```
begin
    import Pkg
    Pkg.activate(".")
    using Bootstrap, Statistics, JLD2, SpinTruncatedWigner
    include("../src/numerics.jl")
    import .Numerics
end
```

Another implementation of Pair renyi entropy

This implementation is based on the idea to first extract all of the required correlators from every cTWA state vector and then use these to compute the Renyi entropy. This has the following advantages:

1. Each correlator is computed only once. The current implementation computes the single-spin expectation values repeatedly for every pair (so a total of $\binom{N}{2}$ times)
2. Flexibility: If we just want to compute the Renyi entropy, we don't need to materialize all of the intermediate vector of correlators simultaneously and reuse the memory instead. On the other hand, if we wanted to do statistical bootstrapping, we can compute all correlators once and thus drastically reduce the overhead for each subsequent computation of Renyi entropies.
3. We found another good speedup by forcing inlining of the 3-arg `lookupClusterOp`. which Julia did not do automatically. This reduced the number of allocations to just the ones for the return vector. With this optimization this code is faster than the one in `numerics.jl`

Implementation

```
prepare_vector! (generic function with 1 method)
```

```
begin
function prepare_vector(cb, state)
    N = length(cb.clusters)
    total_length = 3*N + binomial(N,2)*9 # ∑_k (N choose k) 3^k
    ret = zeros(total_length)
    return prepare_vector!(ret, cb, state)
end
function prepare_vector!(ret, cb, state)
    N = length(cb.clusters)
    # ret stores all single and two spin values
    # -> [1:3N] contains the single spin values,
    #    where [1:3] are [X,Y,Z] of spin 1, [4:6] of spin 2 and so on
    # -> [3N+1:end] contains the correlators of spins (x,y) in lexicographic
    order
    #    so the correlators of spins (i,j) are offset by Δ=(i-1)N - i(i+1)/2 + j
    at
    #    [3N+9Δ-8:3N+9Δ]
    # note that the code below stores the 2 spin correlators in different orders
    # within the respective blocks depending on whether the spins form a cluster.
    # For computing the renyi entropy this does not matter.
    for i in 1:N
        ret[3i-2] = state[lookupClusterOp(cb, (i, 1))]
        ret[3i-1] = state[lookupClusterOp(cb, (i, 2))]
        ret[3i-0] = state[lookupClusterOp(cb, (i, 3))]
    end
    offset = 3N+1
    for i in 1:N
        for j in i+1:N
            # Δ = (i-1)*N - binomial(i+1,2) + j
            # offset = 3N+9Δ-8
            if sameCluster(cb, i, j)
                for d1 in 1:3
                    for d2 in 1:3
                        ret[offset] = state[@inline lookupClusterOp(cb, (i, d1),
                        (j, d2))]
                        offset += 1
                    end
                end
            else
                kron!(@view(ret[offset:offset+8]),
                    @view(ret[3i-2:3i]),
                    @view(ret[3j-2:3j]))
                offset += 9
            end
        end
    end
    return ret
end
end
```

```
const sameCluster = sameCluster (generic function with 1 method)
```

```
const sameCluster = SpinTruncatedWigner.sameCluster
```

```
all_pair_renyi_prepared! (generic function with 1 method)
```

```
begin
    function all_pair_renyi_prepared(states, N)
        all_pair_renyi_prepared!(similar(states[1]), states, N)
    end
    function all_pair_renyi_prepared!(means, states, N)
        # means = mean(states)
        fill!(means, zero(eltype(means)))
        for state in states
            means .+= state
        end
        means ./= length(states)
        means .^= 2
        renyi = 0
        for i in 1:N
            for j in i+1:N
                temp = 1
                temp += means[3i-2]
                temp += means[3i-1]
                temp += means[3i-0]
                temp += means[3j-2]
                temp += means[3j-1]
                temp += means[3j-0]

                Δ = (i-1)*N - binomial(i+1,2) + j
                offset = 3N+9Δ-8
                temp += sum(@view means[offset:offset+8])
                renyi += log2(temp)
            end
        end
        return 2 - renyi/binomial(N,2)
    end
end
```

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

```
function all_pair_renyi_ctwa(clusterbasis, states)
    tmp1 = prepare_vector(clusterbasis, states[1])
    tmp2 = similar(tmp1)
    # note that this looks a bit like dangerous reuse of memory
    # it works reliably because all_pair_renyi_prepared!
    # only accesses the prepared states sequentially
    return all_pair_renyi_prepared!(
        tmp1,
        (prepare_vector!(tmp2,clusterbasis,s) for s in states),
        length(clusterbasis.clusters))
end
```

