtm","UpperTriangular",1024)]`	2.40 (45%) ❌	1.00 (1%)
`["micro","pisum"]`	0.52 (15%) ✅	1.00 (1%)
`["misc","parse","DateTime"]`	1.22 (15%) ❌	1.00 (1%)

A ratio greater than `1.0` denotes a possible regression (marked with ❌), while a ratio less than `1.0` denotes a possible improvement (marked with ✅). Only significant results – results that indicate possible regressions or improvements – are shown below (thus, an empty table means that all benchmark results remained invariant between builds).

ID	time ratio	memory ratio
`["array","comprehension",("collect","Array{Float64,1}")]`	0.36 (15%) ✅	1.00 (1%)
`["array","comprehension",("collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]`	0.66 (15%) ✅	1.00 (1%)
`["array","comprehension",("comprehension_collect","Array{Float64,1}")]`	0.54 (15%) ✅	1.00 (1%)
`["array","comprehension",("comprehension_collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]`	0.66 (15%) ✅	1.00 (1%)
`["array","comprehension",("comprehension_iteration","Array{Float64,1}")]`	0.84 (15%) ✅	1.00 (1%)
`["array","convert",("Int","Float64")]`	1.65 (15%) ❌	1.00 (1%)
`["array","index",("sumelt","Array{Int32,2}")]`	4.34 (50%) ❌	1.00 (1%)
`["array","index",("sumelt_boundscheck","Array{Int32,2}")]`	4.34 (50%) ❌	1.00 (1%)
`["broadcast","sparse",((1000,1000),2)]`	0.83 (15%) ✅	1.00 (1%)
`["dates","parse","DateTime"]`	1.26 (15%) ❌	1.00 (1%)
`["dates","parse",("DateTime","ISODateTimeFormat")]`	1.23 (15%) ❌	1.00 (1%)
`["linalg","arithmetic",("sqrtm","Base.LinAlg.UnitUpperTriangular",1024)]`	2.41 (45%) ❌	1.00 (1%)
`["linalg","arithmetic",("sqrtm","UpperTriangular",1024)]`	2.40 (45%) ❌	1.00 (1%)
`["micro","pisum"]`	0.52 (15%) ✅	1.00 (1%)
`["misc","parse","DateTime"]`	1.22 (15%) ❌	1.00 (1%)
`["nullable","basic",("get2","Nullable{BigFloat}(1.000000000000000000000000000000000000000000000000000000000000000000000000000000)")]`	1.89 (60%) ❌	1.00 (1%)
`["nullable","basic",("get2","Nullable{BigInt}(1)")]`	1.90 (60%) ❌	1.00 (1%)
`["parallel","remotecall",("identity",1024)]`	0.97 (15%)	0.99 (1%) ✅
`["parallel","remotecall",("identity",2)]`	0.96 (15%)	0.99 (1%) ✅

A ratio greater than `1.0` denotes a possible regression (marked with ❌), while a ratio less than `1.0` denotes a possible improvement (marked with ✅). Only significant results – results that indicate possible regressions or improvements – are shown below (thus, an empty table means that all benchmark results remained invariant between builds).

ID	time ratio	memory ratio
["array","comprehension",("collect","Array{Float64,1}")]	0.36 (15%) ✅	1.00 (1%)
["array","comprehension",("collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]	0.66 (15%) ✅	1.00 (1%)
["array","comprehension",("comprehension_collect","Array{Float64,1}")]	0.54 (15%) ✅	1.00 (1%)
["array","comprehension",("comprehension_collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]	0.66 (15%) ✅	1.00 (1%)
["array","comprehension",("comprehension_iteration","Array{Float64,1}")]	0.84 (15%) ✅	1.00 (1%)
["array","convert",("Int","Float64")]	1.65 (15%) ❌	1.00 (1%)
["array","index",("sumelt","Array{Int32,2}")]	4.34 (50%) ❌	1.00 (1%)
["array","index",("sumelt_boundscheck","Array{Int32,2}")]	4.34 (50%) ❌	1.00 (1%)
["broadcast","sparse",((1000,1000),2)]	0.83 (15%) ✅	1.00 (1%)
["dates","parse","DateTime"]	1.26 (15%) ❌	1.00 (1%)
["dates","parse",("DateTime","ISODateTimeFormat")]	1.23 (15%) ❌	1.00 (1%)
["linalg","arithmetic",("sqrtm","Base.LinAlg.UnitUpperTriangular",1024)]	2.41 (45%) ❌	1.00 (1%)
["linalg","arithmetic",("sqrtm","UpperTriangular",1024)]	2.40 (45%) ❌	1.00 (1%)
["micro","pisum"]	0.52 (15%) ✅	1.00 (1%)
["misc","parse","DateTime"]	1.22 (15%) ❌	1.00 (1%)
["nullable","basic",("get2","Nullable{BigFloat}(1.000000000000000000000000000000000000000000000000000000000000000000000000000000)")]	1.89 (60%) ❌	1.00 (1%)
["nullable","basic",("get2","Nullable{BigInt}(1)")]	1.90 (60%) ❌	1.00 (1%)
["parallel","remotecall",("identity",1024)]	0.97 (15%)	0.99 (1%) ✅
["parallel","remotecall",("identity",2)]	0.96 (15%)	0.99 (1%) ✅

A ratio greater than `1.0` denotes a possible regression (marked with ❌), while a ratio less than `1.0` denotes a possible improvement (marked with ✅). Only significant results – results that indicate possible regressions or improvements – are shown below (thus, an empty table means that all benchmark results remained invariant between builds).

ID	time ratio	memory ratio
`["array","comprehension",("collect","Array{Float64,1}")]`	0.36 (15%) ✅	1.00 (1%)
`["array","comprehension",("collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]`	0.66 (15%) ✅	1.00 (1%)
`["array","comprehension",("comprehension_collect","Array{Float64,1}")]`	0.54 (15%) ✅	1.00 (1%)
`["array","comprehension",("comprehension_collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]`	0.66 (15%) ✅	1.00 (1%)
`["array","comprehension",("comprehension_iteration","Array{Float64,1}")]`	0.84 (15%) ✅	1.00 (1%)
`["array","convert",("Int","Float64")]`	1.65 (15%) ❌	1.00 (1%)
`["array","index",("sumelt","Array{Int32,2}")]`	4.34 (50%) ❌	1.00 (1%)
`["array","index",("sumelt_boundscheck","Array{Int32,2}")]`	4.34 (50%) ❌	1.00 (1%)
`["broadcast","sparse",((1000,1000),2)]`	0.83 (15%) ✅	1.00 (1%)
`["dates","parse","DateTime"]`	1.26 (15%) ❌	1.00 (1%)
`["dates","parse",("DateTime","ISODateTimeFormat")]`	1.23 (15%) ❌	1.00 (1%)
`["linalg","arithmetic",("sqrtm","Base.LinAlg.UnitUpperTriangular",1024)]`	2.41 (45%) ❌	1.00 (1%)
`["linalg","arithmetic",("sqrtm","UpperTriangular",1024)]`	2.40 (45%) ❌	1.00 (1%)
`["micro","pisum"]`	0.52 (15%) ✅	1.00 (1%)
`["misc","parse","DateTime"]`	1.22 (15%) ❌	1.00 (1%)
`["nullable","basic",("get2","Nullable{BigFloat}(1.000000000000000000000000000000000000000000000000000000000000000000000000000000000)")]`	1.89 (60%) ❌	1.00 (1%)
`["nullable","basic",("get2","Nullable{BigInt}(1)")]`	1.90 (60%) ❌	1.00 (1%)
`["parallel","remotecall",("identity",1024)]`	0.97 (15%)	0.99 (1%) ✅
`["parallel","remotecall",("identity",2)]`	0.96 (15%)	0.99 (1%) ✅

A ratio greater than 1.0 denotes a possible regression (marked with ❌), while a ratio less than 1.0 denotes a possible improvement (marked with ✅). Only significant results - results that indicate possible regressions or improvements - are shown below (thus, an empty table means that all benchmark results remained invariant between builds).

ID	time ratio	memory ratio
["array","comprehension",("collect","Array{Float64,1}")]	0.36 (15%) ✅	1.00 (1%)
["array","comprehension",("collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]	0.66 (15%) ✅	1.00 (1%)
["array","comprehension",("comprehension_collect","Array{Float64,1}")]	0.54 (15%) ✅	1.00 (1%)
["array","comprehension",("comprehension_collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]	0.66 (15%) ✅	1.00 (1%)
["array","comprehension",("comprehension_iteration","Array{Float64,1}")]	0.84 (15%) ✅	1.00 (1%)
["array","convert",("Int","Float64")]	1.65 (15%) ❌	1.00 (1%)
["array","index",("sumelt","Array{Int32,2}")]	4.34 (50%) ❌	1.00 (1%)
["array","index",("sumelt_boundscheck","Array{Int32,2}")]	4.34 (50%) ❌	1.00 (1%)
["broadcast","sparse",((1000,1000),2)]	0.83 (15%) ✅	1.00 (1%)
["dates","parse","DateTime"]	1.26 (15%) ❌	1.00 (1%)
["dates","parse",("DateTime","ISODateTimeFormat")]	1.23 (15%) ❌	1.00 (1%)
["linalg","arithmetic",("sqrtm","Base.LinAlg.UnitUpperTriangular",1024)]	2.41 (45%) ❌	1.00 (1%)
["linalg","arithmetic",("sqrtm","UpperTriangular",1024)]	2.40 (45%) ❌	1.00 (1%)
["micro","pisum"]	0.52 (15%) ✅	1.00 (1%)
["misc","parse","DateTime"]	1.22 (15%) ❌	1.00 (1%)
["nullable","basic",("get2","Nullable{BigFloat}(1.000000000000000000000000000000000000000000000000000000000000000000000000000)")]	1.89 (60%) ❌	1.00 (1%)
["nullable","basic",("get2","Nullable{BigInt}(1)")]	1.90 (60%) ❌	1.00 (1%)
["parallel","remotecall",("identity",1024)]	0.97 (15%)	0.99 (1%) ✅
["parallel","remotecall",("identity",2)]	0.96 (15%)	0.99 (1%) ✅
["parallel","remotecall",("identity",512)]	0.96 (15%)	0.99 (1%) ✅

["array","comprehension",("collect","Array{Float64,1}")]	0.36 (15%) ✅	1.00 (1%)
["array","comprehension",("collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]	0.66 (15%) ✅	1.00 (1%)
["array","comprehension",("comprehension_collect","Array{Float64,1}")]	0.54 (15%) ✅	1.00 (1%)
["array","comprehension",("comprehension_collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]	0.66 (15%) ✅	1.00 (1%)
["array","comprehension",("comprehension_iteration","Array{Float64,1}")]	0.84 (15%) ✅	1.00 (1%)
["array","convert",("Int","Float64")]	1.65 (15%) ❌	1.00 (1%)
["array","index",("sumelt","Array{Int32,2}")]	4.34 (50%) ❌	1.00 (1%)
["array","index",("sumelt_boundscheck","Array{Int32,2}")]	4.34 (50%) ❌	1.00 (1%)
["broadcast","sparse",((1000,1000),2)]	0.83 (15%) ✅	1.00 (1%)
["dates","parse","DateTime"]	1.26 (15%) ❌	1.00 (1%)
["dates","parse",("DateTime","ISODateTimeFormat")]	1.23 (15%) ❌	1.00 (1%)
["linalg","arithmetic",("sqrtm","Base.LinAlg.UnitUpperTriangular",1024)]	2.41 (45%) ❌	1.00 (1%)
["linalg","arithmetic",("sqrtm","UpperTriangular",1024)]	2.40 (45%) ❌	1.00 (1%)
["micro","pisum"]	0.52 (15%) ✅	1.00 (1%)
["misc","parse","DateTime"]	1.22 (15%) ❌	1.00 (1%)
["nullable","basic",("get2","Nullable{BigFloat}(1.000000000000000000000000000000000000000000000000000000000000000000000000000000)")]	1.89 (60%) ❌	1.00 (1%)
["nullable","basic",("get2","Nullable{BigInt}(1)")]	1.90 (60%) ❌	1.00 (1%)
["parallel","remotecall",("identity",1024)]	0.97 (15%)	0.99 (1%) ✅
["parallel","remotecall",("identity",2)]	0.96 (15%)	0.99 (1%) ✅
["parallel","remotecall",("identity",512)]	0.96 (15%)	0.99 (1%) ✅
["parallel","remotecall",("identity",64)]	0.97 (15%)	0.99 (1%) ✅
["problem","raytrace","raytrace"]	1.13 (15%)	1.02 (1%) ❌
["scalar","iteration","in"]	14553.04 (25%)	1.00 (1%)

("comprehension_collect","StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}")]	0.66 (15%) ✅	1.00 (1%)
["array","comprehension",("comprehension_iteration","Array{Float64,1}")]	0.84 (15%) ✅	1.00 (1%)
["array","convert",("Int","Float64")]	1.65 (15%) ❌	1.00 (1%)
["array","index",("sumelt","Array{Int32,2}")]	4.34 (50%) ❌	1.00 (1%)
["array","index",("sumelt_boundscheck","Array{Int32,2}")]	4.34 (50%) ❌	1.00 (1%)
["broadcast","sparse",((1000,1000),2)]	0.83 (15%) ✅	1.00 (1%)
["dates","parse","DateTime"]	1.26 (15%) ❌	1.00 (1%)
["dates","parse",("DateTime","ISODateTimeFormat")]	1.23 (15%) ❌	1.00 (1%)
["linalg","arithmetic",("sqrtm","Base.LinAlg.UnitUpperTriangular",1024)]	2.41 (45%) ❌	1.00 (1%)
["linalg","arithmetic",("sqrtm","UpperTriangular",1024)]	2.40 (45%) ❌	1.00 (1%)
["micro","pisum"]	0.52 (15%) ✅	1.00 (1%)
["misc","parse","DateTime"]	1.22 (15%) ❌	1.00 (1%)
["nullable","basic",("get2","Nullable{BigFloat}(1.000000000000000000000000000000000000000000000000000000000000000000000000000000)")]	1.89 (60%) ❌	1.00 (1%)
["nullable","basic",("get2","Nullable{BigInt}(1)")]	1.90 (60%) ❌	1.00 (1%)
["parallel","remotecall",("identity",1024)]	0.97 (15%)	0.99 (1%) ✅
["parallel","remotecall",("identity",2)]	0.96 (15%)	0.99 (1%) ✅
["parallel","remotecall",("identity",512)]	0.96 (15%)	0.99 (1%) ✅
["parallel","remotecall",("identity",64)]	0.97 (15%)	0.99 (1%) ✅
["problem","raytrace","raytrace"]	1.13 (15%)	1.02 (1%) ❌
["scalar","iteration","in"]	14553.04 (25%) ❌	1.00 (1%)
["shootout","fasta"]	0.78 (15%) ✅	1.00 (1%)
["simd",("inner","Float32",4095)]	1.27 (20%) ❌	1.00 (1%)
["simd",("inner","Float32",4096)]	1.38 (20%) ❌	1.00 (1%)

["dates","parse",("DateTime","ISODateTimeFormat")]	1.23 (15%) ❌	1.00 (1%)
["linalg","arithmetic",("sqrtm","Base.LinAlg.UnitUpperTriangular",1024)]	2.41 (45%) ❌	1.00 (1%)
["linalg","arithmetic",("sqrtm","UpperTriangular",1024)]	2.40 (45%) ❌	1.00 (1%)
["micro","pisum"]	0.52 (15%) ✅	1.00 (1%)
["misc","parse","DateTime"]	1.22 (15%) ❌	1.00 (1%)
["nullable","basic",("get2","Nullable{BigFloat}(1.000000000000000000000000000000000000000000000000000000000000000000000000000000000)")]	1.89 (60%) ❌	1.00 (1%)
["nullable","basic",("get2","Nullable{BigInt}(1)")]	1.90 (60%) ❌	1.00 (1%)
["parallel","remotecall",("identity",1024)]	0.97 (15%)	0.99 (1%) ✅
["parallel","remotecall",("identity",2)]	0.96 (15%)	0.99 (1%) ✅
["parallel","remotecall",("identity",512)]	0.96 (15%)	0.99 (1%) ✅
["parallel","remotecall",("identity",64)]	0.97 (15%)	0.99 (1%) ✅
["problem","raytrace","raytrace"]	1.13 (15%)	1.02 (1%) ❌
["scalar","iteration","in"]	14553.04 (25%) ❌	1.00 (1%)
["shootout","fasta"]	0.78 (15%) ✅	1.00 (1%)
["simd",("inner","Float32",4095)]	1.27 (20%) ❌	1.00 (1%)
["simd",("inner","Float32",4096)]	1.38 (20%) ❌	1.00 (1%)
["simd",("sum_reduce","Float32",4095)]	1.90 (20%) ❌	1.00 (1%)
["simd",("sum_reduce","Float32",4096)]	1.53 (20%) ❌	1.00 (1%)
["simd",("sum_reduce","Int32",4095)]	1.33 (20%) ❌	1.00 (1%)
["simd",("sum_reduce","Int32",4096)]	1.42 (20%) ❌	1.00 (1%)
["simd",("two_reductions","Float32",4095)]	1.45 (20%) ❌	1.00 (1%)
["simd",("two_reductions","Float32",4096)]	1.48 (20%) ❌	1.00 (1%)
["sort","mergesort",("sort forwards","ones")]	1.37 (30%) ❌	1.00 (1%)

["nullable","basic",("get2","Nullable{BigInt}(1)")]	1.90 (60%) ❌	1.00 (1%)
["parallel","remotecall",("identity",1024)]	0.97 (15%)	0.99 (1%) ✅
["parallel","remotecall",("identity",2)]	0.96 (15%)	0.99 (1%) ✅
["parallel","remotecall",("identity",512)]	0.96 (15%)	0.99 (1%) ✅
["parallel","remotecall",("identity",64)]	0.97 (15%)	0.99 (1%) ✅
["problem","raytrace","raytrace"]	1.13 (15%)	1.02 (1%) ❌
["scalar","iteration","in"]	14553.04 (25%) ❌	1.00 (1%)
["shootout","fasta"]	0.78 (15%) ✅	1.00 (1%)
["simd",("inner","Float32",4095)]	1.27 (20%) ❌	1.00 (1%)
["simd",("inner","Float32",4096)]	1.38 (20%) ❌	1.00 (1%)
["simd",("sum_reduce","Float32",4095)]	1.90 (20%) ❌	1.00 (1%)
["simd",("sum_reduce","Float32",4096)]	1.53 (20%) ❌	1.00 (1%)
["simd",("sum_reduce","Int32",4095)]	1.33 (20%) ❌	1.00 (1%)
["simd",("sum_reduce","Int32",4096)]	1.42 (20%) ❌	1.00 (1%)
["simd",("two_reductions","Float32",4095)]	1.45 (20%) ❌	1.00 (1%)
["simd",("two_reductions","Float32",4096)]	1.48 (20%) ❌	1.00 (1%)
["sort","mergesort",("sort forwards","ones")]	1.37 (30%) ❌	1.00 (1%)
["sort","mergesort",("sort reverse","descending")]	1.32 (30%) ❌	1.00 (1%)
["sort","mergesort",("sort! forwards","ones")]	1.40 (30%) ❌	1.00 (1%)
["sort","mergesort",("sort! reverse","descending")]	1.35 (30%) ❌	1.00 (1%)
["sort","mergesort",("sort! reverse","ones")]	1.31 (30%) ❌	1.00 (1%)
["sort","mergesort",("sortperm forwards","ascending")]	1.44 (30%) ❌	1.00 (1%)
["sort","mergesort",("sortperm forwards","descending")]	1.31 (30%) ❌	1.00 (1%)

["parallel","remotecall",("identity",1024)]	0.97 (15%)	0.99 (1%) ✅
["parallel","remotecall",("identity",2)]	0.96 (15%)	0.99 (1%) ✅
["parallel","remotecall",("identity",512)]	0.96 (15%)	0.99 (1%) ✅
["parallel","remotecall",("identity",64)]	0.97 (15%)	0.99 (1%) ✅
["problem","raytrace","raytrace"]	1.13 (15%)	1.02 (1%) ❌
["scalar","iteration","in"]	14553.04 (25%) ❌	1.00 (1%)
["shootout","fasta"]	0.78 (15%) ✅	1.00 (1%)
["simd",("inner","Float32",4095)]	1.27 (20%) ❌	1.00 (1%)
["simd",("inner","Float32",4096)]	1.38 (20%) ❌	1.00 (1%)
["simd",("sum_reduce","Float32",4095)]	1.90 (20%) ❌	1.00 (1%)
["simd",("sum_reduce","Float32",4096)]	1.53 (20%) ❌	1.00 (1%)
["simd",("sum_reduce","Int32",4095)]	1.33 (20%) ❌	1.00 (1%)
["simd",("sum_reduce","Int32",4096)]	1.42 (20%) ❌	1.00 (1%)
["simd",("two_reductions","Float32",4095)]	1.45 (20%) ❌	1.00 (1%)
["simd",("two_reductions","Float32",4096)]	1.48 (20%) ❌	1.00 (1%)
["sort","mergesort",("sort forwards","ones")]	1.37 (30%) ❌	1.00 (1%)
["sort","mergesort",("sort reverse","descending")]	1.32 (30%) ❌	1.00 (1%)
["sort","mergesort",("sort! forwards","ones")]	1.40 (30%) ❌	1.00 (1%)
["sort","mergesort",("sort! reverse","descending")]	1.35 (30%) ❌	1.00 (1%)
["sort","mergesort",("sort! reverse","ones")]	1.31 (30%) ❌	1.00 (1%)
["sort","mergesort",("sortperm forwards","ascending")]	1.44 (30%) ❌	1.00 (1%)
["sort","mergesort",("sortperm forwards","descending")]	1.31 (30%) ❌	1.00 (1%)
["sort","mergesort",("sortperm forwards","ones")]	1.50 (30%) ❌	1.00 (1%)

["parallel","remotecall",("identity",2)]	0.96 (15%)	0.99 (1%) ✅
["parallel","remotecall",("identity",512)]	0.96 (15%)	0.99 (1%) ✅
["parallel","remotecall",("identity",64)]	0.97 (15%)	0.99 (1%) ✅
["problem","raytrace","raytrace"]	1.13 (15%)	1.02 (1%) ❌
["scalar","iteration","in"]	14553.04 (25%) ❌	1.00 (1%)
["shootout","fasta"]	0.78 (15%) ✅	1.00 (1%)
["simd",("inner","Float32",4095)]	1.27 (20%) ❌	1.00 (1%)
["simd",("inner","Float32",4096)]	1.38 (20%) ❌	1.00 (1%)
["simd",("sum_reduce","Float32",4095)]	1.90 (20%) ❌	1.00 (1%)
["simd",("sum_reduce","Float32",4096)]	1.53 (20%) ❌	1.00 (1%)
["simd",("sum_reduce","Int32",4095)]	1.33 (20%) ❌	1.00 (1%)
["simd",("sum_reduce","Int32",4096)]	1.42 (20%) ❌	1.00 (1%)
["simd",("two_reductions","Float32",4095)]	1.45 (20%) ❌	1.00 (1%)
["simd",("two_reductions","Float32",4096)]	1.48 (20%) ❌	1.00 (1%)
["sort","mergesort",("sort forwards","ones")]	1.37 (30%) ❌	1.00 (1%)
["sort","mergesort",("sort reverse","descending")]	1.32 (30%) ❌	1.00 (1%)
["sort","mergesort",("sort! forwards","ones")]	1.40 (30%) ❌	1.00 (1%)
["sort","mergesort",("sort! reverse","descending")]	1.35 (30%) ❌	1.00 (1%)
["sort","mergesort",("sort! reverse","ones")]	1.31 (30%) ❌	1.00 (1%)
["sort","mergesort",("sortperm forwards","ascending")]	1.44 (30%) ❌	1.00 (1%)
["sort","mergesort",("sortperm forwards","descending")]	1.31 (30%) ❌	1.00 (1%)

["parallel","remotecall",("identity",2)]	0.96 (15%)	0.99 (1%) ✅
["parallel","remotecall",("identity",512)]	0.96 (15%)	0.99 (1%) ✅
["parallel","remotecall",("identity",64)]	0.97 (15%)	0.99 (1%) ✅
["problem","raytrace","raytrace"]	1.13 (15%)	1.02 (1%) ❌
["scalar","iteration","in"]	14553.04 (25%) ❌	1.00 (1%)
["shootout","fasta"]	0.78 (15%) ✅	1.00 (1%)
["simd",("inner","Float32",4095)]	1.27 (20%) ❌	1.00 (1%)
["simd",("inner","Float32",4096)]	1.38 (20%) ❌	1.00 (1%)
["simd",("sum_reduce","Float32",4095)]	1.90 (20%) ❌	1.00 (1%)
["simd",("sum_reduce","Float32",4096)]	1.53 (20%) ❌	1.00 (1%)
["simd",("sum_reduce","Int32",4095)]	1.33 (20%) ❌	1.00 (1%)
["simd",("sum_reduce","Int32",4096)]	1.42 (20%) ❌	1.00 (1%)
["simd",("two_reductions","Float32",4095)]	1.45 (20%) ❌	1.00 (1%)
["simd",("two_reductions","Float32",4096)]	1.48 (20%) ❌	1.00 (1%)
["sort","mergesort",("sort forwards","ones")]	1.37 (30%) ❌	1.00 (1%)
["sort","mergesort",("sort reverse","descending")]	1.32 (30%) ❌	1.00 (1%)
["sort","mergesort",("sort! forwards","ones")]	1.40 (30%) ❌	1.00 (1%)
["sort","mergesort",("sort! reverse","descending")]	1.35 (30%) ❌	1.00 (1%)
["sort","mergesort",("sort! reverse","ones")]	1.31 (30%) ❌	1.00 (1%)
["sort","mergesort",("sortperm forwards","ascending")]	1.44 (30%) ❌	1.00 (1%)
["sort","mergesort",("sortperm forwards","descending")]	1.31 (30%) ❌	1.00 (1%)

6

tkelman commented 12 days ago — Member

@nanosoldier `runbenchmarks(ALL, vs = ":master")`

nanosoldier commented 12 days ago — Member

Your benchmark job has completed - possible performance regressions were detected. A full report can be found here. cc @jrevels

