**PhD Work**

**Week 20-28 April**

**Summary**

1. Syntax and use of StaticArrays (LAB1)
2. Benchmark svector, mvector, and normal vector in fetch and update operations

-fetch (LABs 2-3-4-5)

-update (LABs 6-7-8)

1. Unknown size problem...where{N} (LAB9)
2. Macros and generated functions definition and benchmarking (LABs 10-11-12)
3. Unknown size problem revisited with macros (LAB 13)
4. qs\_solver prototype

-simplify design by deleting some structs and files

-use of immutable structs and static arrays

-compute derivative(fixed allocation error, testing use of transpose)

-added SAVEat feature

-benchmarking

-analytic solution

**Detailed report**

**I. StaticArrays Syntax and benchmarking**

**1. Hard-code size-known creation:**

**LAB1: Create a 3-element SVector{3, Int64} # or float64**

1. **Svector**

v0=**SA**[1,2,3]

v1 = **SVector**(1,2,3)

v2 = **@SVector** [1,2,3]

v3=**SVector**{3,Int}(1,2,3)

vect=(1,2,3) or [1,2,3]

v4=**SVector**{3,Int}(vect)

v5 = **@SVector** **zeros**(3) #Float64

v6 = **@SVector** **randn**(Float64, 40)

Access it via V[1];1

setIndex! Is illegal. Setindex Modification creates a new svector. v5=**setindex**(v5,10,2)

1. **Mvector**

Same as Svector change S to M except **SA**[1,2,3].

1. **Matrices**

2×2 SMatrix{2, 2, Int64, 4}

m1 = **SMatrix**{2,2}(1, 2, 3, 4)

m2 = **SMatrix**{2,2}([1 3 ; 2 4])

m3 = **@SMatrix**[1 2;3 4]

m4 = zeros(3,3)

#SMatrix(m4) # this is an error

SMatrix{3,3}(m4) # correct - size is inferrable

SArray{Tuple{3,3}}(m4) # correct, note Tuple{3,3}

**2. Benchmark fetch and update of size-known created Svectors and Mvectors:**

In the following experiments, we will vary the type of the operation (fetch, update), the number of states (5,10,100), the number of times (1e+3,1e+6) of the operations, and we will measure time and memory allocation. We will also check the effect of random values and random access.

1. **Fetch Operation**

**LAB2: compare stack-fetch vs heap-fetch via Svectors,vectors,Mvectors with 5 state variables of given values.**

|  |  |  |
| --- | --- | --- |
| function **simulatorSvector**()  s=0.0  v1=**@SVector**[1.1,2.2,3.3,4.4,5.5]  for i = 1:1e+3 for j = 1:5  s=s+v1[j]  end end  s  end | function **simulatorvector**()  s=0.0  v1 = [1.1,2.2,3.3,4.4,5.5]  for i = 1:1e+3 for j = 1:5  s=s+v1[j]  end end  s  end | function **simulatorMvector**()  s=0.0  v1=**@MVector**[1.1,2.2,3.3,4.4,5.5]  for i = 1:1e+3 for j = 1:5  s=s+v1[j]  end end  s  end |

**@btime** **simulatorSvector**()**@btime** **simulatorvector**()**@btime** **simulatorMvector**()

4.039 μs (0 allocations: 0 bytes)

4.089 μs (1 allocation: 96 bytes)

4.043 μs (0 allocations: 0 bytes)

**LAB3: repeated LAB1 with states variables of random values.**

v1 = **@SVector** **rand**(Float64, 5)

v2 = **rand**(5)

v3 = **@MVector** **rand**(Float64, 5)

For state variables=5

4.067 μs (0 allocations: 0 bytes)

4.107 μs (1 allocation: 96 bytes)

4.080 μs (1 allocation: 48 bytes) ——> rand caused an allocation for M

For state variables=100

81.604 μs (0 allocations: 0 bytes)

81.623 μs (1 allocation: 896 bytes)

81.626 μs (1 allocation: 816 bytes) ———->allocation increases with states

**LAB4: repeated LAB3 with random access of the vectors.**

n=**rand**(1:100) ; s=s+v1[n]

For state variables=100

610.482 μs (0 allocations: 0 bytes)

605.228 μs (1 allocation: 896 bytes)

603.799 μs (1 allocation: 816 bytes)——->random access affected only time

For state variables=100 and i=:1e+6

84.996 ms (0 allocations: 0 bytes)

85.259 ms (1 allocation: 896 bytes)

84.984 ms (1 allocation: 816 bytes) —->number\_of\_operations affected only time

**LAB5: repeated LAB3 while taking rand values from outside**; create the vector outside.

For state variables=100

600.086 μs (0 allocations: 0 bytes)

602.160 μs (0 allocations: 0 bytes)

601.399 μs (0 allocations: 0 bytes)

Conclusion for fetch:

ComputeTime:is affected by all factors: Number\_of\_operations -Random access- Number of states.

-The 3 types of vectors take the same time in Fetch operations

Allocation:is affected by only the number of states for normal and Mvectors.

-Rand values caused allocation for M

–no allocation if vectors created outside

—->Svector does not win in compute time for fetch-op!!!!!.