Consistency checks

```
shouldtest = true
```

```
# only execute tests in notebook mode or if requested
shouldtest = @isdefined(PlutoRunner) || in("--perform-test", ARGS)
```

```
Dict("clustersize" ⇒ 2, "Δ" ⇒ 0, "N" ⇒ 16, "tlist" ⇒ 0.0:0.02:10.0, "alg" ⇒ "dcTWA
```

```
if shouldtest
    testdata = JLD2.load(readdir(joinpath(@__DIR__, "../data/fulldata-
    simulations"), join=true)[1])
end
```

```
@info "" shouldtest
```

```
shouldtest: true
```

```
shouldtest && let results = testdata["results"][1],
    cb = ClusterBasis(results.clusters)
    states = results.fulldata[1]

    # precompile
    all_pair_renyi_ctwa(cb, states[1:2])
    Numerics.all_pair_renyi_ctwa(cb, states[1:2])

    # run to compare
    @info @time "New Implementation" all_pair_renyi_ctwa(cb, states)
    @info @time "From numerics.jl  " Numerics.all_pair_renyi_ctwa(cb, states)
    @info @time "New Implementation" all_pair_renyi_ctwa(cb, results.fulldata[5])
    @info @time "From numerics.jl  " Numerics.all_pair_renyi_ctwa(cb,
    results.fulldata[5])
end
```

```
-0.00045567058527540283
```

```
-0.00045567058527562856
```

```
0.004862598337452839
```

```
0.004862598337452299
```

```
New Implementation: 0.026655 seconds (4 allocations: 17.938 KiB)
From numerics.jl  : 0.137414 seconds (842 allocations: 51.688 KiB)
New Implementation: 0.027702 seconds (4 allocations: 17.938 KiB)
From numerics.jl  : 0.124215 seconds (842 allocations: 51.688 KiB)
```

Data Evaluation

We perform the following statistical analysis:

Staggered Magentization

We just estimate the Monte-Carlo shot noise by computing the standard deviation for each time point over all trajectories.

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

```
function analyze_magnetization(results)
    cb = ClusterBasis(results.clusters)
    mags = Numerics.staggered_magnetization.(Ref(cb), results.fulldata)
    (; magnetization_std = std.(mags), magnetization_mean = mean.(mags))
end
```

Pair Renyi entropy

We particition the 10,000 shots into chunks and compute the Renyi entropy for each of the chunks. From this we estimate a mean and a variance. We repeat this for different sizes of the chunks.

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

```
function analyze_renyi(results, chunksize)
    cb = ClusterBasis(results.clusters)
    nshots = length(results.fulldata[1])
    !iszero(nshots % chunksize) && @warn "Chunksize $chunksize does not evenly divide the number of shots $nshots"
    chunks = Iterators.partition(1:nshots, chunksize)
    entropies = [[all_pair_renyi_ctwa(cb, states[chunk]) for chunk in chunks] for states in results.fulldata] # [t][chunk]
    (; renyi_std = std.(entropies), renyi_mean = mean.(entropies))
end
```

Analysis function

```
analyze (generic function with 2 methods)
```

```
function analyze(file, renyi_chunksizes=[50,100,200,250,500,1000])
    data = JLD2.load(file)
    results = data["results"][1]

    (; magnetization_std, magnetization_mean) = analyze_magnetization(results)
    renyi_stds = Vector{Float64}[]
    renyi_means = Vector{Float64}[]
    for chunksize in renyi_chunksizes
        (; renyi_std, renyi_mean) = analyze_renyi(results, chunksize)
        push!(renyi_stds, renyi_std)
        push!(renyi_means, renyi_mean)
    end
    new_results = (; renyi_stds, renyi_means, renyi_chunksizes,
                     magnetization_std, magnetization_mean)
    data["results"] = [new_results]

    filename = basename(file)
    dir = dirname(file)
    savedir = joinpath(dir, "..", basename(dir)*"-avg")
    mkpath(savedir)
    JLD2.save(joinpath(savedir, filename), data)
end
```

Perform the analysis

```
let files = readdir(joinpath(@__DIR__, "../data/fulldata-simulations"),
join=true)
    @info "Found $(length(files)) files"
    for file in files
        @info "Doing $(basename(file))"
        @time analyze(file)
    end
end
```

```
Found 4 files

Doing N=16_alg=dcTWA_chunkID=1_chunksize=1_clustering=RG_clustersize=2_fillin
g=0.1_fulldata=true_trajectories=10000_Δ=0_α=0.5.jld2

Doing N=16_alg=dcTWA_chunkID=1_chunksize=1_clustering=RG_clustersize=4_fillin
g=0.1_fulldata=true_trajectories=10000_Δ=0_α=0.5.jld2

Doing N=16_alg=gcTWA_chunkID=1_chunksize=1_clustering=RG_clustersize=2_fillin
g=0.1_fulldata=true_trajectories=10000_Δ=0_α=0.5.jld2

Doing N=16_alg=gcTWA_chunkID=1_chunksize=1_clustering=RG_clustersize=4_fillin
g=0.1_fulldata=true_trajectories=10000_Δ=0_α=0.5.jld2
```

```
122.908813 seconds (76.35 M allocations: 12.791 GiB, 11.40% gc time, 0.5
1% compilation time: 10% of which was recompilation)
129.928121 seconds (75.80 M allocations: 12.755 GiB, 11.83% gc time)
129.818630 seconds (75.80 M allocations: 12.755 GiB, 12.20% gc time)
130.541347 seconds (75.80 M allocations: 12.755 GiB, 12.39% gc time)
```