1

vtjnash reviewed 9 days ago — View changes

src/codegen.cpp — Show outdated

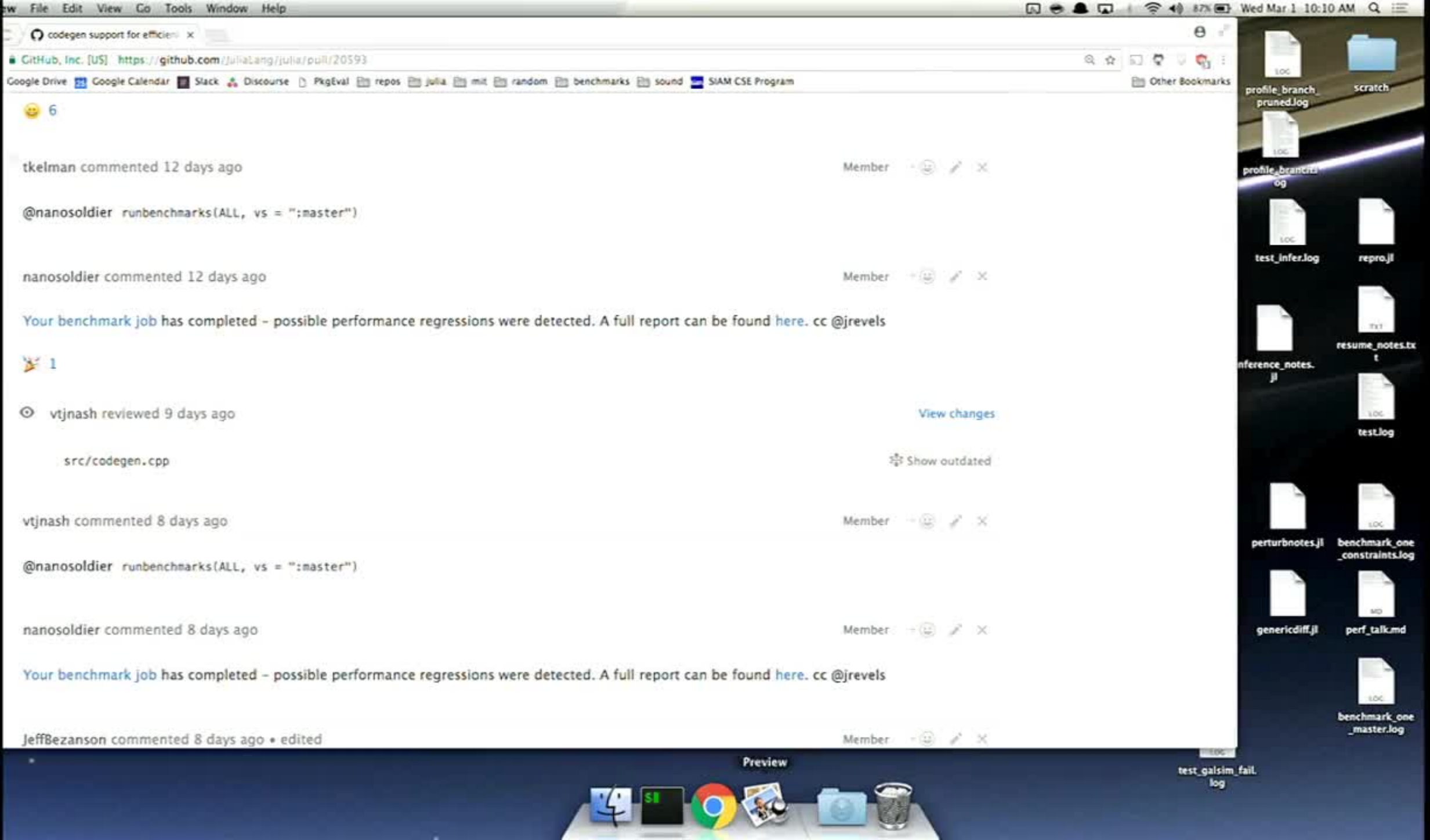
vtjnash commented 8 days ago — Member

@nanosoldier `runbenchmarks(ALL, vs = ":master")`

nanosoldier commented 8 days ago — Member

Your benchmark job has completed - possible performance regressions were detected. A full report can be found here. cc @jrevels

JeffBezanson commented 8 days ago • edited — Member


File Edit View Go Tools Window Help
Wed Mar 1 10:10 AM
codegen support for efficien
GitHub, Inc. [US] https://github.com/JuliaLang/julia/pull/20593
Google Drive Google Calendar Slack Discourse PkgEval repos julia mit random benchmarks sound SIAM CSE Program
Other Bookmarks
6
tkelman commented 12 days ago
Member
@nanosoldier runbenchmarks(ALL, vs = ":master")
nanosoldier commented 12 days ago
Member
Your benchmark job has completed - possible performance regressions were detected. A full report can be found here. cc @jrevels
1
vtjnash reviewed 9 days ago
View changes
src/codegen.cpp
Show outdated
vtjnash commented 8 days ago
Member
@nanosoldier runbenchmarks(ALL, vs = ":master")
nanosoldier commented 8 days ago
Member
Your benchmark job has completed - possible performance regressions were detected. A full report can be found here. cc @jrevels
JeffBezanson commented 8 days ago • edited
Member
profile_branch_pruned.log
scratch
profile_branch.log
test_infer.log
repro.jl
resume_notes.txt
test.log
perturbnotes.jl
benchmark_one_constraints.log
genericdiff.jl
perf_talk.md
benchmark_one_master.log
Preview
test_galsim_fail.log

Okay, but what about in the real world?

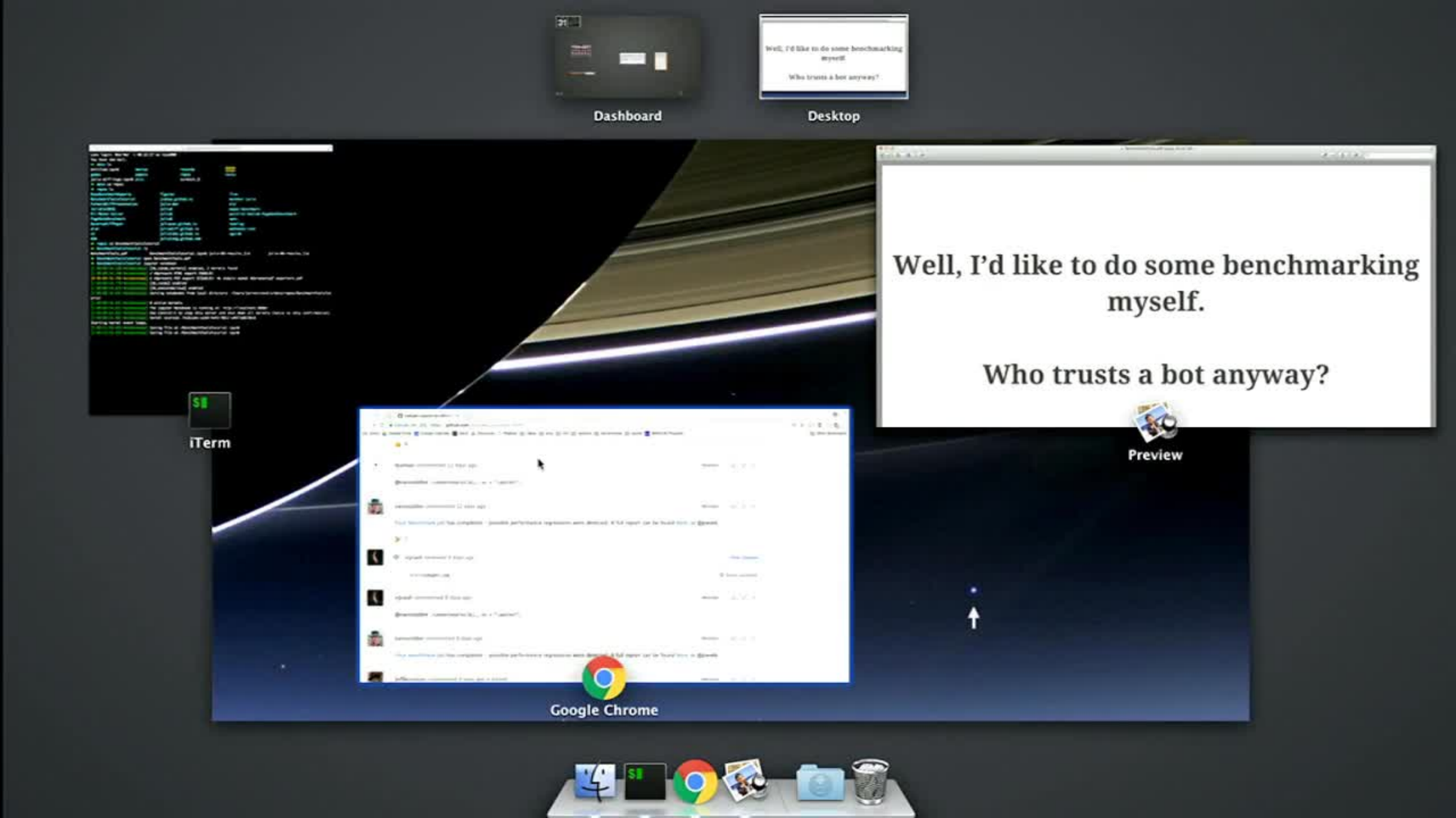
https://github.com/JuliaLang/julia/pull/20593

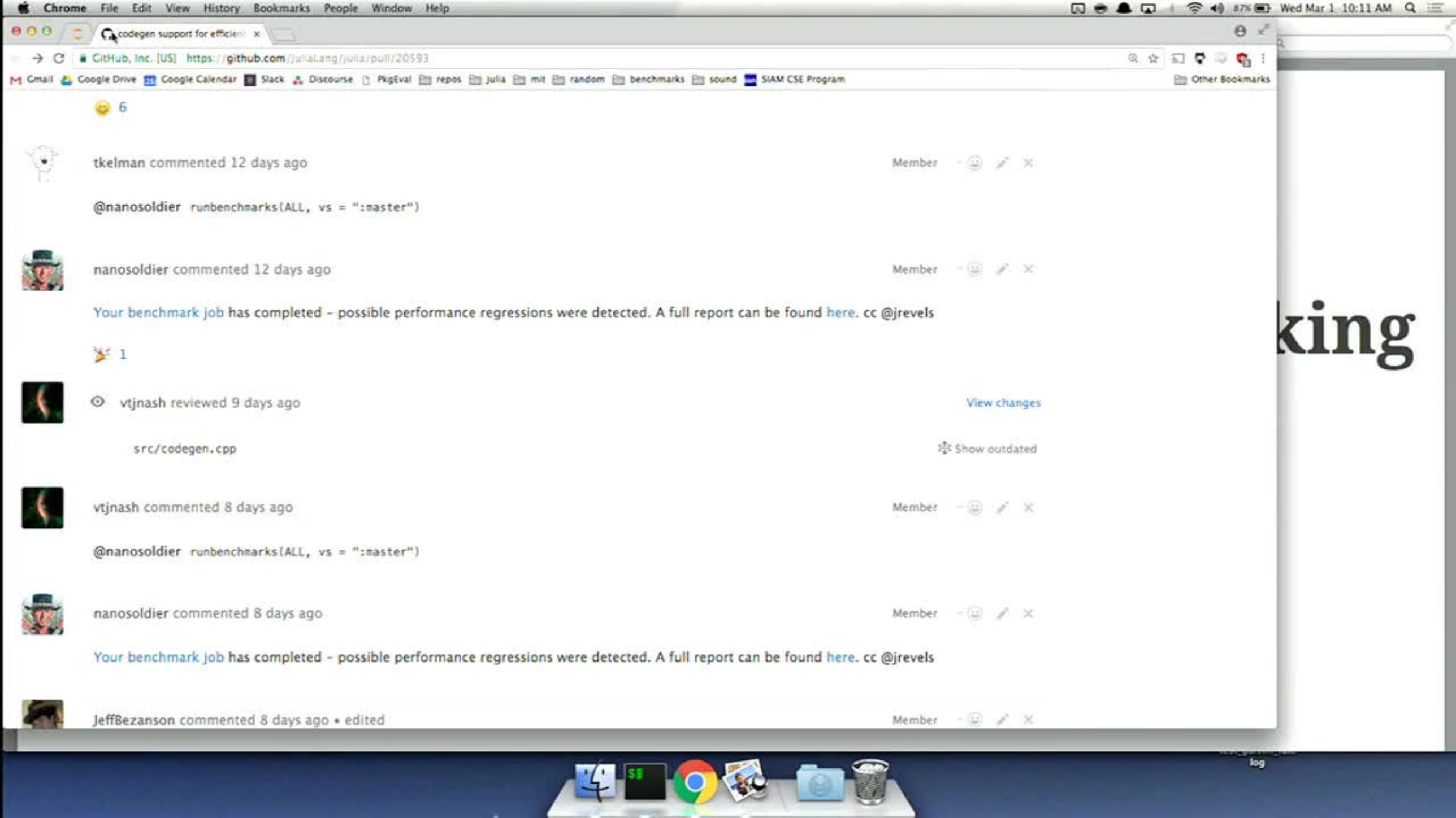
Well, I'd like to do some benchmarking myself.

Who trusts a bot anyway?

Well, I'd like to do some benchmarking myself.

Who trusts a bot anyway?


Dashboard
Desktop
Well, I'd like to do some benchmarking myself.
Who trusts a bot anyway?
iTerm
Preview
Google Chrome


Chrome File Edit View History Bookmarks People Window Help
Wed Mar 1 10:11 AM
codegen support for efficien
GitHub, Inc. [US] https://github.com/JuliaLang/julia/pull/20593
Gmail Google Drive Google Calendar Slack Discourse PkgEval repos julia mit random benchmarks sound SIAM CSE Program
Other Bookmarks
6
tkelman commented 12 days ago
Member
@nanosoldier runbenchmarks(ALL, vs = ":master")
nanosoldier commented 12 days ago
Member
Your benchmark job has completed - possible performance regressions were detected. A full report can be found here. cc @jrevels
1
vtjnash reviewed 9 days ago
View changes
src/codegen.cpp
Show outdated
vtjnash commented 8 days ago
Member
@nanosoldier runbenchmarks(ALL, vs = ":master")
nanosoldier commented 8 days ago
Member
Your benchmark job has completed - possible performance regressions were detected. A full report can be found here. cc @jrevels
JeffBezanson commented 8 days ago • edited
Member
king

Chrome File Edit View History Bookmarks People Window Help

Wed Mar 1 10:11 AM

codegen support for efficient

localhost:8888/notebooks/BenchmarkToolsTutorial.ipynb

Gmail Google Drive Google Calendar Slack Discourse PkgEval repos julia mit random benchmarks sound SIAM CSE Program Other Bookmarks

jupyter BenchmarkToolsTutorial Last Checkpoint: a day ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Julia 0.5.1-pre

Code CellToolbar

In [1]:
```
# Load BenchmarkTools and BaseBenchmarks
using BenchmarkTools, BaseBenchmarks

# Load BenchmarkTools' plotting convenience functions
BenchmarkTools.loadplotting()

# Load all the benchmark suites into BaseBenchmarks.SUITE
BaseBenchmarks.loadall!()

suite = BaseBenchmarks.SUITE
```

```
loading group "string"...done (took 0.639372886 seconds)
loading group "linalg"...done (took 9.283544043 seconds)
loading group "parallel"...done (took 0.029750334 seconds)
loading group "tuple"...done (took 0.781131452 seconds)
loading group "dates"...done (took 1.700275288 seconds)
loading group "micro"...done (took 0.223731621 seconds)
loading group "io"...done (took 0.499256112 seconds)
loading group "scalar"...done (took 24.248784867 seconds)
loading group "sparse"...done (took 5.167897328 seconds)
loading group "broadcast"...done (took 0.594702414 seconds)
loading group "simd"...done (took 2.345228074 seconds)
loading group "problem"...done (took 2.367438322 seconds)
loading group "array"...done (took 21.728793088 seconds)
loading group "misc"...done (took 1.033896241 seconds)
```

king

log

Chrome File Edit View History Bookmarks People Window Help

Wed Mar 1 10:11 AM

codegen support for efficient

localhost:8888/notebooks/BenchmarkToolsTutorial.ipynb

Gmail Google Drive Google Calendar Slack Discourse PkgEval repos julia mit random benchmarks sound SIAM CSE Program Other Bookmarks

jupyter BenchmarkToolsTutorial Last Checkpoint: a day ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Julia 0.5.1-pre

Code CellToolbar

In [1]:
```
# Load BenchmarkTools and BaseBenchmarks
using BenchmarkTools, BaseBenchmarks

# Load BenchmarkTools' plotting convenience functions
BenchmarkTools.loadplotting()

# Load all the benchmark suites into BaseBenchmarks.SUITE
BaseBenchmarks.loadall!()

suite = BaseBenchmarks.SUITE
```

```
loading group "string"...done (took 0.639372886 seconds)
loading group "linalg"...done (took 9.283544043 seconds)
loading group "parallel"...done (took 0.029750334 seconds)
loading group "tuple"...done (took 0.781131452 seconds)
loading group "dates"...done (took 1.700275288 seconds)
loading group "micro"...done (took 0.223731621 seconds)
loading group "io"...done (took 0.499256112 seconds)
loading group "scalar"...done (took 24.248784867 seconds)
loading group "sparse"...done (took 5.167897328 seconds)
loading group "broadcast"...done (took 0.594702414 seconds)
loading group "simd"...done (took 2.345228074 seconds)
loading group "problem"...done (took 2.367438322 seconds)
loading group "array"...done (took 21.728793088 seconds)
loading group "misc"...done (took 1.033896241 seconds)
```

king

log

jupyter BenchmarkToolsTutorial Last Checkpoint: a day ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help Julia 0.5.1-pre

Toggle Header
Toggle Toolbar
Cell Toolbar
Toggle Presentation

Code CellToolbar

In [1]:
```
ools and BaseBenchmarks
using BenchmarkTools, BaseBenchmarks

# Load BenchmarkTools' plotting convenience functions
BenchmarkTools.loadplotting()

# Load all the benchmark suites into BaseBenchmarks.SUITE
BaseBenchmarks.loadall!()

suite = BaseBenchmarks.SUITE
```

```
loading group "string"...done (took 0.639372886 seconds)
loading group "linalg"...done (took 9.283544043 seconds)
loading group "parallel"...done (took 0.029750334 seconds)
loading group "tuple"...done (took 0.781131452 seconds)
loading group "dates"...done (took 1.700275288 seconds)
loading group "micro"...done (took 0.223731621 seconds)
loading group "io"...done (took 0.499256112 seconds)
loading group "scalar"...done (took 24.248784867 seconds)
loading group "sparse"...done (took 5.167897328 seconds)
loading group "broadcast"...done (took 0.594702414 seconds)
loading group "simd"...done (took 2.345228074 seconds)
loading group "problem"...done (took 2.367438322 seconds)
loading group "array"...done (took 21.728793088 seconds)
loading group "misc"...done (took 1.033896241 seconds)
loading group "sort"...done (took 3.709516544 seconds)
loading group "nullable"...done (took 5.823330827 seconds)
loading group "shootout"...done (took 1.002384346 seconds)
```

Out[1]:
```
17-element BenchmarkTools.BenchmarkGroup:
  tags: []
```

In [ ]:
```
# Let's get a closer look...
showall(suite)
```

File Edit View Insert Cell Kernel Widgets Help

Julia 0.5.1-pre

Toggle Header
Toggle Toolbar
Cell Toolbar
Toggle Presentation

Code CellToolbar

In [1]:
```
ools and BaseBenchmarks
using BenchmarkTools, BaseBenchmarks

# Load BenchmarkTools' plotting convenience functions
BenchmarkTools.loadplotting()

# Load all the benchmark suites into BaseBenchmarks.SUITE
BaseBenchmarks.loadall!()

suite = BaseBenchmarks.SUITE
```

```
loading group "string"...done (took 0.639372886 seconds)
loading group "linalg"...done (took 9.283544043 seconds)
loading group "parallel"...done (took 0.029750334 seconds)
loading group "tuple"...done (took 0.781131452 seconds)
loading group "dates"...done (took 1.700275288 seconds)
loading group "micro"...done (took 0.223731621 seconds)
loading group "io"...done (took 0.499256112 seconds)
loading group "scalar"...done (took 24.248784867 seconds)
loading group "sparse"...done (took 5.167897328 seconds)
loading group "broadcast"...done (took 0.594702414 seconds)
loading group "simd"...done (took 2.345228074 seconds)
loading group "problem"...done (took 2.367438322 seconds)
loading group "array"...done (took 21.728793088 seconds)
loading group "misc"...done (took 1.033896241 seconds)
loading group "sort"...done (took 3.709516544 seconds)
loading group "nullable"...done (took 5.823330827 seconds)
loading group "shootout"...done (took 1.002384346 seconds)
```

Out[1]:
```
17-element BenchmarkTools.BenchmarkGroup:
  tags: []
```

In [ ]:
```
# Let's get a closer look...
showall(suite)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
```

File Edit View Insert Cell Kernel Widgets Help

Julia 0.5.1-pre

In [1]:
```
# Load BenchmarkTools and BaseBenchmarks
using BenchmarkTools, BaseBenchmarks

# Load BenchmarkTools' plotting convenience functions
BenchmarkTools.loadplotting()

# Load all the benchmark suites into BaseBenchmarks.SUITE
BaseBenchmarks.loadall!()

suite = BaseBenchmarks.SUITE
```

```
loading group "string"...done (took 0.639372886 seconds)
loading group "linalg"...done (took 9.283544043 seconds)
loading group "parallel"...done (took 0.029750334 seconds)
loading group "tuple"...done (took 0.781131452 seconds)
loading group "dates"...done (took 1.700275288 seconds)
loading group "micro"...done (took 0.223731621 seconds)
loading group "io"...done (took 0.499256112 seconds)
loading group "scalar"...done (took 24.248784867 seconds)
loading group "sparse"...done (took 5.167897328 seconds)
loading group "broadcast"...done (took 0.594702414 seconds)
loading group "simd"...done (took 2.345228074 seconds)
loading group "problem"...done (took 2.367438322 seconds)
loading group "array"...done (took 21.728793088 seconds)
loading group "misc"...done (took 1.033896241 seconds)
loading group "sort"...done (took 3.709516544 seconds)
loading group "nullable"...done (took 5.823330827 seconds)
loading group "shootout"...done (took 1.002384346 seconds)
```

Out[1]:
```
17-element BenchmarkTools.BenchmarkGroup:
  tags: []
```

In [ ]:
```
# Let's get a closer look...
showall(suite)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

File Edit View Insert Cell Kernel Widgets Help

Julia 0.5.1-pre

In [1]:
```
# Load BenchmarkTools and BaseBenchmarks
using BenchmarkTools, BaseBenchmarks

# Load BenchmarkTools' plotting convenience functions
BenchmarkTools.loadplotting()

# Load all the benchmark suites into BaseBenchmarks.SUITE
BaseBenchmarks.loadall!()

suite = BaseBenchmarks.SUITE
```

```
loading group "string"...done (took 0.639372886 seconds)
loading group "linalg"...done (took 9.283544043 seconds)
loading group "parallel"...done (took 0.029750334 seconds)
loading group "tuple"...done (took 0.781131452 seconds)
loading group "dates"...done (took 1.700275288 seconds)
loading group "micro"...done (took 0.223731621 seconds)
loading group "io"...done (took 0.499256112 seconds)
loading group "scalar"...done (took 24.248784867 seconds)
loading group "sparse"...done (took 5.167897328 seconds)
loading group "broadcast"...done (took 0.594702414 seconds)
loading group "simd"...done (took 2.345228074 seconds)
loading group "problem"...done (took 2.367438322 seconds)
loading group "array"...done (took 21.728793088 seconds)
loading group "misc"...done (took 1.033896241 seconds)
loading group "sort"...done (took 3.709516544 seconds)
loading group "nullable"...done (took 5.823330827 seconds)
loading group "shootout"...done (took 1.002384346 seconds)
```

Out[1]:
```
17-element BenchmarkTools.BenchmarkGroup:
  tags: []
```

In [ ]:
```
# Let's get a closer look...
showall(suite)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

File Edit View Insert Cell Kernel Widgets Help

Julia 0.5.1-pre

In [1]:
```
# Load BenchmarkTools and BaseBenchmarks
using BenchmarkTools, BaseBenchmarks

# Load BenchmarkTools' plotting convenience functions
BenchmarkTools.loadplotting()

# Load all the benchmark suites into BaseBenchmarks.SUITE
BaseBenchmarks.loadall!()

suite = BaseBenchmarks.SUITE
```

```
loading group "string"...done (took 0.639372886 seconds)
loading group "linalg"...done (took 9.283544043 seconds)
loading group "parallel"...done (took 0.029750334 seconds)
loading group "tuple"...done (took 0.781131452 seconds)
loading group "dates"...done (took 1.700275288 seconds)
loading group "micro"...done (took 0.223731621 seconds)
loading group "io"...done (took 0.499256112 seconds)
loading group "scalar"...done (took 24.248784867 seconds)
loading group "sparse"...done (took 5.167897328 seconds)
loading group "broadcast"...done (took 0.594702414 seconds)
loading group "simd"...done (took 2.345228074 seconds)
loading group "problem"...done (took 2.367438322 seconds)
loading group "array"...done (took 21.728793088 seconds)
loading group "misc"...done (took 1.033896241 seconds)
loading group "sort"...done (took 3.709516544 seconds)
loading group "nullable"...done (took 5.823330827 seconds)
loading group "shootout"...done (took 1.002384346 seconds)
```

Out[1]:
```
17-element BenchmarkTools.BenchmarkGroup:
  tags: []
```

In [ ]:
```
# Let's get a closer look...
showall(suite)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
```

```
using BenchmarkTools, BaseBenchmarks

# Load BenchmarkTools' plotting convenience functions
BenchmarkTools.loadplotting()

# Load all the benchmark suites into BaseBenchmarks.SUITE
BaseBenchmarks.loadall!()

suite = BaseBenchmarks.SUITE
```

```
loading group "string"...done (took 0.639372886 seconds)
loading group "linalg"...done (took 9.283544043 seconds)
loading group "parallel"...done (took 0.029750334 seconds)
loading group "tuple"...done (took 0.781131452 seconds)
loading group "dates"...done (took 1.700275288 seconds)
loading group "micro"...done (took 0.223731621 seconds)
loading group "io"...done (took 0.499256112 seconds)
loading group "scalar"...done (took 24.248784867 seconds)
loading group "sparse"...done (took 5.167897328 seconds)
loading group "broadcast"...done (took 0.594702414 seconds)
loading group "simd"...done (took 2.345228074 seconds)
loading group "problem"...done (took 2.367438322 seconds)
loading group "array"...done (took 21.728793088 seconds)
loading group "misc"...done (took 1.033896241 seconds)
loading group "sort"...done (took 3.709516544 seconds)
loading group "nullable"...done (took 5.823330827 seconds)
loading group "shootout"...done (took 1.002384346 seconds)
```

