Using @benchmark on unoptimized code:

BenchmarkTools.Trial: 1 sample with 1 evaluation.

Single result which took 8.457 s (4.78% GC) to evaluate,

with a memory estimate of 16.25 GiB, over 1\_366\_823 allocations.

# Changes to tempTesting.jl

* Changed scan1 – scan2 to use broadcasting. scan1 .- scan 2

# Changes to MERIT.jl

## Strictly typing the points vector in domain\_hemisphere

Before changes:

A screen shot of a computer

Description automatically generated

Changes:

Changed the vector of points from [[], [], []] to Vector[Float64[], Float64[], Float64[]]

After changes:

A screenshot of a computer

Description automatically generated

## Removing casting to Float64 at the end

Before Changes

A screenshot of a computer

Description automatically generated

Change:

Removing call to Float64.() at the end of domain\_hemisphere.

After Change:

A screenshot of a computer

Description automatically generated

# Changes to MERIT.Beamform

(This is running with optimized MERIT.domain\_hemisphere as well)

## Change most operations to use broadcasting

Before change:

A screenshot of a computer

Description automatically generated

Change:

At the end where “time” is calculated, I changed the addition and subtraction to use broadcasting.

After Change:

A screenshot of a computer

Description automatically generated

## Adding Base.sqrt\_llvm

I read a forum post that said using Base.sqrt\_llvm was quicker than the Base.sqrt() implementation since sqrt\_llvm doesn’t check for negative numbers and so it can be SIMD’d. Forum post: <https://discourse.julialang.org/t/sqrt-abs-x-is-even-faster-than-sqrt/58154/4>

However in my testing, I didn’t notice any timing differences. But still left it in there since functionally it is similar to sqrt() and I know there won’t be negative numbers in this block of code.

## Using Broadcasting when equating to a variable

Before change:

A screenshot of a computer

Description automatically generated

Change:

Used the broadcasted version of equating (i.e. .= instead of =) in distances and time. In order to do this I had to pre-allocate the distances and time matrices.

After change:

A screenshot of a computer

Description automatically generated

## Using @inbounds in distance calculations

Before change:

A screenshot of a computer

Description automatically generated

Change:

In the for loop I added the @inbounds macro to stop Julia from checking if the index is in the bounds of the array.

After change:

A screenshot of a computer

Description automatically generated

# Changes to Process.delay\_signal

(I couldn’t test it by itself, so I had to estimate the benefits via testing the beamforming function).

## Broadcasting the operations in delay\_signal

Before change:

A screenshot of a computer

Description automatically generated

Change:

Broadcasting the scalar operations in delay\_signal

After change:

A screenshot of a computer

Description automatically generated

## Strictly Typing the delay\_signal function

Before Change:

A screenshot of a computer

Description automatically generated

Change:

I strictly typed the delay\_signal function. This means I can get rid of the isreal() if check. This also means I have to implement a delay\_signal function for the time domain.

After Change:

A screenshot of a computer

Description automatically generated

# Changes to Beamformer.DAS

## Remove call to permutedims

Before Change:

A screenshot of a computer

Description automatically generated

Change:

Removing the call to permutedims in the beamformer.DAS

After Change:

A screenshot of a computer

Description automatically generated

# Changes to Beamform.beamform

# Ideas

Maybe use @inbounds when accessing arrays in for loops to skip the inbounds check?