Out[1]:
```
17-element BenchmarkTools.BenchmarkGroup:
  tags: []
```

In [ ]:
```
# Let's get a closer look...
showall(suite)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
```

```
# Load BenchmarkTools' plotting convenience functions
BenchmarkTools.loadplotting()

# Load all the benchmark suites into BaseBenchmarks.SUITE
BaseBenchmarks.loadall!()

suite = BaseBenchmarks.SUITE
```

```
loading group "string"...done (took 0.639372886 seconds)
loading group "linalg"...done (took 9.283544043 seconds)
loading group "parallel"...done (took 0.029750334 seconds)
loading group "tuple"...done (took 0.781131452 seconds)
loading group "dates"...done (took 1.700275288 seconds)
loading group "micro"...done (took 0.223731621 seconds)
loading group "io"...done (took 0.499256112 seconds)
loading group "scalar"...done (took 24.248784867 seconds)
loading group "sparse"...done (took 5.167897328 seconds)
loading group "broadcast"...done (took 0.594702414 seconds)
loading group "simd"...done (took 2.345228074 seconds)
loading group "problem"...done (took 2.367438322 seconds)
loading group "array"...done (took 21.728793088 seconds)
loading group "misc"...done (took 1.033896241 seconds)
loading group "sort"...done (took 3.709516544 seconds)
loading group "nullable"...done (took 5.823330827 seconds)
loading group "shootout"...done (took 1.002384346 seconds)
```

Out[1]:
```
17-element BenchmarkTools.BenchmarkGroup:
  tags: []
```

In [ ]:
```
# Let's get a closer look...
showall(suite)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

File Edit View Insert Cell Kernel Widgets Help Julia 0.5.1-pre

```
BaseBenchmarks.loadall!()

suite = BaseBenchmarks.SUITE
```

```
loading group "string"...done (took 0.639372886 seconds)
loading group "linalg"...done (took 9.283544043 seconds)
loading group "parallel"...done (took 0.029750334 seconds)
loading group "tuple"...done (took 0.781131452 seconds)
loading group "dates"...done (took 1.700275288 seconds)
loading group "micro"...done (took 0.223731621 seconds)
loading group "io"...done (took 0.499256112 seconds)
loading group "scalar"...done (took 24.248784867 seconds)
loading group "sparse"...done (took 5.167897328 seconds)
loading group "broadcast"...done (took 0.594702414 seconds)
loading group "simd"...done (took 2.345228074 seconds)
loading group "problem"...done (took 2.367438322 seconds)
loading group "array"...done (took 21.728793088 seconds)
loading group "misc"...done (took 1.033896241 seconds)
loading group "sort"...done (took 3.709516544 seconds)
loading group "nullable"...done (took 5.823330827 seconds)
loading group "shootout"...done (took 1.002384346 seconds)
```

Out[1]:
```
17-element BenchmarkTools.BenchmarkGroup:
  tags: []
```

In [ ]:
```
# Let's get a closer look...
showall(suite)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

```
BaseBenchmarks.loadall!()

suite = BaseBenchmarks.SUITE
```

```
loading group "string"...done (took 0.639372886 seconds)
loading group "linalg"...done (took 9.283544043 seconds)
loading group "parallel"...done (took 0.029750334 seconds)
loading group "tuple"...done (took 0.781131452 seconds)
loading group "dates"...done (took 1.700275288 seconds)
loading group "micro"...done (took 0.223731621 seconds)
loading group "io"...done (took 0.499256112 seconds)
loading group "scalar"...done (took 24.248784867 seconds)
loading group "sparse"...done (took 5.167897328 seconds)
loading group "broadcast"...done (took 0.594702414 seconds)
loading group "simd"...done (took 2.345228074 seconds)
loading group "problem"...done (took 2.367438322 seconds)
loading group "array"...done (took 21.728793088 seconds)
loading group "misc"...done (took 1.033896241 seconds)
loading group "sort"...done (took 3.709516544 seconds)
loading group "nullable"...done (took 5.823330827 seconds)
loading group "shootout"...done (took 1.002384346 seconds)
```

Out[1]:
```
17-element BenchmarkTools.BenchmarkGroup:
  tags: []
```

In [ ]:
```
# Let's get a closer look...
showall(suite)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

```
# Load all the benchmark suites into BaseBenchmarks.SUITE
BaseBenchmarks.loadall!()

suite = BaseBenchmarks.SUITE
```

```
loading group "micro"...done (took 0.223731621 seconds)
loading group "io"...done (took 0.499256112 seconds)
loading group "scalar"...done (took 24.248784867 seconds)
loading group "sparse"...done (took 5.167897328 seconds)
loading group "broadcast"...done (took 0.594702414 seconds)
loading group "simd"...done (took 2.345228074 seconds)
loading group "problem"...done (took 2.367438322 seconds)
loading group "array"...done (took 21.728793088 seconds)
loading group "misc"...done (took 1.033896241 seconds)
loading group "sort"...done (took 3.709516544 seconds)
loading group "nullable"...done (took 5.823330827 seconds)
loading group "shootout"...done (took 1.002384346 seconds)
```

Out[1]:
```
17-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkGroup([])
  "linalg" => 3-element BenchmarkGroup(["array"])
  "parallel" => 0-element BenchmarkGroup([])
  "tuple" => 1-element BenchmarkGroup([])
  "dates" => 7-element BenchmarkGroup([])
  "micro" => 7-element BenchmarkGroup(["recursion", "fibonacci", "fib", "parse", "parseint", "mandel", "mandelbrot",
```

In [ ]:
```
# Let's get a closer look...
showall(suite)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

```
# Load all the benchmark suites into BaseBenchmarks.SUITE
BaseBenchmarks.loadall!()

suite = BaseBenchmarks.SUITE
```

```
loading group "broadcast"...done (took 0.594702414 seconds)
loading group "simd"...done (took 2.345228074 seconds)
loading group "problem"...done (took 2.367438322 seconds)
loading group "array"...done (took 21.728793088 seconds)
loading group "misc"...done (took 1.033896241 seconds)
loading group "sort"...done (took 3.709516544 seconds)
loading group "nullable"...done (took 5.823330827 seconds)
loading group "shootout"...done (took 1.002384346 seconds)
```

Out[1]:

```
17-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkGroup([])
  "linalg" => 3-element BenchmarkGroup(["array"])
  "parallel" => 0-element BenchmarkGroup([])
  "tuple" => 1-element BenchmarkGroup([])
  "dates" => 7-element BenchmarkGroup([])
  "micro" => 7-element BenchmarkGroup(["recursion", "fibonacci", "fib", "parse", "parseint", "mandel", "mandelbrot",
 "sort", "quicksort", "pi", "π", "sum", "pisum", "πsum", "rand", "randmatstat", "rand", "randmatmul"])
  "io" => 2-element BenchmarkGroup([])
  "scalar" => 5-element BenchmarkGroup([])
```

In [ ]:

```
# Let's get a closer look...
showall(suite)
```

In [ ]:

```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

In [ ]:

```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:

```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

```
# Load all the benchmark suites into BaseBenchmarks.SUITE
BaseBenchmarks.loadall!()

suite = BaseBenchmarks.SUITE
```

```
loading group "broadcast"...done (took 0.594702414 seconds)
loading group "simd"...done (took 2.345228074 seconds)
loading group "problem"...done (took 2.367438322 seconds)
loading group "array"...done (took 21.728793088 seconds)
loading group "misc"...done (took 1.033896241 seconds)
loading group "sort"...done (took 3.709516544 seconds)
loading group "nullable"...done (took 5.823330827 seconds)
loading group "shootout"...done (took 1.002384346 seconds)
```

Out[1]:

```
17-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkGroup([])
  "linalg" => 3-element BenchmarkGroup(["array"])
  "parallel" => 0-element BenchmarkGroup([])
  "tuple" => 1-element BenchmarkGroup([])
  "dates" => 7-element BenchmarkGroup([])
  "micro" => 7-element BenchmarkGroup(["recursion", "fibonacci", "fib", "parse", "parseint", "mandel", "mandelbrot",
 "sort", "quicksort", "pi", "π", "sum", "pisum", "πsum", "rand", "randmatstat", "rand", "randmatmul"])
  "io" => 2-element BenchmarkGroup([])
  "scalar" => 5-element BenchmarkGroup([])
```

In [ ]:

```
# Let's get a closer look...
showall(suite)
```

In [ ]:

```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

In [ ]:

```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:

```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

```
# Load all the benchmark suites into BaseBenchmarks.SUITE
BaseBenchmarks.loadall!()

suite = BaseBenchmarks.SUITE
```

```
loading group "simd"...done (took 2.345228074 seconds)
loading group "problem"...done (took 2.367438322 seconds)
loading group "array"...done (took 21.728793088 seconds)
loading group "misc"...done (took 1.033896241 seconds)
loading group "sort"...done (took 3.709516544 seconds)
loading group "nullable"...done (took 5.823330827 seconds)
loading group "shootout"...done (took 1.002384346 seconds)
```

Out[1]:
```
17-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkGroup([])
  "linalg" => 3-element BenchmarkGroup(["array"])
  "parallel" => 0-element BenchmarkGroup([])
  "tuple" => 1-element BenchmarkGroup([])
  "dates" => 7-element BenchmarkGroup([])
  "micro" => 7-element BenchmarkGroup(["recursion", "fibonacci", "fib", "parse", "parseint", "mandel", "mandelbrot",
 "sort", "quicksort", "pi", "π", "sum", "pisum", "πsum", "rand", "randmatstat", "rand", "randmatmul"])
  "io" => 2-element BenchmarkGroup([])
  "scalar" => 5-element BenchmarkGroup([])
  "sparse" => 3-element BenchmarkGroup(["array"])
```

In [ ]:
```
# Let's get a closer look...
showall(suite)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

```
loading group "misc"...done (took 1.033896241 seconds)
loading group "sort"...done (took 3.709516544 seconds)
loading group "nullable"...done (took 5.823330827 seconds)
loading group "shootout"...done (took 1.002384346 seconds)
```

Out[1]:
```
17-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkGroup([])
  "linalg" => 3-element BenchmarkGroup(["array"])
  "parallel" => 0-element BenchmarkGroup([])
  "tuple" => 1-element BenchmarkGroup([])
  "dates" => 7-element BenchmarkGroup([])
  "micro" => 7-element BenchmarkGroup(["recursion", "fibonacci", "fib", "parse", "parseint", "mandel", "mandelbrot",
 "sort", "quicksort", "pi", "π", "sum", "pisum", "πsum", "rand", "randmatstat", "rand", "randmatmul"])
  "io" => 2-element BenchmarkGroup([])
  "scalar" => 5-element BenchmarkGroup([])
  "sparse" => 3-element BenchmarkGroup(["array"])
  "broadcast" => 3-element BenchmarkGroup([])
  "simd" => 72-element BenchmarkGroup(["array", "inbounds"])
  ⋮
```

In [ ]:
```
# Let's get a closer look...
showall(suite)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
```

Out[1]:
```
17-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkGroup([])
  "linalg" => 3-element BenchmarkGroup(["array"])
  "parallel" => 0-element BenchmarkGroup([])
  "tuple" => 1-element BenchmarkGroup([])
  "dates" => 7-element BenchmarkGroup([])
  "micro" => 7-element BenchmarkGroup(["recursion", "fibonacci", "fib", "parse", "parseint", "mandel", "mandelbrot",
 "sort", "quicksort", "pi", "π", "sum", "pisum", "πsum", "rand", "randmatstat", "rand", "randmatmul"])
  "io" => 2-element BenchmarkGroup([])
  "scalar" => 5-element BenchmarkGroup([])
  "sparse" => 3-element BenchmarkGroup(["array"])
  "broadcast" => 3-element BenchmarkGroup([])
  "simd" => 72-element BenchmarkGroup(["array", "inbounds"])
  ⋮
```

In [*]:
```
# Let's get a closer look...
showall(suite)
```

```
17-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "join" => Benchmark(evals=1, seconds=1.0, samples=10000)
          "replace" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "linalg" => 3-element BenchmarkTools.BenchmarkGroup:
          tags: ["array"]
          "factorization" => 64-element BenchmarkTools.BenchmarkGroup:
                  tags: ["eig", "svd", "lu", "qr", "schur", "chol"]
                  ("eig","LowerTriangular",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("svdfact","UpperTriangular",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("schur","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("eigfact","UpperTriangular",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("qr","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("eig","SymTridiagonal",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("lu","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("qrfact","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("eigfact","Diagonal",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("eig","LowerTriangular",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
```

```
                ("svd","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

In [ ]:
```
vs = judge(minimum(r6), minimum(r5), time_tolerance = 0.05)
showall(vs)
```

In [ ]:
```
showall(regressions(vs))
```

In [ ]:
```
showall(improvements(vs))
```

In [ ]:
```
id = ["io", "serialization", ("serialize","Matrix{Float64}")]
```

In [ ]:
```
t5 = r5[id]
```

In [ ]:
```
t6 = r6[id]
```

In [ ]:
```
plotkde([t6, t5])
```

In [ ]:

```
                ("svd","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

In [ ]:
```
vs = judge(minimum(r6), minimum(r5), time_tolerance = 0.05)
showall(vs)
```

In [ ]:
```
showall(regressions(vs))
```

In [ ]:
```
showall(improvements(vs))
```

In [ ]:
```
id = ["io", "serialization", ("serialize","Matrix{Float64}")]
```

In [ ]:
```
t5 = r5[id]
```

In [ ]:
```
t6 = r6[id]
```

In [ ]:
```
plotkde([t6, t5])
```

In [ ]:

```
                ("svd","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

In [ ]:
```
vs = judge(minimum(r6), minimum(r5), time_tolerance = 0.05)
showall(vs)
```

In [ ]:
```
showall(regressions(vs))
```

In [ ]:
```
showall(improvements(vs))
```

In [ ]:
```
id = ["io", "serialization", ("serialize","Matrix{Float64}")]
```

In [ ]:
```
t5 = r5[id]
```

In [ ]:
```
t6 = r6[id]
```

In [ ]:
```
plotkde([t6, t5])
```

In [ ]:

```
  "linalg" => 3-element BenchmarkTools.BenchmarkGroup:
          tags: ["array"]
          "factorization" => 64-element BenchmarkTools.BenchmarkGroup:
                  tags: ["eig", "svd", "lu", "qr", "schur", "chol"]
                  ("eig","LowerTriangular",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("svdfact","UpperTriangular",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("schur","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("eigfact","UpperTriangular",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("qr","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("eig","SymTridiagonal",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("lu","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("qrfact","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("eigfact","Diagonal",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("eig","LowerTriangular",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("lufact","Tridiagonal",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("schurfact","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("eig","Bidiagonal",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

In [ ]:
```
vs = judge(minimum(r6), minimum(r5), time_tolerance = 0.05)
showall(vs)
```

In [ ]:
```
showall(regressions(vs))
```

```
        tags: []
        "join" => Benchmark(evals=1, seconds=1.0, samples=10000)
        "replace" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "linalg" => 3-element BenchmarkTools.BenchmarkGroup:
        tags: ["array"]
        "factorization" => 64-element BenchmarkTools.BenchmarkGroup:
                tags: ["eig", "svd", "lu", "qr", "schur", "chol"]
                ("eig","LowerTriangular",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("svdfact","UpperTriangular",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("schur","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("eigfact","UpperTriangular",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("qr","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("eig","SymTridiagonal",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("lu","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("qrfact","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("eigfact","Diagonal",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("eig","LowerTriangular",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("lufact","Tridiagonal",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("schurfact","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("eig","Bidiagonal",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

In [ ]:
```
vs = judge(minimum(r6), minimum(r5), time_tolerance = 0.05)
```

```
  "broadcast" => 3-element BenchmarkGroup([])
  "simd" => 72-element BenchmarkGroup(["array", "inbounds"])
  ⋮
```

In [2]:
```
# Let's get a closer look...
showall(suite)
```

```
        tags: []
        "join" => Benchmark(evals=1, seconds=1.0, samples=10000)
        "replace" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "linalg" => 3-element BenchmarkTools.BenchmarkGroup:
        tags: ["array"]
        "factorization" => 64-element BenchmarkTools.BenchmarkGroup:
                tags: ["eig", "svd", "lu", "qr", "schur", "chol"]
                ("eig","LowerTriangular",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("svdfact","UpperTriangular",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("schur","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("eigfact","UpperTriangular",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("qr","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("eig","SymTridiagonal",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("lu","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("qrfact","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("eigfact","Diagonal",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("eig","LowerTriangular",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("lufact","Tridiagonal",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("schurfact","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("eig","Bidiagonal",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

```
  tags: []
  "string" => 2-element BenchmarkGroup([])
  "linalg" => 3-element BenchmarkGroup(["array"])
  "parallel" => 0-element BenchmarkGroup([])
  "tuple" => 1-element BenchmarkGroup([])
  "dates" => 7-element BenchmarkGroup([])
  "micro" => 7-element BenchmarkGroup(["recursion", "fibonacci", "fib", "parse", "parseint", "mandel", "mandelbrot",
"sort", "quicksort", "pi", "π", "sum", "pisum", "πsum", "rand", "randmatstat", "rand", "randmatmul"])
  "io" => 2-element BenchmarkGroup([])
  "scalar" => 5-element BenchmarkGroup([])
  "sparse" => 3-element BenchmarkGroup(["array"])
  "broadcast" => 3-element BenchmarkGroup([])
  "simd" => 72-element BenchmarkGroup(["array", "inbounds"])
  ⋮
```

In [2]:
```
# Let's get a closer look...
showall(suite)
```

```
tags: ["eig", "svd", "lu", "qr", "schur", "chol"]
("eig","LowerTriangular",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("svdfact","UpperTriangular",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("schur","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eigfact","UpperTriangular",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("qr","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eig","SymTridiagonal",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("lu","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("qrfact","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eigfact","Diagonal",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eig","LowerTriangular",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("lufact","Tridiagonal",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("schurfact","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eig","Bidiagonal",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eigfact","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eig","UpperTriangular",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("svdfact","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("lufact","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("svd","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("qrfact","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
```

```
  tags: []
  "string" => 2-element BenchmarkGroup([])
  "linalg" => 3-element BenchmarkGroup(["array"])
  "parallel" => 0-element BenchmarkGroup([])
  "tuple" => 1-element BenchmarkGroup([])
  "dates" => 7-element BenchmarkGroup([])
  "micro" => 7-element BenchmarkGroup(["recursion", "fibonacci", "fib", "parse", "parseint", "mandel", "mandelbrot",
"sort", "quicksort", "pi", "π", "sum", "pisum", "πsum", "rand", "randmatstat", "rand", "randmatmul"])
  "io" => 2-element BenchmarkGroup([])
  "scalar" => 5-element BenchmarkGroup([])
  "sparse" => 3-element BenchmarkGroup(["array"])
  "broadcast" => 3-element BenchmarkGroup([])
  "simd" => 72-element BenchmarkGroup(["array", "inbounds"])
  ⋮
```

In [2]:
```
# Let's get a closer look...
showall(suite)
```

```
("svdfact","UpperTriangular",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("schur","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eigfact","UpperTriangular",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("qr","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eig","SymTridiagonal",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("lu","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("qrfact","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eigfact","Diagonal",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eig","LowerTriangular",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("lufact","Tridiagonal",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("schurfact","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eig","Bidiagonal",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eigfact","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eig","UpperTriangular",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("svdfact","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("lufact","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("svd","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("qrfact","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eigfact","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("svd","Diagonal",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
```

```
  tags: []
  "string" => 2-element BenchmarkGroup([])
  "linalg" => 3-element BenchmarkGroup(["array"])
  "parallel" => 0-element BenchmarkGroup([])
  "tuple" => 1-element BenchmarkGroup([])
  "dates" => 7-element BenchmarkGroup([])
  "micro" => 7-element BenchmarkGroup(["recursion", "fibonacci", "fib", "parse", "parseint", "mandel", "mandelbrot",
"sort", "quicksort", "pi", "π", "sum", "pisum", "πsum", "rand", "randmatstat", "rand", "randmatmul"])
  "io" => 2-element BenchmarkGroup([])
  "scalar" => 5-element BenchmarkGroup([])
  "sparse" => 3-element BenchmarkGroup(["array"])
  "broadcast" => 3-element BenchmarkGroup([])
  "simd" => 72-element BenchmarkGroup(["array", "inbounds"])
  ⋮
```

In [2]:
```
# Let's get a closer look...
showall(suite)
```

```
("lufact","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("svd","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("qrfact","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eigfact","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("svd","Diagonal",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("svdfact","Diagonal",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("lufact","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("schur","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eigfact","SymTridiagonal",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("svd","LowerTriangular",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eig","UpperTriangular",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eig","Bidiagonal",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("svdfact","LowerTriangular",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("lu","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eig","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eigfact","SymTridiagonal",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eigfact","UpperTriangular",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("svdfact","Bidiagonal",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("lu","Tridiagonal",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("svd","UpperTriangular",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
```

```
  tags: []
  "string" => 2-element BenchmarkGroup([])
  "linalg" => 3-element BenchmarkGroup(["array"])
  "parallel" => 0-element BenchmarkGroup([])
  "tuple" => 1-element BenchmarkGroup([])
  "dates" => 7-element BenchmarkGroup([])
  "micro" => 7-element BenchmarkGroup(["recursion", "fibonacci", "fib", "parse", "parseint", "mandel", "mandelbrot",
"sort", "quicksort", "pi", "π", "sum", "pisum", "πsum", "rand", "randmatstat", "rand", "randmatmul"])
  "io" => 2-element BenchmarkGroup([])
  "scalar" => 5-element BenchmarkGroup([])
  "sparse" => 3-element BenchmarkGroup(["array"])
  "broadcast" => 3-element BenchmarkGroup([])
  "simd" => 72-element BenchmarkGroup(["array", "inbounds"])
  ⋮
```

In [2]:
```
# Let's get a closer look...
showall(suite)
```

```
("lu","Tridiagonal",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("cholfact","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eig","Diagonal",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eigfact","LowerTriangular",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eigfact","LowerTriangular",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("svdfact","Diagonal",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("lufact","Tridiagonal",256) => Benchmark(evals=7, seconds=1.0, samples=10000)
("svd","Bidiagonal",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eig","Diagonal",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("svd","Bidiagonal",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("eig","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("svdfact","UpperTriangular",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("chol","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("svdfact","LowerTriangular",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("svdfact","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("svd","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("qr","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("schurfact","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
("cholfact","Matrix",1024) => Benchmark(evals=1, seconds=1.0, samples=10000)
("chol","Matrix",256) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
```

```
  tags: []
  "string" => 2-element BenchmarkGroup([])
  "linalg" => 3-element BenchmarkGroup(["array"])
  "parallel" => 0-element BenchmarkGroup([])
  "tuple" => 1-element BenchmarkGroup([])
  "dates" => 7-element BenchmarkGroup([])
  "micro" => 7-element BenchmarkGroup(["recursion", "fibonacci", "fib", "parse", "parseint", "mandel", "mandelbrot",
"sort", "quicksort", "pi", "π", "sum", "pisum", "πsum", "rand", "randmatstat", "rand", "randmatmul"])
  "io" => 2-element BenchmarkGroup([])
  "scalar" => 5-element BenchmarkGroup([])
  "sparse" => 3-element BenchmarkGroup(["array"])
  "broadcast" => 3-element BenchmarkGroup([])
  "simd" => 72-element BenchmarkGroup(["array", "inbounds"])
  ⋮
```

In [2]:
```
# Let's get a closer look...
showall(suite)
```

```
                    "herk" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "syrk" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "trsm" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "asum" => Benchmark(evals=867, seconds=1.0, samples=10000)
                    "trmv!" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "gemm" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "symm!" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "scal" => Benchmark(evals=10, seconds=1.0, samples=10000)
                    "nrm2" => Benchmark(evals=313, seconds=1.0, samples=10000)
                    "trmm!" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "trmm" => Benchmark(evals=1, seconds=1.0, samples=10000)
            "arithmetic" => 108-element BenchmarkTools.BenchmarkGroup:
                    tags: []
                    ("cumsum!",Int32,256) => Benchmark(evals=934, seconds=20.0, samples=100)
                    ("\\","Bidiagonal","Vector",256) => Benchmark(evals=9, seconds=20.0, samples=100)
                    ("*","Diagonal","Diagonal",1024) => Benchmark(evals=34, seconds=20.0, samples=100)
                    ("+","Tridiagonal","Tridiagonal",1024) => Benchmark(evals=8, seconds=20.0, samples=100)
                    ("-","Matrix","Matrix",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("cumsum!",Float32,1024) => Benchmark(evals=10, seconds=20.0, samples=100)
                    ("\\","Diagonal","Vector",256) => Benchmark(evals=166, seconds=20.0, samples=100)
                    ("+","Matrix","Matrix",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
```

```
  tags: []
  "string" => 2-element BenchmarkGroup([])
  "linalg" => 3-element BenchmarkGroup(["array"])
  "parallel" => 0-element BenchmarkGroup([])
  "tuple" => 1-element BenchmarkGroup([])
  "dates" => 7-element BenchmarkGroup([])
  "micro" => 7-element BenchmarkGroup(["recursion", "fibonacci", "fib", "parse", "parseint", "mandel", "mandelbrot",
"sort", "quicksort", "pi", "π", "sum", "pisum", "πsum", "rand", "randmatstat", "rand", "randmatmul"])
  "io" => 2-element BenchmarkGroup([])
  "scalar" => 5-element BenchmarkGroup([])
  "sparse" => 3-element BenchmarkGroup(["array"])
  "broadcast" => 3-element BenchmarkGroup([])
  "simd" => 72-element BenchmarkGroup(["array", "inbounds"])
  ⋮
```

In [2]:
```
# Let's get a closer look...
showall(suite)
```

```
                    "herk" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "syrk" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "trsm" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "asum" => Benchmark(evals=867, seconds=1.0, samples=10000)
                    "trmv!" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "gemm" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "symm!" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "scal" => Benchmark(evals=10, seconds=1.0, samples=10000)
                    "nrm2" => Benchmark(evals=313, seconds=1.0, samples=10000)
                    "trmm!" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "trmm" => Benchmark(evals=1, seconds=1.0, samples=10000)
            "arithmetic" => 108-element BenchmarkTools.BenchmarkGroup:
                    tags: []
                    ("cumsum!",Int32,256) => Benchmark(evals=934, seconds=20.0, samples=100)
                    ("\\","Bidiagonal","Vector",256) => Benchmark(evals=9, seconds=20.0, samples=100)
                    ("*","Diagonal","Diagonal",1024) => Benchmark(evals=34, seconds=20.0, samples=100)
                    ("+","Tridiagonal","Tridiagonal",1024) => Benchmark(evals=8, seconds=20.0, samples=100)
                    ("-","Matrix","Matrix",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("cumsum!",Float32,1024) => Benchmark(evals=10, seconds=20.0, samples=100)
                    ("\\","Diagonal","Vector",256) => Benchmark(evals=166, seconds=20.0, samples=100)
                    ("+","Matrix","Matrix",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
```

```
  tags: []
  "string" => 2-element BenchmarkGroup([])
  "linalg" => 3-element BenchmarkGroup(["array"])
  "parallel" => 0-element BenchmarkGroup([])
  "tuple" => 1-element BenchmarkGroup([])
  "dates" => 7-element BenchmarkGroup([])
  "micro" => 7-element BenchmarkGroup(["recursion", "fibonacci", "fib", "parse", "parseint", "mandel", "mandelbrot",
"sort", "quicksort", "pi", "π", "sum", "pisum", "πsum", "rand", "randmatstat", "rand", "randmatmul"])
  "io" => 2-element BenchmarkGroup([])
  "scalar" => 5-element BenchmarkGroup([])
  "sparse" => 3-element BenchmarkGroup(["array"])
  "broadcast" => 3-element BenchmarkGroup([])
  "simd" => 72-element BenchmarkGroup(["array", "inbounds"])
  ⋮
```

In [2]:
```
# Let's get a closer look...
showall(suite)
```

```
                    "dotc" => Benchmark(evals=349, seconds=1.0, samples=10000)
                    "axpy!" => Benchmark(evals=555, seconds=1.0, samples=10000)
                    "syrk!" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "sbmv" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "sbmv!" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "symv!" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "symm" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "herk" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "syrk" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "trsm" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "asum" => Benchmark(evals=867, seconds=1.0, samples=10000)
                    "trmv!" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "gemm" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "symm!" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "scal" => Benchmark(evals=10, seconds=1.0, samples=10000)
                    "nrm2" => Benchmark(evals=313, seconds=1.0, samples=10000)
                    "trmm!" => Benchmark(evals=1, seconds=1.0, samples=10000)
                    "trmm" => Benchmark(evals=1, seconds=1.0, samples=10000)
                "arithmetic" => 108-element BenchmarkTools.BenchmarkGroup:
                        tags: []
                        ("gcdgcd!", Int32, 256) => Benchmark(evals=834, seconds=20.0, samples=100)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
```

```
  tags: []
  "string" => 2-element BenchmarkGroup([])
  "linalg" => 3-element BenchmarkGroup(["array"])
  "parallel" => 0-element BenchmarkGroup([])
  "tuple" => 1-element BenchmarkGroup([])
  "dates" => 7-element BenchmarkGroup([])
  "micro" => 7-element BenchmarkGroup(["recursion", "fibonacci", "fib", "parse", "parseint", "mandel", "mandelbrot",
"sort", "quicksort", "pi", "π", "sum", "pisum", "πsum", "rand", "randmatstat", "rand", "randmatmul"])
  "io" => 2-element BenchmarkGroup([])
  "scalar" => 5-element BenchmarkGroup([])
  "sparse" => 3-element BenchmarkGroup(["array"])
  "broadcast" => 3-element BenchmarkGroup([])
  "simd" => 72-element BenchmarkGroup(["array", "inbounds"])
  ⋮
```

In [2]:
```
# Let's get a closer look...
showall(suite)
```

```
tags: []
"ger!" => Benchmark(evals=1, seconds=1.0, samples=10000)
"gbmv!" => Benchmark(evals=1, seconds=1.0, samples=10000)
"gbmv" => Benchmark(evals=1, seconds=1.0, samples=10000)
"gemm!" => Benchmark(evals=1, seconds=1.0, samples=10000)
"gemv!" => Benchmark(evals=1, seconds=1.0, samples=10000)
"blascopy!" => Benchmark(evals=864, seconds=1.0, samples=10000)
"scal!" => Benchmark(evals=530, seconds=1.0, samples=10000)
"gemv" => Benchmark(evals=1, seconds=1.0, samples=10000)
"syr!" => Benchmark(evals=1, seconds=1.0, samples=10000)
"trsm!" => Benchmark(evals=1, seconds=1.0, samples=10000)
"trsv!" => Benchmark(evals=1, seconds=1.0, samples=10000)
"her!" => Benchmark(evals=1, seconds=1.0, samples=10000)
"trsv" => Benchmark(evals=1, seconds=1.0, samples=10000)
"dot" => Benchmark(evals=944, seconds=1.0, samples=10000)
"dotu" => Benchmark(evals=349, seconds=1.0, samples=10000)
"herk!" => Benchmark(evals=1, seconds=1.0, samples=10000)
"trmv" => Benchmark(evals=1, seconds=1.0, samples=10000)
"symv" => Benchmark(evals=1, seconds=1.0, samples=10000)
"dotc" => Benchmark(evals=349, seconds=1.0, samples=10000)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
```

```
  tags: []
  "string" => 2-element BenchmarkGroup([])
  "linalg" => 3-element BenchmarkGroup(["array"])
  "parallel" => 0-element BenchmarkGroup([])
  "tuple" => 1-element BenchmarkGroup([])
  "dates" => 7-element BenchmarkGroup([])
  "micro" => 7-element BenchmarkGroup(["recursion", "fibonacci", "fib", "parse", "parseint", "mandel", "mandelbrot",
"sort", "quicksort", "pi", "π", "sum", "pisum", "πsum", "rand", "randmatstat", "rand", "randmatmul"])
  "io" => 2-element BenchmarkGroup([])
  "scalar" => 5-element BenchmarkGroup([])
  "sparse" => 3-element BenchmarkGroup(["array"])
  "broadcast" => 3-element BenchmarkGroup([])
  "simd" => 72-element BenchmarkGroup(["array", "inbounds"])
  ⋮
```

In [2]:
```
# Let's get a closer look...
showall(suite)
```

```
                        "gbmv!" => Benchmark(evals=1, seconds=1.0, samples=10000)
                        "gbmv" => Benchmark(evals=1, seconds=1.0, samples=10000)
                        "gemm!" => Benchmark(evals=1, seconds=1.0, samples=10000)
                        "gemv!" => Benchmark(evals=1, seconds=1.0, samples=10000)
                        "blascopy!" => Benchmark(evals=864, seconds=1.0, samples=10000)
                        "scal!" => Benchmark(evals=530, seconds=1.0, samples=10000)
                        "gemv" => Benchmark(evals=1, seconds=1.0, samples=10000)
                        "syr!" => Benchmark(evals=1, seconds=1.0, samples=10000)
                        "trsm!" => Benchmark(evals=1, seconds=1.0, samples=10000)
                        "trsv!" => Benchmark(evals=1, seconds=1.0, samples=10000)
                        "her!" => Benchmark(evals=1, seconds=1.0, samples=10000)
                        "trsv" => Benchmark(evals=1, seconds=1.0, samples=10000)
                        "dot" => Benchmark(evals=944, seconds=1.0, samples=10000)
                        "dotu" => Benchmark(evals=349, seconds=1.0, samples=10000)
                        "herk!" => Benchmark(evals=1, seconds=1.0, samples=10000)
                        "trmv" => Benchmark(evals=1, seconds=1.0, samples=10000)
                        "symv" => Benchmark(evals=1, seconds=1.0, samples=10000)
                        "dotc" => Benchmark(evals=349, seconds=1.0, samples=10000)
                        "axpy!" => Benchmark(evals=555, seconds=1.0, samples=10000)
                        "syrk!" => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
```

```
  tags: []
  "string" => 2-element BenchmarkGroup([])
  "linalg" => 3-element BenchmarkGroup(["array"])
  "parallel" => 0-element BenchmarkGroup([])
  "tuple" => 1-element BenchmarkGroup([])
  "dates" => 7-element BenchmarkGroup([])
  "micro" => 7-element BenchmarkGroup(["recursion", "fibonacci", "fib", "parse", "parseint", "mandel", "mandelbrot",
"sort", "quicksort", "pi", "π", "sum", "pisum", "πsum", "rand", "randmatstat", "rand", "randmatmul"])
  "io" => 2-element BenchmarkGroup([])
  "scalar" => 5-element BenchmarkGroup([])
  "sparse" => 3-element BenchmarkGroup(["array"])
  "broadcast" => 3-element BenchmarkGroup([])
  "simd" => 72-element BenchmarkGroup(["array", "inbounds"])
  ⋮
```

In [2]:
```
# Let's get a closer look...
showall(suite)
```

```
"blascopy!" => Benchmark(evals=864, seconds=1.0, samples=10000)
"scal!" => Benchmark(evals=530, seconds=1.0, samples=10000)
"gemv" => Benchmark(evals=1, seconds=1.0, samples=10000)
"syr!" => Benchmark(evals=1, seconds=1.0, samples=10000)
"trsm!" => Benchmark(evals=1, seconds=1.0, samples=10000)
"trsv!" => Benchmark(evals=1, seconds=1.0, samples=10000)
"her!" => Benchmark(evals=1, seconds=1.0, samples=10000)
"trsv" => Benchmark(evals=1, seconds=1.0, samples=10000)
"dot" => Benchmark(evals=944, seconds=1.0, samples=10000)
"dotu" => Benchmark(evals=349, seconds=1.0, samples=10000)
"herk!" => Benchmark(evals=1, seconds=1.0, samples=10000)
"trmv" => Benchmark(evals=1, seconds=1.0, samples=10000)
"symv" => Benchmark(evals=1, seconds=1.0, samples=10000)
"dotc" => Benchmark(evals=349, seconds=1.0, samples=10000)
"axpy!" => Benchmark(evals=555, seconds=1.0, samples=10000)
"syrk!" => Benchmark(evals=1, seconds=1.0, samples=10000)
"sbmv" => Benchmark(evals=1, seconds=1.0, samples=10000)
"sbmv!" => Benchmark(evals=1, seconds=1.0, samples=10000)
"symv!" => Benchmark(evals=1, seconds=1.0, samples=10000)
"symm" => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
```

```
  tags: []
  "string" => 2-element BenchmarkGroup([])
  "linalg" => 3-element BenchmarkGroup(["array"])
  "parallel" => 0-element BenchmarkGroup([])
  "tuple" => 1-element BenchmarkGroup([])
  "dates" => 7-element BenchmarkGroup([])
  "micro" => 7-element BenchmarkGroup(["recursion", "fibonacci", "fib", "parse", "parseint", "mandel", "mandelbrot",
"sort", "quicksort", "pi", "π", "sum", "pisum", "πsum", "rand", "randmatstat", "rand", "randmatmul"])
  "io" => 2-element BenchmarkGroup([])
  "scalar" => 5-element BenchmarkGroup([])
  "sparse" => 3-element BenchmarkGroup(["array"])
  "broadcast" => 3-element BenchmarkGroup([])
  "simd" => 72-element BenchmarkGroup(["array", "inbounds"])
  ⋮
```

In [2]:
```
# Let's get a closer look...
showall(suite)
```

```
                    ("-","Bidiagonal","Bidiagonal",256) => Benchmark(evals=10, seconds=20.0, samples=100)
                    ("*","LowerTriangular","LowerTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("/","Matrix","Matrix",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("\\","UpperTriangular","UpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("\\","SymTridiagonal","Vector",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("-","UpperTriangular","UpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("-","Vector","Vector",1024) => Benchmark(evals=42, seconds=20.0, samples=100)
                    ("\\","Matrix","Matrix",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("+","SymTridiagonal","SymTridiagonal",1024) => Benchmark(evals=10, seconds=20.0, samples=100)
                    ("sqrtm","NPDUpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("A_mul_B!","Matrix{Float32}","Matrix{Float64}","Matrix{Float64}",1024) => Benchmark(evals=1, secon
ds=20.0, samples=100)
                    ("sqrtm","Hermitian",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("\\","Bidiagonal","Vector",1024) => Benchmark(evals=3, seconds=20.0, samples=100)
                    ("\\","UpperTriangular","Vector",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("sqrtm","Base.LinAlg.UnitUpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("*","Tridiagonal","Vector",256) => Benchmark(evals=193, seconds=20.0, samples=100)
                    ("-","Matrix","Matrix",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("-","Tridiagonal","Tridiagonal",1024) => Benchmark(evals=8, seconds=20.0, samples=100)
                    ("\\","Matrix","Vector",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
```

```
  tags: []
  "string" => 2-element BenchmarkGroup([])
  "linalg" => 3-element BenchmarkGroup(["array"])
  "parallel" => 0-element BenchmarkGroup([])
  "tuple" => 1-element BenchmarkGroup([])
  "dates" => 7-element BenchmarkGroup([])
  "micro" => 7-element BenchmarkGroup(["recursion", "fibonacci", "fib", "parse", "parseint", "mandel", "mandelbrot",
"sort", "quicksort", "pi", "π", "sum", "pisum", "πsum", "rand", "randmatstat", "rand", "randmatmul"])
  "io" => 2-element BenchmarkGroup([])
  "scalar" => 5-element BenchmarkGroup([])
  "sparse" => 3-element BenchmarkGroup(["array"])
  "broadcast" => 3-element BenchmarkGroup([])
  "simd" => 72-element BenchmarkGroup(["array", "inbounds"])
  ⋮
```

In [2]:
```
# Let's get a closer look...
showall(suite)
```

```
                        ("-","UpperTriangular","UpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                        ("-","Vector","Vector",1024) => Benchmark(evals=42, seconds=20.0, samples=100)
                        ("\\","Matrix","Matrix",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                        ("+","SymTridiagonal","SymTridiagonal",1024) => Benchmark(evals=10, seconds=20.0, samples=100)
                        ("sqrtm","NPDUpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                        ("A_mul_B!","Matrix{Float32}","Matrix{Float64}","Matrix{Float64}",1024) => Benchmark(evals=1, secon
ds=20.0, samples=100)
                        ("sqrtm","Hermitian",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
                        ("\\","Bidiagonal","Vector",1024) => Benchmark(evals=3, seconds=20.0, samples=100)
                        ("\\","UpperTriangular","Vector",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                        ("sqrtm","Base.LinAlg.UnitUpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                        ("*","Tridiagonal","Vector",256) => Benchmark(evals=193, seconds=20.0, samples=100)
                        ("-","Matrix","Matrix",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
                        ("-","Tridiagonal","Tridiagonal",1024) => Benchmark(evals=8, seconds=20.0, samples=100)
                        ("\\","Matrix","Vector",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
                        ("\\","LowerTriangular","LowerTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
                        ("-","Vector","Vector",256) => Benchmark(evals=232, seconds=20.0, samples=100)
                        ("+","Bidiagonal","Bidiagonal",1024) => Benchmark(evals=7, seconds=20.0, samples=100)
                        ("cumsum!",Int32,1024) => Benchmark(evals=202, seconds=20.0, samples=100)
                        ("cumsum!",Int64,1024) => Benchmark(evals=202, seconds=20.0, samples=100)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
```

```
  tags: []
  "string" => 2-element BenchmarkGroup([])
  "linalg" => 3-element BenchmarkGroup(["array"])
  "parallel" => 0-element BenchmarkGroup([])
  "tuple" => 1-element BenchmarkGroup([])
  "dates" => 7-element BenchmarkGroup([])
  "micro" => 7-element BenchmarkGroup(["recursion", "fibonacci", "fib", "parse", "parseint", "mandel", "mandelbrot",
"sort", "quicksort", "pi", "π", "sum", "pisum", "πsum", "rand", "randmatstat", "rand", "randmatmul"])
  "io" => 2-element BenchmarkGroup([])
  "scalar" => 5-element BenchmarkGroup([])
  "sparse" => 3-element BenchmarkGroup(["array"])
  "broadcast" => 3-element BenchmarkGroup([])
  "simd" => 72-element BenchmarkGroup(["array", "inbounds"])
  ⋮
```

In [2]:
```
# Let's get a closer look...
showall(suite)
```

```
("+","Vector","Vector",1024) => Benchmark(evals=22, seconds=20.0, samples=100)
("\\","Tridiagonal","Vector",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("\\","Diagonal","Diagonal",256) => Benchmark(evals=164, seconds=20.0, samples=100)
("+","SymTridiagonal","SymTridiagonal",256) => Benchmark(evals=86, seconds=20.0, samples=100)
("+","Matrix","Matrix",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("\\","SymTridiagonal","Vector",256) => Benchmark(evals=7, seconds=20.0, samples=100)
("\\","LowerTriangular","Vector",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","LowerTriangular","LowerTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","Tridiagonal","Vector",1024) => Benchmark(evals=10, seconds=20.0, samples=100)
("+","Tridiagonal","Tridiagonal",256) => Benchmark(evals=10, seconds=20.0, samples=100)
("\\","LowerTriangular","Vector",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("+","UpperTriangular","UpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","SymTridiagonal","SymTridiagonal",256) => Benchmark(evals=136, seconds=20.0, samples=100)
("sqrtm","Hermitian",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("sqrtm","NPDUpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","UpperTriangular","UpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("/","LowerTriangular","LowerTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("/","Diagonal","Diagonal",1024) => Benchmark(evals=9, seconds=20.0, samples=100)
("*","Diagonal","Diagonal",256) => Benchmark(evals=228, seconds=20.0, samples=100)
("*","SymTridiagonal","Vector",1024) => Benchmark(evals=10, seconds=20.0, samples=100)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
```

```
  tags: []
  "string" => 2-element BenchmarkGroup([])
  "linalg" => 3-element BenchmarkGroup(["array"])
  "parallel" => 0-element BenchmarkGroup([])
  "tuple" => 1-element BenchmarkGroup([])
  "dates" => 7-element BenchmarkGroup([])
  "micro" => 7-element BenchmarkGroup(["recursion", "fibonacci", "fib", "parse", "parseint", "mandel", "mandelbrot",
"sort", "quicksort", "pi", "π", "sum", "pisum", "πsum", "rand", "randmatstat", "rand", "randmatmul"])
  "io" => 2-element BenchmarkGroup([])
  "scalar" => 5-element BenchmarkGroup([])
  "sparse" => 3-element BenchmarkGroup(["array"])
  "broadcast" => 3-element BenchmarkGroup([])
  "simd" => 72-element BenchmarkGroup(["array", "inbounds"])
  ⋮
```

In [2]:
```
# Let's get a closer look...
showall(suite)
```

```
("\\","LowerTriangular","Vector",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","LowerTriangular","LowerTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","Tridiagonal","Vector",1024) => Benchmark(evals=10, seconds=20.0, samples=100)
("+","Tridiagonal","Tridiagonal",256) => Benchmark(evals=10, seconds=20.0, samples=100)
("\\","LowerTriangular","Vector",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("+","UpperTriangular","UpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","SymTridiagonal","SymTridiagonal",256) => Benchmark(evals=136, seconds=20.0, samples=100)
("sqrtm","Hermitian",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("sqrtm","NPDUpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","UpperTriangular","UpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("/","LowerTriangular","LowerTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("/","Diagonal","Diagonal",1024) => Benchmark(evals=9, seconds=20.0, samples=100)
("*","Diagonal","Diagonal",256) => Benchmark(evals=228, seconds=20.0, samples=100)
("*","SymTridiagonal","Vector",1024) => Benchmark(evals=10, seconds=20.0, samples=100)
("/","Diagonal","Diagonal",256) => Benchmark(evals=162, seconds=20.0, samples=100)
("*","Matrix","Matrix",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","Diagonal","Diagonal",1024) => Benchmark(evals=19, seconds=20.0, samples=100)
("sqrtm","UpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","Tridiagonal","Tridiagonal",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","LowerTriangular","Vector",256) => Benchmark(evals=8, seconds=20.0, samples=100)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
```

```
  tags: []
  "string" => 2-element BenchmarkGroup([])
  "linalg" => 3-element BenchmarkGroup(["array"])
  "parallel" => 0-element BenchmarkGroup([])
  "tuple" => 1-element BenchmarkGroup([])
  "dates" => 7-element BenchmarkGroup([])
  "micro" => 7-element BenchmarkGroup(["recursion", "fibonacci", "fib", "parse", "parseint", "mandel", "mandelbrot",
"sort", "quicksort", "pi", "π", "sum", "pisum", "πsum", "rand", "randmatstat", "rand", "randmatmul"])
  "io" => 2-element BenchmarkGroup([])
  "scalar" => 5-element BenchmarkGroup([])
  "sparse" => 3-element BenchmarkGroup(["array"])
  "broadcast" => 3-element BenchmarkGroup([])
  "simd" => 72-element BenchmarkGroup(["array", "inbounds"])
  ⋮
```

In [2]:
```
# Let's get a closer look...
showall(suite)
```

```
("+","Tridiagonal","Tridiagonal",256) => Benchmark(evals=10, seconds=20.0, samples=100)
("\\","LowerTriangular","Vector",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("+","UpperTriangular","UpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","SymTridiagonal","SymTridiagonal",256) => Benchmark(evals=136, seconds=20.0, samples=100)
("sqrtm","Hermitian",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("sqrtm","NPDUpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","UpperTriangular","UpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("/","LowerTriangular","LowerTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("/","Diagonal","Diagonal",1024) => Benchmark(evals=9, seconds=20.0, samples=100)
("*","Diagonal","Diagonal",256) => Benchmark(evals=228, seconds=20.0, samples=100)
("*","SymTridiagonal","Vector",1024) => Benchmark(evals=10, seconds=20.0, samples=100)
("/","Diagonal","Diagonal",256) => Benchmark(evals=162, seconds=20.0, samples=100)
("*","Matrix","Matrix",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","Diagonal","Diagonal",1024) => Benchmark(evals=19, seconds=20.0, samples=100)
("sqrtm","UpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","Tridiagonal","Tridiagonal",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","LowerTriangular","Vector",256) => Benchmark(evals=8, seconds=20.0, samples=100)
("*","Tridiagonal","Tridiagonal",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("\\","Matrix","Vector",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","Diagonal","Vector",1024) => Benchmark(evals=29, seconds=20.0, samples=100)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
```

```
  tags: []
  "string" => 2-element BenchmarkGroup([])
  "linalg" => 3-element BenchmarkGroup(["array"])
  "parallel" => 0-element BenchmarkGroup([])
  "tuple" => 1-element BenchmarkGroup([])
  "dates" => 7-element BenchmarkGroup([])
  "micro" => 7-element BenchmarkGroup(["recursion", "fibonacci", "fib", "parse", "parseint", "mandel", "mandelbrot",
"sort", "quicksort", "pi", "π", "sum", "pisum", "πsum", "rand", "randmatstat", "rand", "randmatmul"])
  "io" => 2-element BenchmarkGroup([])
  "scalar" => 5-element BenchmarkGroup([])
  "sparse" => 3-element BenchmarkGroup(["array"])
  "broadcast" => 3-element BenchmarkGroup([])
  "simd" => 72-element BenchmarkGroup(["array", "inbounds"])
  ⋮
```

In [2]:
```
# Let's get a closer look...
showall(suite)
```

```
                ("sumelt","TupleWrapper",60,Float64) => Benchmark(evals=993, seconds=1.0, samples=10000)
                ("sumelt","NTuple",3,Float32) => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("sumelt","NTuple",60,Float32) => Benchmark(evals=993, seconds=1.0, samples=10000)
                ("sumelt","NTuple",3,Float64) => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("sumelt","TupleWrapper",3,Float64) => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("sumelt","TupleWrapper",8,Float64) => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("sumelt","TupleWrapper",30,Float64) => Benchmark(evals=999, seconds=1.0, samples=10000)
                ("sumelt","TupleWrapper",60,Float32) => Benchmark(evals=994, seconds=1.0, samples=10000)
                ("sumelt","NTuple",60,Float64) => Benchmark(evals=994, seconds=1.0, samples=10000)
                ("sumelt","TupleWrapper",30,Float32) => Benchmark(evals=999, seconds=1.0, samples=10000)
                ("sumelt","NTuple",8,Float32) => Benchmark(evals=1000, seconds=1.0, samples=10000)
  "dates" => 7-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "string" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  "Date" => Benchmark(evals=394, seconds=1.0, samples=10000)
                  "DateTime" => Benchmark(evals=183, seconds=1.0, samples=10000)
          "construction" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  "Date" => Benchmark(evals=999, seconds=1.0, samples=10000)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
```

```
  tags: []
  "string" => 2-element BenchmarkGroup([])
  "linalg" => 3-element BenchmarkGroup(["array"])
  "parallel" => 0-element BenchmarkGroup([])
  "tuple" => 1-element BenchmarkGroup([])
  "dates" => 7-element BenchmarkGroup([])
  "micro" => 7-element BenchmarkGroup(["recursion", "fibonacci", "fib", "parse", "parseint", "mandel", "mandelbrot",
"sort", "quicksort", "pi", "π", "sum", "pisum", "πsum", "rand", "randmatstat", "rand", "randmatmul"])
  "io" => 2-element BenchmarkGroup([])
  "scalar" => 5-element BenchmarkGroup([])
  "sparse" => 3-element BenchmarkGroup(["array"])
  "broadcast" => 3-element BenchmarkGroup([])
  "simd" => 72-element BenchmarkGroup(["array", "inbounds"])
  ⋮
```

In [2]:
```
# Let's get a closer look...
showall(suite)
```

```
        tags: []
        "Date" => Benchmark(evals=999, seconds=1.0, samples=10000)
        "DateTime" => Benchmark(evals=998, seconds=1.0, samples=10000)
    "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
        tags: []
        ("Date","Month") => Benchmark(evals=989, seconds=1.0, samples=10000)
        ("DateTime","Millisecond") => Benchmark(evals=1000, seconds=1.0, samples=10000)
        ("DateTime","Month") => Benchmark(evals=979, seconds=1.0, samples=10000)
        ("Date","Year") => Benchmark(evals=995, seconds=1.0, samples=10000)
        ("DateTime","Second") => Benchmark(evals=1000, seconds=1.0, samples=10000)
        ("Date","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
        ("DateTime","Year") => Benchmark(evals=987, seconds=1.0, samples=10000)
        ("DateTime","Hour") => Benchmark(evals=1000, seconds=1.0, samples=10000)
        ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
        ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
    "query" => 18-element BenchmarkTools.BenchmarkGroup:
        tags: []
        ("lastdayofweek","Date") => Benchmark(evals=999, seconds=1.0, samples=10000)
        ("firstdayofmonth","DateTime") => Benchmark(evals=997, seconds=1.0, samples=10000)
        ("dayofyear","Date") => Benchmark(evals=997, seconds=1.0, samples=10000)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
```

```
micro => 7-element BenchmarkGroup([ recursion , fibonacci , fib , parse , parseint , mandel , mandelbrot ,
"sort", "quicksort", "pi", "π", "sum", "pisum", "πsum", "rand", "randmatstat", "rand", "randmatmul"])
 "io" => 2-element BenchmarkGroup([])
 "scalar" => 5-element BenchmarkGroup([])
 "sparse" => 3-element BenchmarkGroup(["array"])
 "broadcast" => 3-element BenchmarkGroup([])
 "simd" => 72-element BenchmarkGroup(["array", "inbounds"])
 ⋮
```

In [2]:
```
# Let's get a closer look...
showall(suite)
```

```
        tags: []
        "Date" => Benchmark(evals=999, seconds=1.0, samples=10000)
        "DateTime" => Benchmark(evals=998, seconds=1.0, samples=10000)
    "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
        tags: []
        ("Date","Month") => Benchmark(evals=989, seconds=1.0, samples=10000)
        ("DateTime","Millisecond") => Benchmark(evals=1000, seconds=1.0, samples=10000)
        ("DateTime","Month") => Benchmark(evals=979, seconds=1.0, samples=10000)
        ("Date","Year") => Benchmark(evals=995, seconds=1.0, samples=10000)
        ("DateTime","Second") => Benchmark(evals=1000, seconds=1.0, samples=10000)
        ("Date","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
        ("DateTime","Year") => Benchmark(evals=987, seconds=1.0, samples=10000)
        ("DateTime","Hour") => Benchmark(evals=1000, seconds=1.0, samples=10000)
        ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
        ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
    "query" => 18-element BenchmarkTools.BenchmarkGroup:
        tags: []
        ("lastdayofweek","Date") => Benchmark(evals=999, seconds=1.0, samples=10000)
        ("firstdayofmonth","DateTime") => Benchmark(evals=997, seconds=1.0, samples=10000)
        ("dayofyear","Date") => Benchmark(evals=997, seconds=1.0, samples=10000)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [2]:
```
# Let's get a closer look...
showall(suite)
```

```
                tags: []
                "Date" => Benchmark(evals=999, seconds=1.0, samples=10000)
                "DateTime" => Benchmark(evals=998, seconds=1.0, samples=10000)
        "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("Date","Month") => Benchmark(evals=989, seconds=1.0, samples=10000)
                ("DateTime","Millisecond") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Month") => Benchmark(evals=979, seconds=1.0, samples=10000)
                ("Date","Year") => Benchmark(evals=995, seconds=1.0, samples=10000)
                ("DateTime","Second") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("Date","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Year") => Benchmark(evals=987, seconds=1.0, samples=10000)
                ("DateTime","Hour") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
        "query" => 18-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("lastdayofweek","Date") => Benchmark(evals=999, seconds=1.0, samples=10000)
                ("firstdayofmonth","DateTime") => Benchmark(evals=997, seconds=1.0, samples=10000)
                ("dayofyear","Date") => Benchmark(evals=997, seconds=1.0, samples=10000)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
```

File Edit View Insert Cell Kernel Widgets Help Julia 0.5.1-pre

In [2]:
```
# Let's get a closer look...
showall(suite)
```

```
                tags: []
                "Date" => Benchmark(evals=999, seconds=1.0, samples=10000)
                "DateTime" => Benchmark(evals=998, seconds=1.0, samples=10000)
        "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("Date","Month") => Benchmark(evals=989, seconds=1.0, samples=10000)
                ("DateTime","Millisecond") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Month") => Benchmark(evals=979, seconds=1.0, samples=10000)
                ("Date","Year") => Benchmark(evals=995, seconds=1.0, samples=10000)
                ("DateTime","Second") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("Date","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Year") => Benchmark(evals=987, seconds=1.0, samples=10000)
                ("DateTime","Hour") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
        "query" => 18-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("lastdayofweek","Date") => Benchmark(evals=999, seconds=1.0, samples=10000)
                ("firstdayofmonth","DateTime") => Benchmark(evals=997, seconds=1.0, samples=10000)
                ("dayofyear","Date") => Benchmark(evals=997, seconds=1.0, samples=10000)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
```

```
                tags: []
                ("Date","Month") => Benchmark(evals=989, seconds=1.0, samples=10000)
                ("DateTime","Millisecond") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Month") => Benchmark(evals=979, seconds=1.0, samples=10000)
                ("Date","Year") => Benchmark(evals=995, seconds=1.0, samples=10000)
                ("DateTime","Second") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("Date","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Year") => Benchmark(evals=987, seconds=1.0, samples=10000)
                ("DateTime","Hour") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
  "query" => 18-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("lastdayofweek","Date") => Benchmark(evals=999, seconds=1.0, samples=10000)
                ("firstdayofmonth","DateTime") => Benchmark(evals=997, seconds=1.0, samples=10000)
                ("dayofyear","Date") => Benchmark(evals=997, seconds=1.0, samples=10000)
```

In [ ]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

In [ ]:
```
vs = judge(minimum(r6), minimum(r5), time_tolerance = 0.05)
showall(vs)
```

In [ ]:
```
showall(regressions(vs))
```

```
                    ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                    ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                "query" => 18-element BenchmarkTools.BenchmarkGroup:
                    tags: []
                    ("lastdayofweek","Date") => Benchmark(evals=999, seconds=1.0, samples=10000)
                    ("firstdayofmonth","DateTime") => Benchmark(evals=997, seconds=1.0, samples=10000)
                    ("dayofyear","Date") => Benchmark(evals=997, seconds=1.0, samples=10000)
```

In [*]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

In [ ]:
```
vs = judge(minimum(r6), minimum(r5), time_tolerance = 0.05)
showall(vs)
```

In [ ]:
```
showall(regressions(vs))
```

In [ ]:
```
showall(improvements(vs))
```

In [ ]:
```
id = ["io", "serialization", ("serialize","Matrix{Float64}")]
```

In [ ]:
```
t5 = r5[id]
```

In [ ]:
```
t6 = r6[id]
```

```
                    ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                    ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                "query" => 18-element BenchmarkTools.BenchmarkGroup:
                        tags: []
                        ("lastdayofweek","Date") => Benchmark(evals=999, seconds=1.0, samples=10000)
                        ("firstdayofmonth","DateTime") => Benchmark(evals=997, seconds=1.0, samples=10000)
                        ("dayofyear","Date") => Benchmark(evals=997, seconds=1.0, samples=10000)
```

In [3]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

```
7-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "join" => Benchmark(evals=1, seconds=1.0, samples=10000)
          "replace" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("deserialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("deserialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("serialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
          "read" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "stream", "string"]
                  "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                  "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
  "linalg" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["array"]
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

```
                    ("firstdayofmonth","DateTime") => Benchmark(evals=997, seconds=1.0, samples=10000)
                    ("dayofyear","Date") => Benchmark(evals=997, seconds=1.0, samples=10000)
```

In [3]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

```
7-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "join" => Benchmark(evals=1, seconds=1.0, samples=10000)
          "replace" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("deserialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("deserialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("serialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
          "read" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "stream", "string"]
                  "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                  "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
  "linalg" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["array"]
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

```
                ("firstdayofmonth","DateTime") => Benchmark(evals=997, seconds=1.0, samples=10000)
                ("dayofyear","Date") => Benchmark(evals=997, seconds=1.0, samples=10000)
```

In [3]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

```
                ("deserialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("deserialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("serialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
        "read" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "stream", "string"]
                "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
  "linalg" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["array"]
        "arithmetic" => 108-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("cumsum!",Int32,256) => Benchmark(evals=934, seconds=20.0, samples=100)
                ("\\","Bidiagonal","Vector",256) => Benchmark(evals=9, seconds=20.0, samples=100)
                ("*","Diagonal","Diagonal",1024) => Benchmark(evals=34, seconds=20.0, samples=100)
                ("+","Tridiagonal","Tridiagonal",1024) => Benchmark(evals=8, seconds=20.0, samples=100)
                ("-","Matrix","Matrix",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                ("cumsum!",Float32,1024) => Benchmark(evals=10, seconds=20.0, samples=100)
                ("\\","Diagonal","Vector",256) => Benchmark(evals=166, seconds=20.0, samples=100)
                ("+","Matrix","Matrix",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

```
                ("firstdayofmonth","DateTime") => Benchmark(evals=997, seconds=1.0, samples=10000)
                ("dayofyear","Date") => Benchmark(evals=997, seconds=1.0, samples=10000)
```

In [3]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

```
                "join" => Benchmark(evals=1, seconds=1.0, samples=10000)
                "replace" => Benchmark(evals=1, seconds=1.0, samples=10000)
        "io" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                        tags: ["buffer", "string"]
                        ("deserialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                        ("deserialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                        ("serialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                        ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                "read" => 2-element BenchmarkTools.BenchmarkGroup:
                        tags: ["buffer", "stream", "string"]
                        "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                        "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
        "linalg" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["array"]
                "arithmetic" => 108-element BenchmarkTools.BenchmarkGroup:
                        tags: []
                        ("cumsum!",Int32,256) => Benchmark(evals=934, seconds=20.0, samples=100)
                        ("\\","Bidiagonal","Vector",256) => Benchmark(evals=9, seconds=20.0, samples=100)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

```
                ("firstdayofmonth","DateTime") => Benchmark(evals=997, seconds=1.0, samples=10000)
                ("dayofyear","Date") => Benchmark(evals=997, seconds=1.0, samples=10000)
```

In [3]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

```
        tags: []
        "join" => Benchmark(evals=1, seconds=1.0, samples=10000)
        "replace" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "string"]
                ("deserialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("deserialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("serialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
        "read" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "stream", "string"]
                "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
  "linalg" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["array"]
        "arithmetic" => 108-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("cumsum!",Int32,256) => Benchmark(evals=934, seconds=20.0, samples=100)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

```
                ("firstdayofmonth","DateTime") => Benchmark(evals=997, seconds=1.0, samples=10000)
                ("dayofyear","Date") => Benchmark(evals=997, seconds=1.0, samples=10000)
```

In [3]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

```
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "join" => Benchmark(evals=1, seconds=1.0, samples=10000)
        "replace" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "string"]
                ("deserialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("deserialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("serialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
        "read" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "stream", "string"]
                "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
  "linalg" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["array"]
        "arithmetic" => 108-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("cumsum!",Int32,256) => Benchmark(evals=934, seconds=20.0, samples=100)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

```
                    tags: []
                    ("lastdayofweek","Date") => Benchmark(evals=999, seconds=1.0, samples=10000)
                    ("firstdayofmonth","DateTime") => Benchmark(evals=997, seconds=1.0, samples=10000)
                    ("dayofyear","Date") => Benchmark(evals=997, seconds=1.0, samples=10000)
```

In [3]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

```
7-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "join" => Benchmark(evals=1, seconds=1.0, samples=10000)
          "replace" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("deserialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("deserialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("serialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
          "read" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "stream", "string"]
                  "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                  "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
  "linalg" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["array"]
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
```

```
                tags: []
                ("lastdayofweek","Date") => Benchmark(evals=999, seconds=1.0, samples=10000)
                ("firstdayofmonth","DateTime") => Benchmark(evals=997, seconds=1.0, samples=10000)
                ("dayofyear","Date") => Benchmark(evals=997, seconds=1.0, samples=10000)
```

In [3]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

```
                ("deserialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("deserialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("serialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
        "read" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "stream", "string"]
                "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
  "linalg" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["array"]
          "arithmetic" => 108-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("cumsum!",Int32,256) => Benchmark(evals=934, seconds=20.0, samples=100)
                ("\\","Bidiagonal","Vector",256) => Benchmark(evals=9, seconds=20.0, samples=100)
                ("*","Diagonal","Diagonal",1024) => Benchmark(evals=34, seconds=20.0, samples=100)
                ("+","Tridiagonal","Tridiagonal",1024) => Benchmark(evals=8, seconds=20.0, samples=100)
                ("-","Matrix","Matrix",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                ("cumsum!",Float32,1024) => Benchmark(evals=10, seconds=20.0, samples=100)
                ("\\","Diagonal","Vector",256) => Benchmark(evals=166, seconds=20.0, samples=100)
                ("+","Matrix","Matrix",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
```

```
                        tags: []
                        ("lastdayofweek","Date") => Benchmark(evals=999, seconds=1.0, samples=10000)
                        ("firstdayofmonth","DateTime") => Benchmark(evals=997, seconds=1.0, samples=10000)
                        ("dayofyear","Date") => Benchmark(evals=997, seconds=1.0, samples=10000)
```

In [3]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

```
                        ("\\","Bidiagonal","Vector",256) => Benchmark(evals=9, seconds=20.0, samples=100)
                        ("*","Diagonal","Diagonal",1024) => Benchmark(evals=34, seconds=20.0, samples=100)
                        ("+","Tridiagonal","Tridiagonal",1024) => Benchmark(evals=8, seconds=20.0, samples=100)
                        ("-","Matrix","Matrix",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                        ("cumsum!",Float32,1024) => Benchmark(evals=10, seconds=20.0, samples=100)
                        ("\\","Diagonal","Vector",256) => Benchmark(evals=166, seconds=20.0, samples=100)
                        ("+","Matrix","Matrix",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
                        ("-","LowerTriangular","LowerTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
                        ("A_mul_B!","Matrix{Float32}","Matrix{Float64}","Matrix{Float64}",256) => Benchmark(evals=1, second
s=20.0, samples=100)
                        ("*","SymTridiagonal","SymTridiagonal",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
                        ("sqrtm","UpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                        ("/","UpperTriangular","UpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
                        ("*","SymTridiagonal","Vector",256) => Benchmark(evals=188, seconds=20.0, samples=100)
                        ("/","UpperTriangular","UpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                        ("*","UpperTriangular","Vector",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
                        ("*","UpperTriangular","Vector",256) => Benchmark(evals=8, seconds=20.0, samples=100)
                        ("-","UpperTriangular","UpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
                        ("*","Matrix","Vector",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
                        ("-","Tridiagonal","Tridiagonal",256) => Benchmark(evals=10, seconds=20.0, samples=100)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
```

```
tags: []
("lastdayofweek","Date") => Benchmark(evals=999, seconds=1.0, samples=10000)
("firstdayofmonth","DateTime") => Benchmark(evals=997, seconds=1.0, samples=10000)
("dayofyear","Date") => Benchmark(evals=997, seconds=1.0, samples=10000)
```

In [3]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

```
("*","SymTridiagonal","SymTridiagonal",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("sqrtm","UpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("/","UpperTriangular","UpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","SymTridiagonal","Vector",256) => Benchmark(evals=188, seconds=20.0, samples=100)
("/","UpperTriangular","UpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","UpperTriangular","Vector",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","UpperTriangular","Vector",256) => Benchmark(evals=8, seconds=20.0, samples=100)
("-","UpperTriangular","UpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","Matrix","Vector",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","Tridiagonal","Tridiagonal",256) => Benchmark(evals=10, seconds=20.0, samples=100)
("-","SymTridiagonal","SymTridiagonal",1024) => Benchmark(evals=10, seconds=20.0, samples=100)
("*","LowerTriangular","Vector",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","SymTridiagonal","SymTridiagonal",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("+","Vector","Vector",256) => Benchmark(evals=234, seconds=20.0, samples=100)
("+","LowerTriangular","LowerTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","Bidiagonal","Bidiagonal",1024) => Benchmark(evals=8, seconds=20.0, samples=100)
("+","UpperTriangular","UpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("/","Matrix","Matrix",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","Bidiagonal","Vector",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","Bidiagonal","Bidiagonal",256) => Benchmark(evals=10, seconds=20.0, samples=100)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
```

```
tags: []
("lastdayofweek","Date") => Benchmark(evals=999, seconds=1.0, samples=10000)
("firstdayofmonth","DateTime") => Benchmark(evals=997, seconds=1.0, samples=10000)
("dayofyear","Date") => Benchmark(evals=997, seconds=1.0, samples=10000)
```

In [3]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

```
("*","SymTridiagonal","SymTridiagonal",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("sqrtm","UpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("/","UpperTriangular","UpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","SymTridiagonal","Vector",256) => Benchmark(evals=188, seconds=20.0, samples=100)
("/","UpperTriangular","UpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","UpperTriangular","Vector",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","UpperTriangular","Vector",256) => Benchmark(evals=8, seconds=20.0, samples=100)
("-","UpperTriangular","UpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","Matrix","Vector",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","Tridiagonal","Tridiagonal",256) => Benchmark(evals=10, seconds=20.0, samples=100)
("-","SymTridiagonal","SymTridiagonal",1024) => Benchmark(evals=10, seconds=20.0, samples=100)
("*","LowerTriangular","Vector",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","SymTridiagonal","SymTridiagonal",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("+","Vector","Vector",256) => Benchmark(evals=234, seconds=20.0, samples=100)
("+","LowerTriangular","LowerTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","Bidiagonal","Bidiagonal",1024) => Benchmark(evals=8, seconds=20.0, samples=100)
("+","UpperTriangular","UpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("/","Matrix","Matrix",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","Bidiagonal","Vector",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","Bidiagonal","Bidiagonal",256) => Benchmark(evals=10, seconds=20.0, samples=100)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
```

```
                    tags: []
                    ("lastdayofweek","Date") => Benchmark(evals=999, seconds=1.0, samples=10000)
                    ("firstdayofmonth","DateTime") => Benchmark(evals=997, seconds=1.0, samples=10000)
                    ("dayofyear","Date") => Benchmark(evals=997, seconds=1.0, samples=10000)
```

In [3]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

```
                    ("+","LowerTriangular","LowerTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("-","Bidiagonal","Bidiagonal",1024) => Benchmark(evals=8, seconds=20.0, samples=100)
                    ("+","UpperTriangular","UpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("/","Matrix","Matrix",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("*","Bidiagonal","Vector",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("-","Bidiagonal","Bidiagonal",256) => Benchmark(evals=10, seconds=20.0, samples=100)
                    ("*","LowerTriangular","LowerTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("/","Matrix","Matrix",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("\\","UpperTriangular","UpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("\\","SymTridiagonal","Vector",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("-","UpperTriangular","UpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("-","Vector","Vector",1024) => Benchmark(evals=42, seconds=20.0, samples=100)
                    ("\\","Matrix","Matrix",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("+","SymTridiagonal","SymTridiagonal",1024) => Benchmark(evals=10, seconds=20.0, samples=100)
                    ("sqrtm","NPDUpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("A_mul_B!","Matrix{Float32}","Matrix{Float64}","Matrix{Float64}",1024) => Benchmark(evals=1, secon
ds=20.0, samples=100)
                    ("sqrtm","Hermitian",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
                    ("\\","Bidiagonal","Vector",1024) => Benchmark(evals=3, seconds=20.0, samples=100)
                    ("\\","UpperTriangular","Vector",256) => Benchmark(evals=1, seconds=20.0, samples=100)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
```

```
tags: []
("lastdayofweek","Date") => Benchmark(evals=999, seconds=1.0, samples=10000)
("firstdayofmonth","DateTime") => Benchmark(evals=997, seconds=1.0, samples=10000)
("dayofyear","Date") => Benchmark(evals=997, seconds=1.0, samples=10000)
```

In [3]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

```
("-","Vector","Vector",1024) => Benchmark(evals=42, seconds=20.0, samples=100)
("\\","Matrix","Matrix",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("+","SymTridiagonal","SymTridiagonal",1024) => Benchmark(evals=10, seconds=20.0, samples=100)
("sqrtm","NPDUpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("A_mul_B!","Matrix{Float32}","Matrix{Float64}","Matrix{Float64}",1024) => Benchmark(evals=1, secon
ds=20.0, samples=100)
("sqrtm","Hermitian",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("\\","Bidiagonal","Vector",1024) => Benchmark(evals=3, seconds=20.0, samples=100)
("\\","UpperTriangular","Vector",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("sqrtm","Base.LinAlg.UnitUpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","Tridiagonal","Vector",256) => Benchmark(evals=193, seconds=20.0, samples=100)
("-","Matrix","Matrix",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","Tridiagonal","Tridiagonal",1024) => Benchmark(evals=8, seconds=20.0, samples=100)
("\\","Matrix","Vector",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("\\","LowerTriangular","LowerTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","Vector","Vector",256) => Benchmark(evals=232, seconds=20.0, samples=100)
("+","Bidiagonal","Bidiagonal",1024) => Benchmark(evals=7, seconds=20.0, samples=100)
("cumsum!",Int32,1024) => Benchmark(evals=202, seconds=20.0, samples=100)
("cumsum!",Int64,1024) => Benchmark(evals=202, seconds=20.0, samples=100)
("\\","Matrix","Matrix",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
```

```
tags: []
("lastdayofweek","Date") => Benchmark(evals=999, seconds=1.0, samples=10000)
("firstdayofmonth","DateTime") => Benchmark(evals=997, seconds=1.0, samples=10000)
("dayofyear","Date") => Benchmark(evals=997, seconds=1.0, samples=10000)
```

In [3]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

```
("\\","Matrix","Vector",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("\\","LowerTriangular","LowerTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","Vector","Vector",256) => Benchmark(evals=232, seconds=20.0, samples=100)
("+","Bidiagonal","Bidiagonal",1024) => Benchmark(evals=7, seconds=20.0, samples=100)
("cumsum!",Int32,1024) => Benchmark(evals=202, seconds=20.0, samples=100)
("cumsum!",Int64,1024) => Benchmark(evals=202, seconds=20.0, samples=100)
("\\","Matrix","Matrix",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("+","Vector","Vector",1024) => Benchmark(evals=22, seconds=20.0, samples=100)
("\\","Tridiagonal","Vector",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("\\","Diagonal","Diagonal",256) => Benchmark(evals=164, seconds=20.0, samples=100)
("+","SymTridiagonal","SymTridiagonal",256) => Benchmark(evals=86, seconds=20.0, samples=100)
("+","Matrix","Matrix",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("\\","SymTridiagonal","Vector",256) => Benchmark(evals=7, seconds=20.0, samples=100)
("\\","LowerTriangular","Vector",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","LowerTriangular","LowerTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","Tridiagonal","Vector",1024) => Benchmark(evals=10, seconds=20.0, samples=100)
("+","Tridiagonal","Tridiagonal",256) => Benchmark(evals=10, seconds=20.0, samples=100)
("\\","LowerTriangular","Vector",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("+","UpperTriangular","UpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","SymTridiagonal","SymTridiagonal",256) => Benchmark(evals=136, seconds=20.0, samples=100)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
```

```
tags: []
("lastdayofweek","Date") => Benchmark(evals=999, seconds=1.0, samples=10000)
("firstdayofmonth","DateTime") => Benchmark(evals=997, seconds=1.0, samples=10000)
("dayofyear","Date") => Benchmark(evals=997, seconds=1.0, samples=10000)
```

In [3]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

```
("*","Tridiagonal","Vector",1024) => Benchmark(evals=10, seconds=20.0, samples=100)
("+","Tridiagonal","Tridiagonal",256) => Benchmark(evals=10, seconds=20.0, samples=100)
("\\","LowerTriangular","Vector",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("+","UpperTriangular","UpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","SymTridiagonal","SymTridiagonal",256) => Benchmark(evals=136, seconds=20.0, samples=100)
("sqrtm","Hermitian",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("sqrtm","NPDUpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","UpperTriangular","UpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("/","LowerTriangular","LowerTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("/","Diagonal","Diagonal",1024) => Benchmark(evals=9, seconds=20.0, samples=100)
("*","Diagonal","Diagonal",256) => Benchmark(evals=228, seconds=20.0, samples=100)
("*","SymTridiagonal","Vector",1024) => Benchmark(evals=10, seconds=20.0, samples=100)
("/","Diagonal","Diagonal",256) => Benchmark(evals=162, seconds=20.0, samples=100)
("*","Matrix","Matrix",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","Diagonal","Diagonal",1024) => Benchmark(evals=19, seconds=20.0, samples=100)
("sqrtm","UpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","Tridiagonal","Tridiagonal",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","LowerTriangular","Vector",256) => Benchmark(evals=8, seconds=20.0, samples=100)
("*","Tridiagonal","Tridiagonal",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
```

File Edit View Insert Cell Kernel Widgets Help Julia 0.5.1-pre

```
("dayofyear","Date") => Benchmark(evals=997, seconds=1.0, samples=10000)
```

In [3]:
```
# That's a lot of benchmarks. Let's filter them:
group = suite[@tagged "arithmetic" || "string"]
showall(group)
```

```
("*","Tridiagonal","Vector",1024) => Benchmark(evals=10, seconds=20.0, samples=100)
("+","Tridiagonal","Tridiagonal",256) => Benchmark(evals=10, seconds=20.0, samples=100)
("\\","LowerTriangular","Vector",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("+","UpperTriangular","UpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","SymTridiagonal","SymTridiagonal",256) => Benchmark(evals=136, seconds=20.0, samples=100)
("sqrtm","Hermitian",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("sqrtm","NPDUpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","UpperTriangular","UpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("/","LowerTriangular","LowerTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("/","Diagonal","Diagonal",1024) => Benchmark(evals=9, seconds=20.0, samples=100)
("*","Diagonal","Diagonal",256) => Benchmark(evals=228, seconds=20.0, samples=100)
("*","SymTridiagonal","Vector",1024) => Benchmark(evals=10, seconds=20.0, samples=100)
("/","Diagonal","Diagonal",256) => Benchmark(evals=162, seconds=20.0, samples=100)
("*","Matrix","Matrix",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","Diagonal","Diagonal",1024) => Benchmark(evals=19, seconds=20.0, samples=100)
("sqrtm","UpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","Tridiagonal","Tridiagonal",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","LowerTriangular","Vector",256) => Benchmark(evals=8, seconds=20.0, samples=100)
("*","Tridiagonal","Tridiagonal",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okáy, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

In [ ]:
```
vs = judge(minimum(r6), minimum(r5), time_tolerance = 0.05)
```

```
("+","Tridiagonal","Tridiagonal",256) => Benchmark(evals=10, seconds=20.0, samples=100)
("\\","LowerTriangular","Vector",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("+","UpperTriangular","UpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","SymTridiagonal","SymTridiagonal",256) => Benchmark(evals=136, seconds=20.0, samples=100)
("sqrtm","Hermitian",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("sqrtm","NPDUpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","UpperTriangular","UpperTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("/","LowerTriangular","LowerTriangular",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("/","Diagonal","Diagonal",1024) => Benchmark(evals=9, seconds=20.0, samples=100)
("*","Diagonal","Diagonal",256) => Benchmark(evals=228, seconds=20.0, samples=100)
("*","SymTridiagonal","Vector",1024) => Benchmark(evals=10, seconds=20.0, samples=100)
("/","Diagonal","Diagonal",256) => Benchmark(evals=162, seconds=20.0, samples=100)
("*","Matrix","Matrix",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","Diagonal","Diagonal",1024) => Benchmark(evals=19, seconds=20.0, samples=100)
("sqrtm","UpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","Tridiagonal","Tridiagonal",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","LowerTriangular","Vector",256) => Benchmark(evals=8, seconds=20.0, samples=100)
("*","Tridiagonal","Tridiagonal",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("\\","Matrix","Vector",256) => Benchmark(evals=1, seconds=20.0, samples=100)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

In [ ]:
```
vs = judge(minimum(r6), minimum(r5), time_tolerance = 0.05)
showall(vs)
```

In [ ]:
```
showall(regressions(vs))
```

In [ ]:
```
showall(improvements(vs))
```

```
("*","Matrix","Matrix",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","Diagonal","Diagonal",1024) => Benchmark(evals=19, seconds=20.0, samples=100)
("sqrtm","UpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","Tridiagonal","Tridiagonal",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","LowerTriangular","Vector",256) => Benchmark(evals=8, seconds=20.0, samples=100)
("*","Tridiagonal","Tridiagonal",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("\\","Matrix","Vector",256) => Benchmark(evals=1, seconds=20.0, samples=100)
```

In [ ]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

In [ ]:
```
vs = judge(minimum(r6), minimum(r5), time_tolerance = 0.05)
showall(vs)
```

In [ ]:
```
showall(regressions(vs))
```

In [ ]:
```
showall(improvements(vs))
```

In [ ]:
```
id = ["io", "serialization", ("serialize","Matrix{Float64}")]
```

In [ ]:
```
t5 = r5[id]
```

In [ ]:
```
t6 = r6[id]
```

In [ ]:
```
plotkde([t6, t5])
```

In [ ]:

```
("*","Matrix","Matrix",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","Diagonal","Diagonal",1024) => Benchmark(evals=19, seconds=20.0, samples=100)
("sqrtm","UpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","Tridiagonal","Tridiagonal",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","LowerTriangular","Vector",256) => Benchmark(evals=8, seconds=20.0, samples=100)
("*","Tridiagonal","Tridiagonal",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("\\","Matrix","Vector",256) => Benchmark(evals=1, seconds=20.0, samples=100)
```

In [4]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

```
5-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "join" => Benchmark(evals=1, seconds=1.0, samples=10000)
          "replace" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("deserialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("deserialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("serialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
          "read" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "stream", "string"]
                  "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                  "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
  "problem" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["example", "kernel"]
          "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["spell", "string"]
                  "spellcheck" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "dates" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "string" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  "Date" => Benchmark(evals=394, seconds=1.0, samples=10000)
                  "DateTime" => Benchmark(evals=183, seconds=1.0, samples=10000)
          "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
```

```
("*","Matrix","Matrix",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("-","Diagonal","Diagonal",1024) => Benchmark(evals=19, seconds=20.0, samples=100)
("sqrtm","UpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","Tridiagonal","Tridiagonal",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","LowerTriangular","Vector",256) => Benchmark(evals=8, seconds=20.0, samples=100)
("*","Tridiagonal","Tridiagonal",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("\\","Matrix","Vector",256) => Benchmark(evals=1, seconds=20.0, samples=100)
```

In [4]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

```
5-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "join" => Benchmark(evals=1, seconds=1.0, samples=10000)
          "replace" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("deserialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("deserialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("serialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
          "read" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "stream", "string"]
                  "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                  "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
  "problem" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["example", "kernel"]
          "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["spell", "string"]
                  "spellcheck" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "dates" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "string" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  "Date" => Benchmark(evals=394, seconds=1.0, samples=10000)
                  "DateTime" => Benchmark(evals=183, seconds=1.0, samples=10000)
          "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
```

```
showall(group)

5-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "join" => Benchmark(evals=1, seconds=1.0, samples=10000)
          "replace" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("deserialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("deserialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("serialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
          "read" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "stream", "string"]
                  "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                  "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
  "problem" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["example", "kernel"]
          "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["spell", "string"]
                  "spellcheck" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "dates" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "string" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  "Date" => Benchmark(evals=394, seconds=1.0, samples=10000)
                  "DateTime" => Benchmark(evals=183, seconds=1.0, samples=10000)
          "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  ("Date","Month") => Benchmark(evals=989, seconds=1.0, samples=10000)
                  ("DateTime","Millisecond") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                  ("DateTime","Month") => Benchmark(evals=979, seconds=1.0, samples=10000)
                  ("Date","Year") => Benchmark(evals=995, seconds=1.0, samples=10000)
                  ("DateTime","Second") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                  ("Date","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                  ("DateTime","Year") => Benchmark(evals=987, seconds=1.0, samples=10000)
                  ("DateTime","Hour") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                  ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
```

```
                ("deserialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("deserialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("serialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
        "read" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "stream", "string"]
                "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
"problem" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["example", "kernel"]
        "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["spell", "string"]
                "spellcheck" => Benchmark(evals=1, seconds=1.0, samples=10000)
"dates" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "string" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                "Date" => Benchmark(evals=394, seconds=1.0, samples=10000)
                "DateTime" => Benchmark(evals=183, seconds=1.0, samples=10000)
        "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("Date","Month") => Benchmark(evals=989, seconds=1.0, samples=10000)
                ("DateTime","Millisecond") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Month") => Benchmark(evals=979, seconds=1.0, samples=10000)
                ("Date","Year") => Benchmark(evals=995, seconds=1.0, samples=10000)
                ("DateTime","Second") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("Date","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Year") => Benchmark(evals=987, seconds=1.0, samples=10000)
                ("DateTime","Hour") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
"sparse" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["array"]
        "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("unary minus",(600,600)) => Benchmark(evals=5, seconds=1.0, samples=10000)
                ("unary minus",(20000,20000)) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

```
                ("deserialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("deserialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("serialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
        "read" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "stream", "string"]
                "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
"problem" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["example", "kernel"]
        "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["spell", "string"]
                "spellcheck" => Benchmark(evals=1, seconds=1.0, samples=10000)
"dates" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "string" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                "Date" => Benchmark(evals=394, seconds=1.0, samples=10000)
                "DateTime" => Benchmark(evals=183, seconds=1.0, samples=10000)
        "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("Date","Month") => Benchmark(evals=989, seconds=1.0, samples=10000)
                ("DateTime","Millisecond") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Month") => Benchmark(evals=979, seconds=1.0, samples=10000)
                ("Date","Year") => Benchmark(evals=995, seconds=1.0, samples=10000)
                ("DateTime","Second") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("Date","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Year") => Benchmark(evals=987, seconds=1.0, samples=10000)
                ("DateTime","Hour") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
"sparse" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["array"]
        "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("unary minus",(600,600)) => Benchmark(evals=5, seconds=1.0, samples=10000)
                ("unary minus",(20000,20000)) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

```
                ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
        "read" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "stream", "string"]
                "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
"problem" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["example", "kernel"]
        "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["spell", "string"]
                "spellcheck" => Benchmark(evals=1, seconds=1.0, samples=10000)
"dates" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "string" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                "Date" => Benchmark(evals=394, seconds=1.0, samples=10000)
                "DateTime" => Benchmark(evals=183, seconds=1.0, samples=10000)
        "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("Date","Month") => Benchmark(evals=989, seconds=1.0, samples=10000)
                ("DateTime","Millisecond") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Month") => Benchmark(evals=979, seconds=1.0, samples=10000)
                ("Date","Year") => Benchmark(evals=995, seconds=1.0, samples=10000)
                ("DateTime","Second") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("Date","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Year") => Benchmark(evals=987, seconds=1.0, samples=10000)
                ("DateTime","Hour") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
"sparse" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["array"]
        "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("unary minus",(600,600)) => Benchmark(evals=5, seconds=1.0, samples=10000)
                ("unary minus",(20000,20000)) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
```

```
                tags: ["buffer", "string"]
                ("deserialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("deserialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("serialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
        "read" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "stream", "string"]
                "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
"problem" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["example", "kernel"]
        "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["spell", "string"]
                "spellcheck" => Benchmark(evals=1, seconds=1.0, samples=10000)
"dates" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "string" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                "Date" => Benchmark(evals=394, seconds=1.0, samples=10000)
                "DateTime" => Benchmark(evals=183, seconds=1.0, samples=10000)
        "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("Date","Month") => Benchmark(evals=989, seconds=1.0, samples=10000)
                ("DateTime","Millisecond") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Month") => Benchmark(evals=979, seconds=1.0, samples=10000)
                ("Date","Year") => Benchmark(evals=995, seconds=1.0, samples=10000)
                ("DateTime","Second") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("Date","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Year") => Benchmark(evals=987, seconds=1.0, samples=10000)
                ("DateTime","Hour") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
"sparse" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["array"]
        "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("unary minus",(600,600)) => Benchmark(evals=5, seconds=1.0, samples=10000)
                ("unary minus",(20000,20000)) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
```

```
                tags: ["buffer", "string"]
                ("deserialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("deserialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("serialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
        "read" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "stream", "string"]
                "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
"problem" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["example", "kernel"]
        "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["spell", "string"]
                "spellcheck" => Benchmark(evals=1, seconds=1.0, samples=10000)
"dates" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "string" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                "Date" => Benchmark(evals=394, seconds=1.0, samples=10000)
                "DateTime" => Benchmark(evals=183, seconds=1.0, samples=10000)
        "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("Date","Month") => Benchmark(evals=989, seconds=1.0, samples=10000)
                ("DateTime","Millisecond") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Month") => Benchmark(evals=979, seconds=1.0, samples=10000)
                ("Date","Year") => Benchmark(evals=995, seconds=1.0, samples=10000)
                ("DateTime","Second") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("Date","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Year") => Benchmark(evals=987, seconds=1.0, samples=10000)
                ("DateTime","Hour") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
"sparse" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["array"]
        "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("unary minus",(600,600)) => Benchmark(evals=5, seconds=1.0, samples=10000)
                ("unary minus",(20000,20000)) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
```

```
("*","LowerTriangular","Vector",256) => Benchmark(evals=8, seconds=20.0, samples=100)
("*","Tridiagonal","Tridiagonal",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("\\","Matrix","Vector",256) => Benchmark(evals=1, seconds=20.0, samples=100)
```

In [4]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

```
5-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "join" => Benchmark(evals=1, seconds=1.0, samples=10000)
          "replace" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("deserialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("deserialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("serialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
          "read" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "stream", "string"]
                  "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                  "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
  "problem" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["example", "kernel"]
          "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["spell", "string"]
                  "spellcheck" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "dates" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "string" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  "Date" => Benchmark(evals=394, seconds=1.0, samples=10000)
                  "DateTime" => Benchmark(evals=183, seconds=1.0, samples=10000)
          "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  ("Date","Month") => Benchmark(evals=989, seconds=1.0, samples=10000)
                  ("DateTime","Millisecond") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                  ("DateTime","Month") => Benchmark(evals=979, seconds=1.0, samples=10000)
```

```
("*","LowerTriangular","Vector",256) => Benchmark(evals=8, seconds=20.0, samples=100)
("*","Tridiagonal","Tridiagonal",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("\\","Matrix","Vector",256) => Benchmark(evals=1, seconds=20.0, samples=100)
```

In [4]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

```
5-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "join" => Benchmark(evals=1, seconds=1.0, samples=10000)
          "replace" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("deserialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("deserialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("serialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
          "read" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "stream", "string"]
                  "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                  "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
  "problem" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["example", "kernel"]
          "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["spell", "string"]
                  "spellcheck" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "dates" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "string" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  "Date" => Benchmark(evals=394, seconds=1.0, samples=10000)
                  "DateTime" => Benchmark(evals=183, seconds=1.0, samples=10000)
          "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  ("Date","Month") => Benchmark(evals=989, seconds=1.0, samples=10000)
                  ("DateTime","Millisecond") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                  ("DateTime","Month") => Benchmark(evals=979, seconds=1.0, samples=10000)
```

```
        "join" => Benchmark(evals=1, seconds=1.0, samples=10000)
        "replace" => Benchmark(evals=1, seconds=1.0, samples=10000)
"io" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "string"]
                ("deserialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("deserialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("serialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
        "read" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "stream", "string"]
                "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
"problem" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["example", "kernel"]
        "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["spell", "string"]
                "spellcheck" => Benchmark(evals=1, seconds=1.0, samples=10000)
"dates" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "string" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                "Date" => Benchmark(evals=394, seconds=1.0, samples=10000)
                "DateTime" => Benchmark(evals=183, seconds=1.0, samples=10000)
        "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("Date","Month") => Benchmark(evals=989, seconds=1.0, samples=10000)
                ("DateTime","Millisecond") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Month") => Benchmark(evals=979, seconds=1.0, samples=10000)
                ("Date","Year") => Benchmark(evals=995, seconds=1.0, samples=10000)
                ("DateTime","Second") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("Date","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Year") => Benchmark(evals=987, seconds=1.0, samples=10000)
                ("DateTime","Hour") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
"sparse" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["array"]
        "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("unary minus",(600,600)) => Benchmark(evals=5, seconds=1.0, samples=10000)
```

```
        tags: []
        "join" => Benchmark(evals=1, seconds=1.0, samples=10000)
        "replace" => Benchmark(evals=1, seconds=1.0, samples=10000)
"io" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "string"]
                ("deserialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("deserialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("serialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
        "read" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "stream", "string"]
                "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
"problem" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["example", "kernel"]
        "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["spell", "string"]
                "spellcheck" => Benchmark(evals=1, seconds=1.0, samples=10000)
"dates" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "string" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                "Date" => Benchmark(evals=394, seconds=1.0, samples=10000)
                "DateTime" => Benchmark(evals=183, seconds=1.0, samples=10000)
        "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("Date","Month") => Benchmark(evals=989, seconds=1.0, samples=10000)
                ("DateTime","Millisecond") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Month") => Benchmark(evals=979, seconds=1.0, samples=10000)
                ("Date","Year") => Benchmark(evals=995, seconds=1.0, samples=10000)
                ("DateTime","Second") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("Date","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Year") => Benchmark(evals=987, seconds=1.0, samples=10000)
                ("DateTime","Hour") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
"sparse" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["array"]
        "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
```

```
                    "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                    "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
  "problem" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["example", "kernel"]
          "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["spell", "string"]
                  "spellcheck" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "dates" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "string" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  "Date" => Benchmark(evals=394, seconds=1.0, samples=10000)
                  "DateTime" => Benchmark(evals=183, seconds=1.0, samples=10000)
          "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  ("Date","Month") => Benchmark(evals=989, seconds=1.0, samples=10000)
                  ("DateTime","Millisecond") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                  ("DateTime","Month") => Benchmark(evals=979, seconds=1.0, samples=10000)
                  ("Date","Year") => Benchmark(evals=995, seconds=1.0, samples=10000)
                  ("DateTime","Second") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                  ("Date","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                  ("DateTime","Year") => Benchmark(evals=987, seconds=1.0, samples=10000)
                  ("DateTime","Hour") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                  ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                  ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
  "sparse" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["array"]
          "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  ("unary minus",(600,600)) => Benchmark(evals=5, seconds=1.0, samples=10000)
                  ("unary minus",(20000,20000)) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

```
        tags: ["example", "kernel"]
        "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["spell", "string"]
                "spellcheck" => Benchmark(evals=1, seconds=1.0, samples=10000)
    "dates" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "string" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                "Date" => Benchmark(evals=394, seconds=1.0, samples=10000)
                "DateTime" => Benchmark(evals=183, seconds=1.0, samples=10000)
        "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("Date","Month") => Benchmark(evals=989, seconds=1.0, samples=10000)
                ("DateTime","Millisecond") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Month") => Benchmark(evals=979, seconds=1.0, samples=10000)
                ("Date","Year") => Benchmark(evals=995, seconds=1.0, samples=10000)
                ("DateTime","Second") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("Date","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Year") => Benchmark(evals=987, seconds=1.0, samples=10000)
                ("DateTime","Hour") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
    "sparse" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["array"]
        "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("unary minus",(600,600)) => Benchmark(evals=5, seconds=1.0, samples=10000)
                ("unary minus",(20000,20000)) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

In [ ]:
```
vs = judge(minimum(r6), minimum(r5), time_tolerance = 0.05)
```

```
                tags: []
                "Date" => Benchmark(evals=394, seconds=1.0, samples=10000)
                "DateTime" => Benchmark(evals=183, seconds=1.0, samples=10000)
        "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("Date","Month") => Benchmark(evals=989, seconds=1.0, samples=10000)
                ("DateTime","Millisecond") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Month") => Benchmark(evals=979, seconds=1.0, samples=10000)
                ("Date","Year") => Benchmark(evals=995, seconds=1.0, samples=10000)
                ("DateTime","Second") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("Date","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Year") => Benchmark(evals=987, seconds=1.0, samples=10000)
                ("DateTime","Hour") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
  "sparse" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["array"]
          "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  ("unary minus",(600,600)) => Benchmark(evals=5, seconds=1.0, samples=10000)
                  ("unary minus",(20000,20000)) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

In [ ]:
```
vs = judge(minimum(r6), minimum(r5), time_tolerance = 0.05)
showall(vs)
```

In [ ]:
```
showall(regressions(vs))
```

In [ ]:
```
showall(improvements(vs))
```

In [ ]:
```
id = ["io", "serialization", ("serialize","Matrix{Float64}")]
```

```
                tags: []
                "Date" => Benchmark(evals=394, seconds=1.0, samples=10000)
                "DateTime" => Benchmark(evals=183, seconds=1.0, samples=10000)
        "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("Date","Month") => Benchmark(evals=989, seconds=1.0, samples=10000)
                ("DateTime","Millisecond") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Month") => Benchmark(evals=979, seconds=1.0, samples=10000)
                ("Date","Year") => Benchmark(evals=995, seconds=1.0, samples=10000)
                ("DateTime","Second") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("Date","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Year") => Benchmark(evals=987, seconds=1.0, samples=10000)
                ("DateTime","Hour") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
  "sparse" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["array"]
          "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("unary minus",(600,600)) => Benchmark(evals=5, seconds=1.0, samples=10000)
                ("unary minus",(20000,20000)) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

In [ ]:
```
vs = judge(minimum(r6), minimum(r5), time_tolerance = 0.05)
showall(vs)
```

In [ ]:
```
showall(regressions(vs))
```

In [ ]:
```
showall(improvements(vs))
```

In [ ]:
```
id = ["io", "serialization", ("serialize","Matrix{Float64}")]
```

```
problem" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["example", "kernel"]
        "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["spell", "string"]
                "spellcheck" => Benchmark(evals=1, seconds=1.0, samples=10000)
"dates" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "string" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                "Date" => Benchmark(evals=394, seconds=1.0, samples=10000)
                "DateTime" => Benchmark(evals=183, seconds=1.0, samples=10000)
        "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("Date","Month") => Benchmark(evals=989, seconds=1.0, samples=10000)
                ("DateTime","Millisecond") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Month") => Benchmark(evals=979, seconds=1.0, samples=10000)
                ("Date","Year") => Benchmark(evals=995, seconds=1.0, samples=10000)
                ("DateTime","Second") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("Date","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Year") => Benchmark(evals=987, seconds=1.0, samples=10000)
                ("DateTime","Hour") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
"sparse" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["array"]
        "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("unary minus",(600,600)) => Benchmark(evals=5, seconds=1.0, samples=10000)
                ("unary minus",(20000,20000)) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

In [ ]:
```
vs = judge(minimum(r6), minimum(r5), time_tolerance = 0.05)
```

```
        tags: []
        "join" => Benchmark(evals=1, seconds=1.0, samples=10000)
        "replace" => Benchmark(evals=1, seconds=1.0, samples=10000)
"io" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "string"]
                ("deserialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("deserialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("serialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
        "read" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "stream", "string"]
                "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
"problem" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["example", "kernel"]
        "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["spell", "string"]
                "spellcheck" => Benchmark(evals=1, seconds=1.0, samples=10000)
"dates" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "string" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                "Date" => Benchmark(evals=394, seconds=1.0, samples=10000)
                "DateTime" => Benchmark(evals=183, seconds=1.0, samples=10000)
        "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("Date","Month") => Benchmark(evals=989, seconds=1.0, samples=10000)
                ("DateTime","Millisecond") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Month") => Benchmark(evals=979, seconds=1.0, samples=10000)
                ("Date","Year") => Benchmark(evals=995, seconds=1.0, samples=10000)
                ("DateTime","Second") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("Date","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Year") => Benchmark(evals=987, seconds=1.0, samples=10000)
                ("DateTime","Hour") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
"sparse" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["array"]
        "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
```

```
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

```
5-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "join" => Benchmark(evals=1, seconds=1.0, samples=10000)
          "replace" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("deserialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("deserialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("serialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
          "read" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "stream", "string"]
                  "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                  "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
  "problem" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["example", "kernel"]
          "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["spell", "string"]
                  "spellcheck" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "dates" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "string" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  "Date" => Benchmark(evals=394, seconds=1.0, samples=10000)
                  "DateTime" => Benchmark(evals=183, seconds=1.0, samples=10000)
          "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  ("Date","Month") => Benchmark(evals=989, seconds=1.0, samples=10000)
                  ("DateTime","Millisecond") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                  ("DateTime","Month") => Benchmark(evals=979, seconds=1.0, samples=10000)
                  ("Date","Year") => Benchmark(evals=995, seconds=1.0, samples=10000)
                  ("DateTime","Second") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                  ("Date","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                  ("DateTime","Year") => Benchmark(evals=987, seconds=1.0, samples=10000)
```

```
("-","Diagonal","Diagonal",1024) => Benchmark(evals=19, seconds=20.0, samples=100)
("sqrtm","UpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","Tridiagonal","Tridiagonal",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","LowerTriangular","Vector",256) => Benchmark(evals=8, seconds=20.0, samples=100)
("*","Tridiagonal","Tridiagonal",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("\\","Matrix","Vector",256) => Benchmark(evals=1, seconds=20.0, samples=100)
```

In [4]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

```
5-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "join" => Benchmark(evals=1, seconds=1.0, samples=10000)
          "replace" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("deserialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("deserialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("serialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
          "read" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "stream", "string"]
                  "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                  "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
  "problem" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["example", "kernel"]
          "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["spell", "string"]
                  "spellcheck" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "dates" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "string" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  "Date" => Benchmark(evals=394, seconds=1.0, samples=10000)
                  "DateTime" => Benchmark(evals=183, seconds=1.0, samples=10000)
          "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                  tags: []
```

```
("-","Diagonal","Diagonal",1024) => Benchmark(evals=19, seconds=20.0, samples=100)
("sqrtm","UpperTriangular",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","Tridiagonal","Tridiagonal",256) => Benchmark(evals=1, seconds=20.0, samples=100)
("*","LowerTriangular","Vector",256) => Benchmark(evals=8, seconds=20.0, samples=100)
("*","Tridiagonal","Tridiagonal",1024) => Benchmark(evals=1, seconds=20.0, samples=100)
("\\","Matrix","Vector",256) => Benchmark(evals=1, seconds=20.0, samples=100)
```

In [4]:
```
# Still a lot of benchmarks! Let's filter further:
group = suite[@tagged ("arithmetic" || "string") && !("scalar" || "linalg")]
showall(group)
```

```
5-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "join" => Benchmark(evals=1, seconds=1.0, samples=10000)
          "replace" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("deserialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("deserialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("serialize","Matrix{Float64}") => Benchmark(evals=1, seconds=1.0, samples=10000)
                  ("serialize","Vector{String}") => Benchmark(evals=1, seconds=1.0, samples=10000)
          "read" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "stream", "string"]
                  "readstring" => Benchmark(evals=995, seconds=1.0, samples=10000)
                  "read" => Benchmark(evals=5, seconds=1.0, samples=10000)
  "problem" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["example", "kernel"]
          "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["spell", "string"]
                  "spellcheck" => Benchmark(evals=1, seconds=1.0, samples=10000)
  "dates" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "string" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  "Date" => Benchmark(evals=394, seconds=1.0, samples=10000)
                  "DateTime" => Benchmark(evals=183, seconds=1.0, samples=10000)
          "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                  tags: []
```

```
                    ("Date","Month") => Benchmark(evals=989, seconds=1.0, samples=10000)
                    ("DateTime","Millisecond") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                    ("DateTime","Month") => Benchmark(evals=979, seconds=1.0, samples=10000)
                    ("Date","Year") => Benchmark(evals=995, seconds=1.0, samples=10000)
                    ("DateTime","Second") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                    ("Date","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                    ("DateTime","Year") => Benchmark(evals=987, seconds=1.0, samples=10000)
                    ("DateTime","Hour") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                    ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                    ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
  "sparse" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["array"]
          "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  ("unary minus",(600,600)) => Benchmark(evals=5, seconds=1.0, samples=10000)
                  ("unary minus",(20000,20000)) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

In [ ]:
```
vs = judge(minimum(r6), minimum(r5), time_tolerance = 0.05)
showall(vs)
```

In [ ]:
```
showall(regressions(vs))
```

In [ ]:
```
showall(improvements(vs))
```

In [ ]:
```
id = ["io", "serialization", ("serialize","Matrix{Float64}")]
```

In [ ]:
```
t5 = r5[id]
```

In [ ]:
```
t6 = r6[id]
```

```
                    ("Date","Year") => Benchmark(evals=995, seconds=1.0, samples=10000)
                    ("DateTime","Second") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                    ("Date","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                    ("DateTime","Year") => Benchmark(evals=987, seconds=1.0, samples=10000)
                    ("DateTime","Hour") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                    ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                    ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
  "sparse" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["array"]
          "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  ("unary minus",(600,600)) => Benchmark(evals=5, seconds=1.0, samples=10000)
                  ("unary minus",(20000,20000)) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

In [ ]:
```
vs = judge(minimum(r6), minimum(r5), time_tolerance = 0.05)
showall(vs)
```

In [ ]:
```
showall(regressions(vs))
```

In [ ]:
```
showall(improvements(vs))
```

In [ ]:
```
id = ["io", "serialization", ("serialize","Matrix{Float64}")]
```

In [ ]:
```
t5 = r5[id]
```

In [ ]:
```
t6 = r6[id]
```

In [ ]:
```
plotkde([t6, t5])
```

```
                    ("DateTime","Hour") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                    ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                    ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
  "sparse" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["array"]
          "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  ("unary minus",(600,600)) => Benchmark(evals=5, seconds=1.0, samples=10000)
                  ("unary minus",(20000,20000)) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [ ]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

In [ ]:
```
vs = judge(minimum(r6), minimum(r5), time_tolerance = 0.05)
showall(vs)
```

In [ ]:
```
showall(regressions(vs))
```

In [ ]:
```
showall(improvements(vs))
```

In [ ]:
```
id = ["io", "serialization", ("serialize","Matrix{Float64}")]
```

In [ ]:
```
t5 = r5[id]
```

In [ ]:
```
t6 = r6[id]
```

In [ ]:
```
plotkde([t6, t5])
```

In [ ]:
```
```

```
                ("DateTime","Day") => Benchmark(evals=1000, seconds=1.0, samples=10000)
                ("DateTime","Minute") => Benchmark(evals=1000, seconds=1.0, samples=10000)
        "sparse" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["array"]
                "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                        tags: []
                        ("unary minus",(600,600)) => Benchmark(evals=5, seconds=1.0, samples=10000)
                        ("unary minus",(20000,20000)) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [5]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

```
5-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "join" => Trial(219.973 ms)
          "replace" => Trial(216.335 μs)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("deserialize","Matrix{Float64}") => Trial(1.480 ms)
                  ("deserialize","Vector{String}") => Trial(1.796 ms)
                  ("serialize","Matrix{Float64}") => Trial(4.733 ms)
                  ("serialize","Vector{String}") => Trial(402.742 μs)
          "read" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "stream", "string"]
                  "readstring" => Trial(50.633 ns)
                  "read" => Trial(8.639 μs)
  "problem" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["example", "kernel"]
          "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["spell", "string"]
```

In [5]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

```
5-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "join" => Trial(219.973 ms)
          "replace" => Trial(216.335 μs)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("deserialize","Matrix{Float64}") => Trial(1.480 ms)
                  ("deserialize","Vector{String}") => Trial(1.796 ms)
                  ("serialize","Matrix{Float64}") => Trial(4.733 ms)
                  ("serialize","Vector{String}") => Trial(402.742 μs)
          "read" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "stream", "string"]
                  "readstring" => Trial(50.633 ns)
                  "read" => Trial(8.639 μs)
  "problem" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["example", "kernel"]
          "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["spell", "string"]
                  "spellcheck" => Trial(11.643 s)
  "dates" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "string" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  "Date" => Trial(706.635 ns)
                  "DateTime" => Trial(1.583 μs)
          "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  ("Date","Month") => Trial(62.049 ns)
                  ("DateTime","Millisecond") => Trial(1.646 ns)
                  ("DateTime","Month") => Trial(90.609 ns)
                  ("Date","Year") => Trial(39.496 ns)
                  ("DateTime","Second") => Trial(1.646 ns)
                  ("Date","Day") => Trial(1.644 ns)
```

```
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)

5-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "join" => Trial(219.973 ms)
          "replace" => Trial(216.335 μs)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("deserialize","Matrix{Float64}") => Trial(1.480 ms)
                  ("deserialize","Vector{String}") => Trial(1.796 ms)
                  ("serialize","Matrix{Float64}") => Trial(4.733 ms)
                  ("serialize","Vector{String}") => Trial(402.742 μs)
          "read" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "stream", "string"]
                  "readstring" => Trial(50.633 ns)
                  "read" => Trial(8.639 μs)
  "problem" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["example", "kernel"]
          "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["spell", "string"]
                  "spellcheck" => Trial(11.643 s)
  "dates" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "string" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  "Date" => Trial(706.635 ns)
                  "DateTime" => Trial(1.583 μs)
          "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  ("Date","Month") => Trial(62.049 ns)
                  ("DateTime","Millisecond") => Trial(1.646 ns)
                  ("DateTime","Month") => Trial(90.609 ns)
                  ("Date","Year") => Trial(39.496 ns)
                  ("DateTime","Second") => Trial(1.646 ns)
                  ("Date","Day") => Trial(1.644 ns)
                  ("DateTime","Year") => Trial(68.363 ns)
                  ("DateTime","Hour") => Trial(1.648 ns)
```

```
        tags: []
        "join" => Trial(219.973 ms)
        "replace" => Trial(216.335 μs)
"io" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "string"]
                ("deserialize","Matrix{Float64}") => Trial(1.480 ms)
                ("deserialize","Vector{String}") => Trial(1.796 ms)
                ("serialize","Matrix{Float64}") => Trial(4.733 ms)
                ("serialize","Vector{String}") => Trial(402.742 μs)
        "read" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "stream", "string"]
                "readstring" => Trial(50.633 ns)
                "read" => Trial(8.639 μs)
"problem" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["example", "kernel"]
        "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["spell", "string"]
                "spellcheck" => Trial(11.643 s)
"dates" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "string" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                "Date" => Trial(706.635 ns)
                "DateTime" => Trial(1.583 μs)
        "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("Date","Month") => Trial(62.049 ns)
                ("DateTime","Millisecond") => Trial(1.646 ns)
                ("DateTime","Month") => Trial(90.609 ns)
                ("Date","Year") => Trial(39.496 ns)
                ("DateTime","Second") => Trial(1.646 ns)
                ("Date","Day") => Trial(1.644 ns)
                ("DateTime","Year") => Trial(68.363 ns)
                ("DateTime","Hour") => Trial(1.648 ns)
                ("DateTime","Day") => Trial(1.647 ns)
                ("DateTime","Minute") => Trial(1.649 ns)
"sparse" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["array"]
        "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
```

```
        "replace" => Trial(216.335 μs)
    "io" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "string"]
                ("deserialize","Matrix{Float64}") => Trial(1.480 ms)
                ("deserialize","Vector{String}") => Trial(1.796 ms)
                ("serialize","Matrix{Float64}") => Trial(4.733 ms)
                ("serialize","Vector{String}") => Trial(402.742 μs)
        "read" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "stream", "string"]
                "readstring" => Trial(50.633 ns)
                "read" => Trial(8.639 μs)
    "problem" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["example", "kernel"]
        "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["spell", "string"]
                "spellcheck" => Trial(11.643 s)
    "dates" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "string" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                "Date" => Trial(706.635 ns)
                "DateTime" => Trial(1.583 μs)
        "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("Date","Month") => Trial(62.049 ns)
                ("DateTime","Millisecond") => Trial(1.646 ns)
                ("DateTime","Month") => Trial(90.609 ns)
                ("Date","Year") => Trial(39.496 ns)
                ("DateTime","Second") => Trial(1.646 ns)
                ("Date","Day") => Trial(1.644 ns)
                ("DateTime","Year") => Trial(68.363 ns)
                ("DateTime","Hour") => Trial(1.648 ns)
                ("DateTime","Day") => Trial(1.647 ns)
                ("DateTime","Minute") => Trial(1.649 ns)
    "sparse" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["array"]
        "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("unary minus",(600,600)) => Trial(13.990 μs)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [5]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

```
5-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "join" => Trial(219.973 ms)
          "replace" => Trial(216.335 μs)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("deserialize","Matrix{Float64}") => Trial(1.480 ms)
                  ("deserialize","Vector{String}") => Trial(1.796 ms)
                  ("serialize","Matrix{Float64}") => Trial(4.733 ms)
                  ("serialize","Vector{String}") => Trial(402.742 μs)
          "read" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "stream", "string"]
                  "readstring" => Trial(50.633 ns)
                  "read" => Trial(8.639 μs)
  "problem" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["example", "kernel"]
          "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["spell", "string"]
                  "spellcheck" => Trial(11.643 s)
  "dates" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "string" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  "Date" => Trial(706.635 ns)
                  "DateTime" => Trial(1.583 μs)
          "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  ("Date","Month") => Trial(62.049 ns)
```

```
        tags: ["array"]
        "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("unary minus",(600,600)) => Benchmark(evals=5, seconds=1.0, samples=10000)
                ("unary minus",(20000,20000)) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [5]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

```
5-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "join" => Trial(219.973 ms)
          "replace" => Trial(216.335 μs)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("deserialize","Matrix{Float64}") => Trial(1.480 ms)
                  ("deserialize","Vector{String}") => Trial(1.796 ms)
                  ("serialize","Matrix{Float64}") => Trial(4.733 ms)
                  ("serialize","Vector{String}") => Trial(402.742 μs)
          "read" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "stream", "string"]
                  "readstring" => Trial(50.633 ns)
                  "read" => Trial(8.639 μs)
  "problem" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["example", "kernel"]
          "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["spell", "string"]
                  "spellcheck" => Trial(11.643 s)
  "dates" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
```

```
        tags: ["array"]
        "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("unary minus",(600,600)) => Benchmark(evals=5, seconds=1.0, samples=10000)
                ("unary minus",(20000,20000)) => Benchmark(evals=1, seconds=1.0, samples=10000)
```

In [ ]:
```
# Okay, that seems like a reasonable set. Let's run these:
results = run(group, verbose=true)
BenchmarkTools.save("results.jld", "results", results)
```

In [5]:
```
# assumes `pwd` points to this notebook's directory
r5 = BenchmarkTools.load(joinpath(pwd(), "julia-05-results.jld"), "results")
r6 = BenchmarkTools.load(joinpath(pwd(), "julia-06-results.jld"), "results")
showall(r5)
```

```
5-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "join" => Trial(219.973 ms)
          "replace" => Trial(216.335 μs)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("deserialize","Matrix{Float64}") => Trial(1.480 ms)
                  ("deserialize","Vector{String}") => Trial(1.796 ms)
                  ("serialize","Matrix{Float64}") => Trial(4.733 ms)
                  ("serialize","Vector{String}") => Trial(402.742 μs)
          "read" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "stream", "string"]
                  "readstring" => Trial(50.633 ns)
                  "read" => Trial(8.639 μs)
  "problem" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["example", "kernel"]
          "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["spell", "string"]
                  "spellcheck" => Trial(11.643 s)
  "dates" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
```

```
                ("serialize","Matrix{Float64}") => Trial(4.733 ms)
                ("serialize","Vector{String}") => Trial(402.742 μs)
        "read" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "stream", "string"]
                "readstring" => Trial(50.633 ns)
                "read" => Trial(8.639 μs)
"problem" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["example", "kernel"]
        "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["spell", "string"]
                "spellcheck" => Trial(11.643 s)
"dates" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "string" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                "Date" => Trial(706.635 ns)
                "DateTime" => Trial(1.583 μs)
        "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("Date","Month") => Trial(62.049 ns)
                ("DateTime","Millisecond") => Trial(1.646 ns)
                ("DateTime","Month") => Trial(90.609 ns)
                ("Date","Year") => Trial(39.496 ns)
                ("DateTime","Second") => Trial(1.646 ns)
                ("Date","Day") => Trial(1.644 ns)
                ("DateTime","Year") => Trial(68.363 ns)
                ("DateTime","Hour") => Trial(1.648 ns)
                ("DateTime","Day") => Trial(1.647 ns)
                ("DateTime","Minute") => Trial(1.649 ns)
"sparse" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["array"]
        "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("unary minus",(600,600)) => Trial(13.990 μs)
                ("unary minus",(20000,20000)) => Trial(26.864 ms)
```

In [ ]:
```
vs = judge(minimum(r6), minimum(r5), time_tolerance = 0.05)
showall(vs)
```

In [ ]:
```
showall(regressions(vs))
```

```
"io" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "string"]
                ("deserialize","Matrix{Float64}") => Trial(1.480 ms)
                ("deserialize","Vector{String}") => Trial(1.796 ms)
                ("serialize","Matrix{Float64}") => Trial(4.733 ms)
                ("serialize","Vector{String}") => Trial(402.742 μs)
        "read" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "stream", "string"]
                "readstring" => Trial(50.633 ns)
                "read" => Trial(8.639 μs)
"problem" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["example", "kernel"]
```

In [*]:
```
vs = judge(minimum(r6), minimum(r5), time_tolerance = 0.05)
showall(vs)
```

In [ ]:
```
showall(regressions(vs))
```

In [ ]:
```
showall(improvements(vs))
```

In [ ]:
```
id = ["io", "serialization", ("serialize","Matrix{Float64}")]
```

In [ ]:
```
t5 = r5[id]
```

In [ ]:
```
t6 = r6[id]
```

In [ ]:
```
plotkde([t6, t5])
```

In [ ]:

```
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("deserialize","Matrix{Float64}") => Trial(1.480 ms)
                  ("deserialize","Vector{String}") => Trial(1.796 ms)
                  ("serialize","Matrix{Float64}") => Trial(4.733 ms)
                  ("serialize","Vector{String}") => Trial(402.742 μs)
          "read" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "stream", "string"]
                  "readstring" => Trial(50.633 ns)
                  "read" => Trial(8.639 μs)
  "problem" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["example", "kernel"]
```

In [6]:
```
vs = judge(minimum(r6), minimum(r5), time_tolerance = 0.05)
showall(vs)
```

```
5-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "join" => TrialJudgement(-34.21% => improvement)
          "replace" => TrialJudgement(-24.75% => improvement)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("deserialize","Matrix{Float64}") => TrialJudgement(+0.29% => invariant)
                  ("deserialize","Vector{String}") => TrialJudgement(-83.04% => improvement)
                  ("serialize","Matrix{Float64}") => TrialJudgement(+8.16% => regression)
                  ("serialize","Vector{String}") => TrialJudgement(-64.20% => improvement)
          "read" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "stream", "string"]
                  "readstring" => TrialJudgement(-11.47% => improvement)
                  "read" => TrialJudgement(+0.01% => invariant)
  "problem" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["example", "kernel"]
          "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["spell", "string"]
                  "spellcheck" => TrialJudgement(-36.05% => improvement)
  "dates" => 2-element BenchmarkTools.BenchmarkGroup:
```

```
                    "readstring" => Trial(50.633 ns)
                    "read" => Trial(8.639 μs)
  "problem" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["example", "kernel"]
```

In [6]:
```
vs = judge(minimum(r6), minimum(r5), time_tolerance = 0.05)
showall(vs)
```

```
5-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "join" => TrialJudgement(-34.21% => improvement)
          "replace" => TrialJudgement(-24.75% => improvement)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("deserialize","Matrix{Float64}") => TrialJudgement(+0.29% => invariant)
                  ("deserialize","Vector{String}") => TrialJudgement(-83.04% => improvement)
                  ("serialize","Matrix{Float64}") => TrialJudgement(+8.16% => regression)
                  ("serialize","Vector{String}") => TrialJudgement(-64.20% => improvement)
          "read" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "stream", "string"]
                  "readstring" => TrialJudgement(-11.47% => improvement)
                  "read" => TrialJudgement(+0.01% => invariant)
  "problem" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["example", "kernel"]
          "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["spell", "string"]
                  "spellcheck" => TrialJudgement(-36.05% => improvement)
  "dates" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "string" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  "Date" => TrialJudgement(-52.70% => improvement)
                  "DateTime" => TrialJudgement(-51.71% => improvement)
          "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  ("Date","Month") => TrialJudgement(-6.56% => improvement)
                  ("DateTime","Millisecond") => TrialJudgement(-0.06% => invariant)
```

tags: ["example", "kernel"]

In [6]:
```
vs = judge(minimum(r6), minimum(r5), time_tolerance = 0.05)
showall(vs)
```

```
5-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "join" => TrialJudgement(-34.21% => improvement)
          "replace" => TrialJudgement(-24.75% => improvement)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("deserialize","Matrix{Float64}") => TrialJudgement(+0.29% => invariant)
                  ("deserialize","Vector{String}") => TrialJudgement(-83.04% => improvement)
                  ("serialize","Matrix{Float64}") => TrialJudgement(+8.16% => regression)
                  ("serialize","Vector{String}") => TrialJudgement(-64.20% => improvement)
          "read" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "stream", "string"]
                  "readstring" => TrialJudgement(-11.47% => improvement)
                  "read" => TrialJudgement(+0.01% => invariant)
  "problem" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["example", "kernel"]
          "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["spell", "string"]
                  "spellcheck" => TrialJudgement(-36.05% => improvement)
  "dates" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "string" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  "Date" => TrialJudgement(-52.70% => improvement)
                  "DateTime" => TrialJudgement(-51.71% => improvement)
          "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  ("Date","Month") => TrialJudgement(-6.56% => improvement)
                  ("DateTime","Millisecond") => TrialJudgement(-0.06% => invariant)
                  ("DateTime","Month") => TrialJudgement(-7.19% => improvement)
                  ("Date","Year") => TrialJudgement(-7.57% => improvement)
                  ("DateTime","Second") => TrialJudgement(+0.06% => invariant)
                  ("Date","Day") => TrialJudgement(+0.12% => invariant)
```

In [6]:
```
vs = judge(minimum(r6), minimum(r5), time_tolerance = 0.05)
showall(vs)
```

```
5-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "join" => TrialJudgement(-34.21% => improvement)
          "replace" => TrialJudgement(-24.75% => improvement)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("deserialize","Matrix{Float64}") => TrialJudgement(+0.29% => invariant)
                  ("deserialize","Vector{String}") => TrialJudgement(-83.04% => improvement)
                  ("serialize","Matrix{Float64}") => TrialJudgement(+8.16% => regression)
                  ("serialize","Vector{String}") => TrialJudgement(-64.20% => improvement)
          "read" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "stream", "string"]
                  "readstring" => TrialJudgement(-11.47% => improvement)
                  "read" => TrialJudgement(+0.01% => invariant)
  "problem" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["example", "kernel"]
          "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["spell", "string"]
                  "spellcheck" => TrialJudgement(-36.05% => improvement)
  "dates" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "string" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  "Date" => TrialJudgement(-52.70% => improvement)
                  "DateTime" => TrialJudgement(-51.71% => improvement)
          "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  ("Date","Month") => TrialJudgement(-6.56% => improvement)
                  ("DateTime","Millisecond") => TrialJudgement(-0.06% => invariant)
                  ("DateTime","Month") => TrialJudgement(-7.19% => improvement)
                  ("Date","Year") => TrialJudgement(-7.57% => improvement)
                  ("DateTime","Second") => TrialJudgement(+0.06% => invariant)
                  ("Date","Day") => TrialJudgement(+0.12% => invariant)
                  ("DateTime","Year") => TrialJudgement(-8.18% => improvement)
                  ("DateTime","Hour") => TrialJudgement(+2.73% => invariant)
```

```
showall(vs)

5-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "string" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "join" => TrialJudgement(-34.21% => improvement)
          "replace" => TrialJudgement(-24.75% => improvement)
  "io" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("deserialize","Matrix{Float64}") => TrialJudgement(+0.29% => invariant)
                  ("deserialize","Vector{String}") => TrialJudgement(-83.04% => improvement)
                  ("serialize","Matrix{Float64}") => TrialJudgement(+8.16% => regression)
                  ("serialize","Vector{String}") => TrialJudgement(-64.20% => improvement)
          "read" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "stream", "string"]
                  "readstring" => TrialJudgement(-11.47% => improvement)
                  "read" => TrialJudgement(+0.01% => invariant)
  "problem" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["example", "kernel"]
          "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["spell", "string"]
                  "spellcheck" => TrialJudgement(-36.05% => improvement)
  "dates" => 2-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "string" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  "Date" => TrialJudgement(-52.70% => improvement)
                  "DateTime" => TrialJudgement(-51.71% => improvement)
          "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  ("Date","Month") => TrialJudgement(-6.56% => improvement)
                  ("DateTime","Millisecond") => TrialJudgement(-0.06% => invariant)
                  ("DateTime","Month") => TrialJudgement(-7.19% => improvement)
                  ("Date","Year") => TrialJudgement(-7.57% => improvement)
                  ("DateTime","Second") => TrialJudgement(+0.06% => invariant)
                  ("Date","Day") => TrialJudgement(+0.12% => invariant)
                  ("DateTime","Year") => TrialJudgement(-8.18% => improvement)
                  ("DateTime","Hour") => TrialJudgement(+2.73% => invariant)
```

```
tags: []
"string" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "join" => TrialJudgement(-34.21% => improvement)
        "replace" => TrialJudgement(-24.75% => improvement)
"io" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "serialization" => 4-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "string"]
                ("deserialize","Matrix{Float64}") => TrialJudgement(+0.29% => invariant)
                ("deserialize","Vector{String}") => TrialJudgement(-83.04% => improvement)
                ("serialize","Matrix{Float64}") => TrialJudgement(+8.16% => regression)
                ("serialize","Vector{String}") => TrialJudgement(-64.20% => improvement)
        "read" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "stream", "string"]
                "readstring" => TrialJudgement(-11.47% => improvement)
                "read" => TrialJudgement(+0.01% => invariant)
"problem" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["example", "kernel"]
        "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["spell", "string"]
                "spellcheck" => TrialJudgement(-36.05% => improvement)
"dates" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "string" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                "Date" => TrialJudgement(-52.70% => improvement)
                "DateTime" => TrialJudgement(-51.71% => improvement)
        "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("Date","Month") => TrialJudgement(-6.56% => improvement)
                ("DateTime","Millisecond") => TrialJudgement(-0.06% => invariant)
                ("DateTime","Month") => TrialJudgement(-7.19% => improvement)
                ("Date","Year") => TrialJudgement(-7.57% => improvement)
                ("DateTime","Second") => TrialJudgement(+0.06% => invariant)
                ("Date","Day") => TrialJudgement(+0.12% => invariant)
                ("DateTime","Year") => TrialJudgement(-8.18% => improvement)
                ("DateTime","Hour") => TrialJudgement(+2.73% => invariant)
                ("DateTime","Day") => TrialJudgement(+0.24% => invariant)
                ("DateTime","Minute") => TrialJudgement(-0.12% => invariant)
"sparse" => 1-element BenchmarkTools.BenchmarkGroup:
```

```
                ("deserialize","Matrix{Float64}") => TrialJudgement(+0.29% => invariant)
                ("deserialize","Vector{String}") => TrialJudgement(-83.04% => improvement)
                ("serialize","Matrix{Float64}") => TrialJudgement(+8.16% => regression)
                ("serialize","Vector{String}") => TrialJudgement(-64.20% => improvement)
        "read" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "stream", "string"]
                "readstring" => TrialJudgement(-11.47% => improvement)
                "read" => TrialJudgement(+0.01% => invariant)
"problem" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["example", "kernel"]
        "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["spell", "string"]
                "spellcheck" => TrialJudgement(-36.05% => improvement)
"dates" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "string" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                "Date" => TrialJudgement(-52.70% => improvement)
                "DateTime" => TrialJudgement(-51.71% => improvement)
        "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("Date","Month") => TrialJudgement(-6.56% => improvement)
                ("DateTime","Millisecond") => TrialJudgement(-0.06% => invariant)
                ("DateTime","Month") => TrialJudgement(-7.19% => improvement)
                ("Date","Year") => TrialJudgement(-7.57% => improvement)
                ("DateTime","Second") => TrialJudgement(+0.06% => invariant)
                ("Date","Day") => TrialJudgement(+0.12% => invariant)
                ("DateTime","Year") => TrialJudgement(-8.18% => improvement)
                ("DateTime","Hour") => TrialJudgement(+2.73% => invariant)
                ("DateTime","Day") => TrialJudgement(+0.24% => invariant)
                ("DateTime","Minute") => TrialJudgement(-0.12% => invariant)
"sparse" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["array"]
        "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("unary minus",(600,600)) => TrialJudgement(-55.27% => improvement)
                ("unary minus",(20000,20000)) => TrialJudgement(+3.30% => invariant)
```

In [ ]:
```
showall(regressions(vs))
```

In [ ]:
```
showall(improvements(vs))
```

```
                        ("serialize","Matrix{Float64}") => TrialJudgement(+8.16% => regression)
                        ("serialize","Vector{String}") => TrialJudgement(-64.20% => improvement)
                "read" => 2-element BenchmarkTools.BenchmarkGroup:
                        tags: ["buffer", "stream", "string"]
                        "readstring" => TrialJudgement(-11.47% => improvement)
                        "read" => TrialJudgement(+0.01% => invariant)
        "problem" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["example", "kernel"]
                "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                        tags: ["spell", "string"]
                        "spellcheck" => TrialJudgement(-36.05% => improvement)
        "dates" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                "string" => 2-element BenchmarkTools.BenchmarkGroup:
                        tags: []
                        "Date" => TrialJudgement(-52.70% => improvement)
                        "DateTime" => TrialJudgement(-51.71% => improvement)
                "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                        tags: []
                        ("Date","Month") => TrialJudgement(-6.56% => improvement)
                        ("DateTime","Millisecond") => TrialJudgement(-0.06% => invariant)
                        ("DateTime","Month") => TrialJudgement(-7.19% => improvement)
                        ("Date","Year") => TrialJudgement(-7.57% => improvement)
                        ("DateTime","Second") => TrialJudgement(+0.06% => invariant)
                        ("Date","Day") => TrialJudgement(+0.12% => invariant)
                        ("DateTime","Year") => TrialJudgement(-8.18% => improvement)
                        ("DateTime","Hour") => TrialJudgement(+2.73% => invariant)
                        ("DateTime","Day") => TrialJudgement(+0.24% => invariant)
                        ("DateTime","Minute") => TrialJudgement(-0.12% => invariant)
        "sparse" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["array"]
                "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                        tags: []
                        ("unary minus",(600,600)) => TrialJudgement(-55.27% => improvement)
                        ("unary minus",(20000,20000)) => TrialJudgement(+3.30% => invariant)
```

In [ ]:
```
showall(regressions(vs))
```

In [ ]:
```
showall(improvements(vs))
```

```
                ("serialize",Vector{String}) => TrialJudgement(-64.20% => improvement)
        "read" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "stream", "string"]
                "readstring" => TrialJudgement(-11.47% => improvement)
                "read" => TrialJudgement(+0.01% => invariant)
"problem" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["example", "kernel"]
        "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["spell", "string"]
                "spellcheck" => TrialJudgement(-36.05% => improvement)
"dates" => 2-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "string" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                "Date" => TrialJudgement(-52.70% => improvement)
                "DateTime" => TrialJudgement(-51.71% => improvement)
        "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("Date","Month") => TrialJudgement(-6.56% => improvement)
                ("DateTime","Millisecond") => TrialJudgement(-0.06% => invariant)
                ("DateTime","Month") => TrialJudgement(-7.19% => improvement)
                ("Date","Year") => TrialJudgement(-7.57% => improvement)
                ("DateTime","Second") => TrialJudgement(+0.06% => invariant)
                ("Date","Day") => TrialJudgement(+0.12% => invariant)
                ("DateTime","Year") => TrialJudgement(-8.18% => improvement)
                ("DateTime","Hour") => TrialJudgement(+2.73% => invariant)
                ("DateTime","Day") => TrialJudgement(+0.24% => invariant)
                ("DateTime","Minute") => TrialJudgement(-0.12% => invariant)
"sparse" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["array"]
        "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("unary minus",(600,600)) => TrialJudgement(-55.27% => improvement)
                ("unary minus",(20000,20000)) => TrialJudgement(+3.30% => invariant)
```

In [ ]:
```
showall(regressions(vs))
```

In [ ]:
```
showall(improvements(vs))
```

In [ ]:
```
id = ["io", "serialization", ("serialize","Matrix{Float64}")]
```

```
                "read" => 2-element BenchmarkTools.BenchmarkGroup:
                        tags: ["buffer", "stream", "string"]
                        "readstring" => TrialJudgement(-11.47% => improvement)
                        "read" => TrialJudgement(+0.01% => invariant)
        "problem" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["example", "kernel"]
                "spellcheck" => 1-element BenchmarkTools.BenchmarkGroup:
                        tags: ["spell", "string"]
                        "spellcheck" => TrialJudgement(-36.05% => improvement)
        "dates" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                "string" => 2-element BenchmarkTools.BenchmarkGroup:
                        tags: []
                        "Date" => TrialJudgement(-52.70% => improvement)
                        "DateTime" => TrialJudgement(-51.71% => improvement)
                "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                        tags: []
                        ("Date","Month") => TrialJudgement(-6.56% => improvement)
                        ("DateTime","Millisecond") => TrialJudgement(-0.06% => invariant)
                        ("DateTime","Month") => TrialJudgement(-7.19% => improvement)
                        ("Date","Year") => TrialJudgement(-7.57% => improvement)
                        ("DateTime","Second") => TrialJudgement(+0.06% => invariant)
                        ("Date","Day") => TrialJudgement(+0.12% => invariant)
                        ("DateTime","Year") => TrialJudgement(-8.18% => improvement)
                        ("DateTime","Hour") => TrialJudgement(+2.73% => invariant)
                        ("DateTime","Day") => TrialJudgement(+0.24% => invariant)
                        ("DateTime","Minute") => TrialJudgement(-0.12% => invariant)
        "sparse" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["array"]
                "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                        tags: []
                        ("unary minus",(600,600)) => TrialJudgement(-55.27% => improvement)
                        ("unary minus",(20000,20000)) => TrialJudgement(+3.30% => invariant)
```

In [ ]:
```
showall(regressions(vs))
```

In [ ]:
```
showall(improvements(vs))
```

In [ ]:
```
id = ["io", "serialization", ("serialize","Matrix{Float64}")]
```

```
        tags: []
        "string" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                "Date" => TrialJudgement(-52.70% => improvement)
                "DateTime" => TrialJudgement(-51.71% => improvement)
        "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("Date","Month") => TrialJudgement(-6.56% => improvement)
                ("DateTime","Millisecond") => TrialJudgement(-0.06% => invariant)
                ("DateTime","Month") => TrialJudgement(-7.19% => improvement)
                ("Date","Year") => TrialJudgement(-7.57% => improvement)
                ("DateTime","Second") => TrialJudgement(+0.06% => invariant)
                ("Date","Day") => TrialJudgement(+0.12% => invariant)
                ("DateTime","Year") => TrialJudgement(-8.18% => improvement)
                ("DateTime","Hour") => TrialJudgement(+2.73% => invariant)
                ("DateTime","Day") => TrialJudgement(+0.24% => invariant)
                ("DateTime","Minute") => TrialJudgement(-0.12% => invariant)
  "sparse" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: ["array"]
        "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                tags: []
                ("unary minus",(600,600)) => TrialJudgement(-55.27% => improvement)
                ("unary minus",(20000,20000)) => TrialJudgement(+3.30% => invariant)
```

In [7]:
```
showall(regressions(vs))
```

```
1-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "io" => 1-element BenchmarkTools.BenchmarkGroup:
        tags: []
        "serialization" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["buffer", "string"]
                ("serialize","Matrix{Float64}") => TrialJudgement(+8.16% => regression)
```

In [ ]:
```
showall(improvements(vs))
```

In [ ]:
```
id = ["io", "serialization", ("serialize","Matrix{Float64}")]
```

In [ ]:
```
t5 = r5[id]
```

```
                "string" => 2-element BenchmarkTools.BenchmarkGroup:
                        tags: []
                        "Date" => TrialJudgement(-52.70% => improvement)
                        "DateTime" => TrialJudgement(-51.71% => improvement)
                "arithmetic" => 10-element BenchmarkTools.BenchmarkGroup:
                        tags: []
                        ("Date","Month") => TrialJudgement(-6.56% => improvement)
                        ("DateTime","Millisecond") => TrialJudgement(-0.06% => invariant)
                        ("DateTime","Month") => TrialJudgement(-7.19% => improvement)
                        ("Date","Year") => TrialJudgement(-7.57% => improvement)
                        ("DateTime","Second") => TrialJudgement(+0.06% => invariant)
                        ("Date","Day") => TrialJudgement(+0.12% => invariant)
                        ("DateTime","Year") => TrialJudgement(-8.18% => improvement)
                        ("DateTime","Hour") => TrialJudgement(+2.73% => invariant)
                        ("DateTime","Day") => TrialJudgement(+0.24% => invariant)
                        ("DateTime","Minute") => TrialJudgement(-0.12% => invariant)
        "sparse" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["array"]
                "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                        tags: []
                        ("unary minus",(600,600)) => TrialJudgement(-55.27% => improvement)
                        ("unary minus",(20000,20000)) => TrialJudgement(+3.30% => invariant)
```

In [7]:
```
showall(regressions(vs))
```

```
1-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "io" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("serialize","Matrix{Float64}") => TrialJudgement(+8.16% => regression)
```

In [ ]:
```
showall(improvements(vs))
```

In [ ]:
```
id = ["io", "serialization", ("serialize","Matrix{Float64}")]
```

In [ ]:
```
t5 = r5[id]
```

```
                        ("Date","Year") => TrialJudgement(-7.57% => improvement)
                        ("DateTime","Second") => TrialJudgement(+0.06% => invariant)
                        ("Date","Day") => TrialJudgement(+0.12% => invariant)
                        ("DateTime","Year") => TrialJudgement(-8.18% => improvement)
                        ("DateTime","Hour") => TrialJudgement(+2.73% => invariant)
                        ("DateTime","Day") => TrialJudgement(+0.24% => invariant)
                        ("DateTime","Minute") => TrialJudgement(-0.12% => invariant)
  "sparse" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["array"]
          "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  ("unary minus",(600,600)) => TrialJudgement(-55.27% => improvement)
                  ("unary minus",(20000,20000)) => TrialJudgement(+3.30% => invariant)
```

In [7]:
```
showall(regressions(vs))
```

```
1-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "io" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("serialize","Matrix{Float64}") => TrialJudgement(+8.16% => regression)
```

In [ ]:
```
showall(improvements(vs))
```

In [ ]:
```
id = ["io", "serialization", ("serialize","Matrix{Float64}")]
```

In [ ]:
```
t5 = r5[id]
```

In [ ]:
```
t6 = r6[id]
```

In [ ]:
```
plotkde([t6, t5])
```

In [ ]:

```
                    ("DateTime","Hour") => TrialJudgement(+2.73% => invariant)
                    ("DateTime","Day") => TrialJudgement(+0.24% => invariant)
                    ("DateTime","Minute") => TrialJudgement(-0.12% => invariant)
  "sparse" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: ["array"]
          "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                  tags: []
                  ("unary minus",(600,600)) => TrialJudgement(-55.27% => improvement)
                  ("unary minus",(20000,20000)) => TrialJudgement(+3.30% => invariant)
```

In [7]:
```
showall(regressions(vs))
```

```
1-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "io" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("serialize","Matrix{Float64}") => TrialJudgement(+8.16% => regression)
```

In [ ]:
```
showall(improvements(vs))
```

In [8]:
```
id = ["io", "serialization", ("serialize","Matrix{Float64}")]
```

Out[8]:
```
3-element Array{Any,1}:
 "io"
 "serialization"
 ("serialize","Matrix{Float64}")
```

In [ ]:
```
t5 = r5[id]
```

In [ ]:
```
t6 = r6[id]
```

In [ ]:
```
plotkde([t6, t5])
```

In [ ]:

```
                ("DateTime","Minute") => TrialJudgement(-0.12% => invariant)
        "sparse" => 1-element BenchmarkTools.BenchmarkGroup:
                tags: ["array"]
                "arithmetic" => 2-element BenchmarkTools.BenchmarkGroup:
                        tags: []
                        ("unary minus",(600,600)) => TrialJudgement(-55.27% => improvement)
                        ("unary minus",(20000,20000)) => TrialJudgement(+3.30% => invariant)
```

In [7]:
```
showall(regressions(vs))
```

```
1-element BenchmarkTools.BenchmarkGroup:
  tags: []
  "io" => 1-element BenchmarkTools.BenchmarkGroup:
          tags: []
          "serialization" => 1-element BenchmarkTools.BenchmarkGroup:
                  tags: ["buffer", "string"]
                  ("serialize","Matrix{Float64}") => TrialJudgement(+8.16% => regression)
```

In [ ]:
```
showall(improvements(vs))
```

In [8]:
```
id = ["io", "serialization", ("serialize","Matrix{Float64}")]
```

Out[8]:
```
3-element Array{Any,1}:
 "io"
 "serialization"
 ("serialize","Matrix{Float64}")
```

In [*]:
```
t5 = r5[id]
```

In [ ]:
```
t6 = r6[id]
```

In [ ]:
```
plotkde([t6, t5])
```

In [ ]:

```
                "serialization" => 1-element BenchmarkTools.BenchmarkGroup:
                        tags: ["buffer", "string"]
                        ("serialize","Matrix{Float64}") => TrialJudgement(+8.16% => regression)
```

In [ ]:
```
showall(improvements(vs))
```

In [8]:
```
id = ["io", "serialization", ("serialize","Matrix{Float64}")]
```

Out[8]:
```
3-element Array{Any,1}:
 "io"
 "serialization"
 ("serialize","Matrix{Float64}")
```

In [9]:
```
t5 = r5[id]
```

Out[9]:
```
BenchmarkTools.Trial:
  memory estimate:  8.00 MiB
  allocs estimate:  11
  --------------
  minimum time:     4.733 ms (0.00% GC)
  median time:      4.911 ms (0.00% GC)
  mean time:        6.513 ms (24.42% GC)
  maximum time:     14.084 ms (40.02% GC)
  --------------
  samples:          763
  evals/sample:     1
```

In [ ]:
```
t6 = r6[id]
```

In [ ]:
```
plotkde([t6, t5])
```

In [ ]:

```
In [ ]: showall(improvements(vs))
```

```
In [8]: id = ["io", "serialization", ("serialize","Matrix{Float64}")]
```

```
Out[8]: 3-element Array{Any,1}:
         "io"
         "serialization"
         ("serialize","Matrix{Float64}")
```

```
In [9]: t5 = r5[id]
```

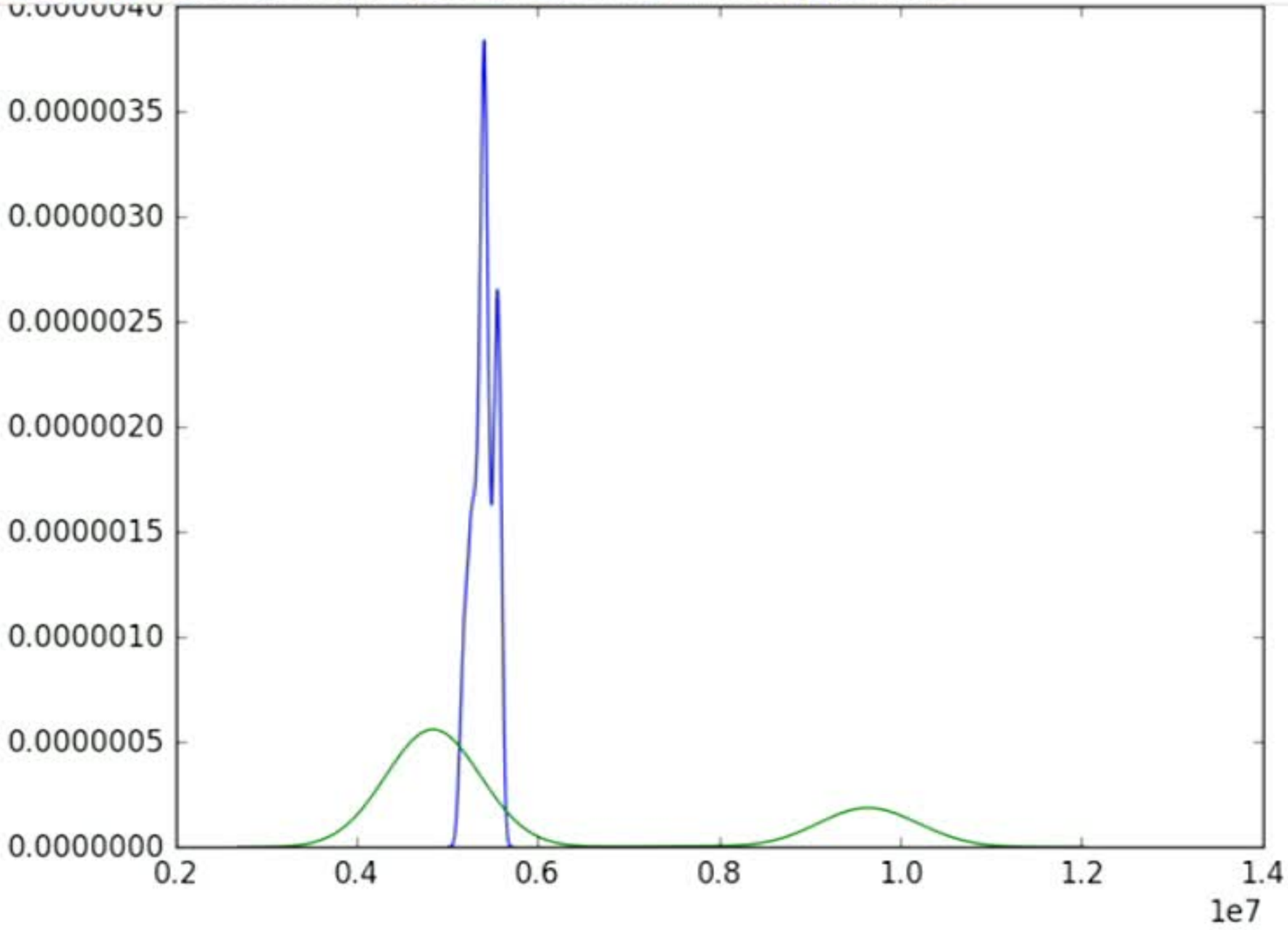
```
Out[9]: BenchmarkTools.Trial:
          memory estimate:  8.00 MiB
          allocs estimate:  11
          --------------
          minimum time:     4.733 ms (0.00% GC)
          median time:      4.911 ms (0.00% GC)
          mean time:        6.513 ms (24.42% GC)
          maximum time:     14.084 ms (40.02% GC)
          --------------
          samples:          763
          evals/sample:     1
```

```
In [10]: t6 = r6[id]
```

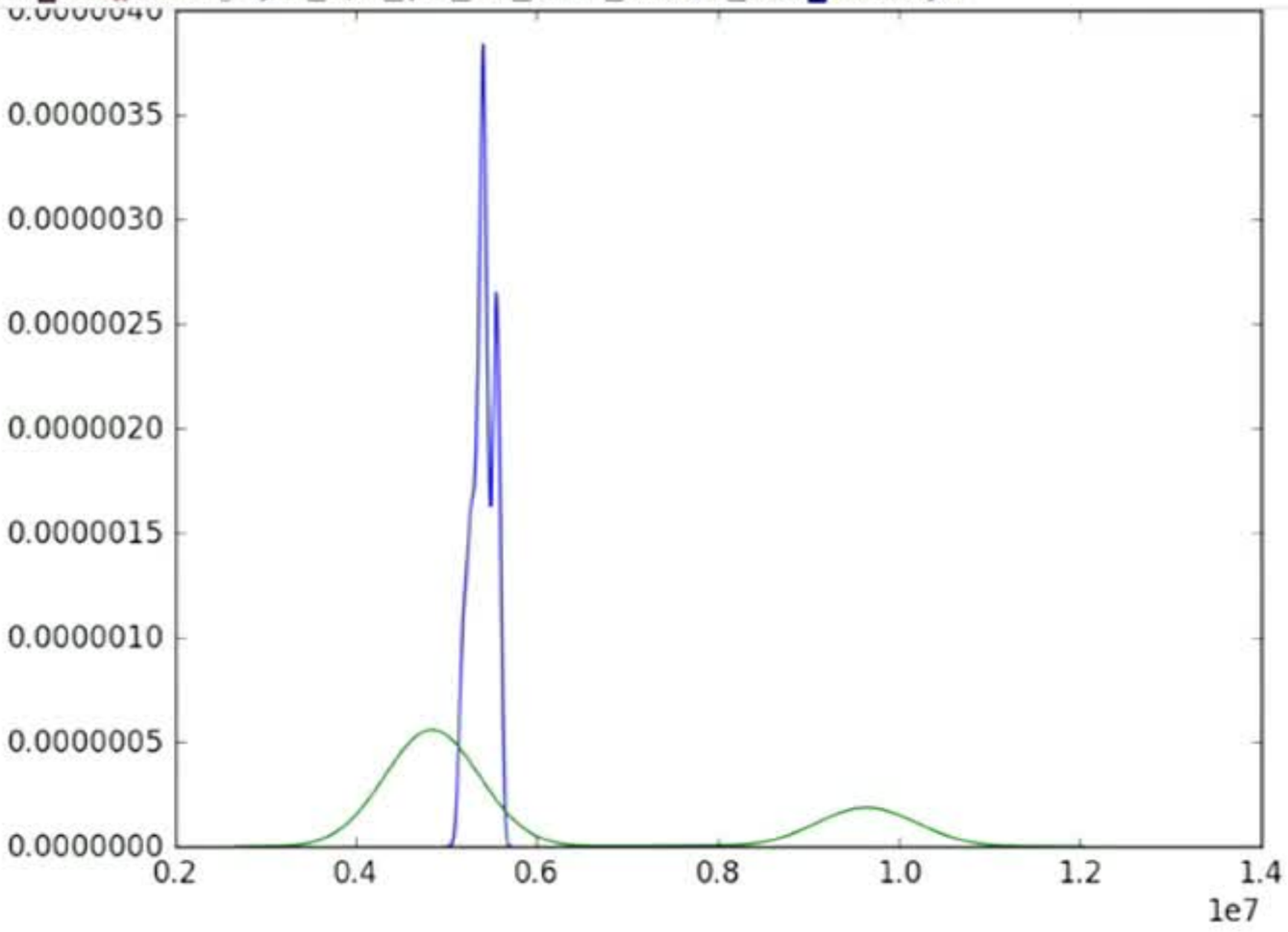
```
Out[10]: BenchmarkTools.Trial:
           memory estimate:  8.00 MiB
           allocs estimate:  6
           --------------
           minimum time:     5.120 ms (0.00% GC)
           median time:      5.417 ms (0.00% GC)
           mean time:        5.433 ms (0.00% GC)
           maximum time:     8.367 ms (0.00% GC)
           --------------
           samples:          698
           evals/sample:     1
```

```
In [ ]: plotkde([t6, t5])
```

```
In [ ]:
```



In [ ]:


0.0000040
0.0000035
0.0000030
0.0000025
0.0000020
0.0000015
0.0000010
0.0000005
0.0000000
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1e7

```
In [ ]:
```

Chrome File Edit View History Bookmarks People Window Help

Wed Mar 1 10:18 AM

codegen support for efficient

localhost:8888/notebooks/BenchmarkToolsTutorial.ipynb

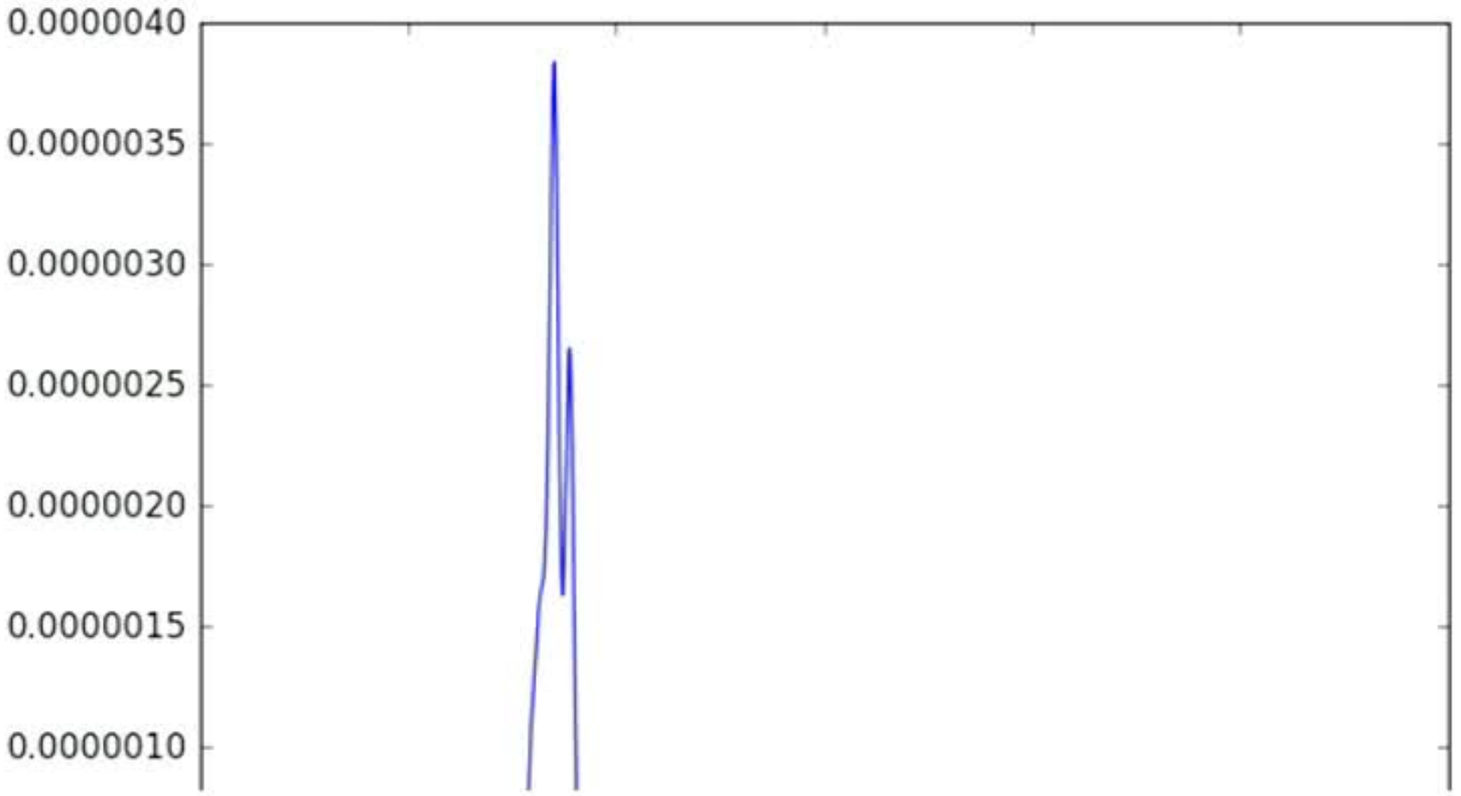
Gmail Google Drive Google Calendar Slack Discourse PkgEval repos julia mit random benchmarks sound SIAM CSE Program Other Bookmarks

File Edit View Insert Cell Kernel Widgets Help

Julia 0.5.1-pre

```
  mean time:          5.433 ms (0.00% GC)
  maximum time:       8.367 ms (0.00% GC)
  --------------
  samples:            698
  evals/sample:       1
```

In [11]:
```
plotkde([t6, t5])
```



king

log

Well, I'd like to do some benchmarking myself.

Who trusts a bot anyway?

Well, I'd like to do some benchmarking myself.

Who trusts a bot anyway?

What We Didn't Cover

- Most of BenchmarkTools (https://github.com/JuliaCI/BenchmarkTools.jl)
- Most of BaseBenchmarks (https://github.com/JuliaCI/BaseBenchmarks.jl)
- Rigorous statistical treatments of timing distributions
  - **Robust benchmarking in noisy environments**. Jiahao Chen, Jarrett Revels, Alan Edelman (2016), *Proceedings of the 20th Annual IEEE High Performance Extreme Computing Conference*, https://arxiv.org/abs/1608.04295.
  - **Coz: Finding Code that Counts with Causal Profiling**. Charlie Curtsinger, Emery D. Berger (2015), *Proceedings of the 25th Symposium on Operating Principles*.
- Julia's built-in profiling utilities (http://docs.julialang.org/en/latest/manual/profile.html#Profiling-1)

Join us on the Julia Discourse!

https://discourse.julialang.org/

Join us on the Julia Discourse!

https://discourse.julialang.org/