—->Svector wins in allocation for fetch-op only if vectors created inside.

1. **Recreate and update operations**

**LAB6: compare stack-recreate vs heap-update via Svectors,vectors,Mvectors**

|  |  |  |
| --- | --- | --- |
| function **simulatorSvector**()  v1 = **@SVector** **zeros**(5)  for i = 1:1e+3 for j = 1:5  v1=**setindex**(v1,(i-j)\*.54,j)  end end  v1  end | function **simulatorvector**()  v1 = **Vector**{Float64}(undef, 5)  for i = 1:1e+3 for j = 1:5  v1 = (i - j) \* 0.54  end end  v1  end | function **simulatorMvector**()  v1 = **@MVector** **zeros**(5)  for i = 1:1e+3 for j = 1:5  v1 = (i - j) \* 0.54  end end  v1  end |

**@btime** **simulatorSvector**()**@btime** **simulatorvector**()**@btime** **simulatorMvector**()

For state variables=5

12.344 μs (0 allocations: 0 bytes)

4.821 μs (1 allocation: 96 bytes)

3.383 μs (1 allocation: 48 bytes)—->time changes for types of vectors

For state variables=10

27.651 μs (0 allocations: 0 bytes)

7.725 μs (1 allocation: 144 bytes)

4.226 μs (1 allocation: 96 bytes)

For state variables=100

1.644 ms (0 allocations: 0 bytes)

89.728 μs (1 allocation: 896 bytes)

88.843 μs (1 allocation: 816 bytes)———->allocation increases with states for noraml and Mvectors.

**LAB7: repeated LAB6 with rand values and random access**

|  |  |  |
| --- | --- | --- |
| function **simulatorSvector**()  v1=**@SVector** **rand**(Float64,10)  for i = 1:1e+3  for j = 1:10  n=**rand**(1:10)  v1=**setindex**(v1,(i-j)\*0.54,n)  end  end  v1  end | function **simulatorvector**()  v1 = **rand**(10)  for i = 1:1e+3  for j = 1:10  n=**rand**(1:10)  v1[n] = (i - j) \* 0.54  end  end  v1  end | function **simulatorMvector**()  v1=**@MVector** **rand**(Float64,10)  for i = 1:1e+3  for j = 1:10  n=**rand**(1:10)  v1[n] = (i - j)\*0.54  end  end  v1  end |

For state variables=5

47.833 μs (0 allocations: 0 bytes)

32.548 μs (1 allocation: 96 bytes)

32.254 μs (1 allocation: 48 bytes)

For state variables=10

90.302 μs (0 allocations: 0 bytes)

62.839 μs (1 allocation: 144 bytes)

61.726 μs (1 allocation: 96 bytes)

For state variables=100

3.058 ms (0 allocations: 0 bytes)

645.668 μs (1 allocation: 896 bytes)

607.353 μs (1 allocation: 816 bytes)

For state variables=100 i=:1e+6

3.113 s (0 allocations: 0 bytes)

678.589 ms (1 allocation: 896 bytes)

634.929 ms (1 allocation: 816 bytes)

———->randomness and the number of operations only increase time

**LAB8: repeated LAB7 while taking rand values from outside**; create the vector outside.

For state variables=100

2.962 ms (0 allocations: 0 bytes)

613.865 μs (0 allocations: 0 bytes)

602.692 μs (0 allocations: 0 bytes)

Conclusion for update-recreate:

ComputeTime:is affected by all factors: Number\_of\_operations -Random access- Number of states.

-the Svector has a higher compute time.The Mvector is slightly faster than the normal vector.

Allocation:is affected by only the number of states for normal and Mvectors.

–no allocation if vectors created outside

—->Svector should not be used for update-op unless large states vectors need to be created inside.

—->Mvector is slightly better than normal vectors in allocation and compute time for update-op.

**3. Passed-size creation**

struct Data

x::SVector{N,Float64} where {N}

y::Int

end

n=2

v1 = **@SVector** **zeros**(n)

p = **Data**(**v1**, 5) #with zeros(n) gives error

LAB9

Where {N}

81.368 μs (1 allocation: 16 bytes)

81.360 μs (0 allocations: 0 bytes)

81.365 μs (1 allocation: 16 bytes)

**II. Macros and generated functions:**

**1.Macros:**

They know only the spelling, and they are executed after parsing. Why Macros? They extend the capability of the language for example when we want to construct a personalized code that takes a function as an argument and determines when the function should run. In a normal function, that passed function would execute and the argument can be returned values which we do not want. Another reason is when we want the code to be written based on what the user entered as in the following example:

macro make\_struct(struct\_name, schema...)

fields=[:($(entry.args[1])::$(entry.args[2])) for entry in schema]

esc(quote struct $struct\_name

$(fields...)

end

end)

end

@make\_struct sim (x, Int) (y, Float64) (wRegister::MVector{2,Float64}(undef)

p=sim(1,2.0)

The name and number of fields of the struct are determined by the user.

**Question1:** why do they trick the @btime? If a change a function to a macro without change the inside, the result time of @btime is very low?

Lab13: benchmark a function against a macro which returns an expression

|  |  |
| --- | --- |
| macro computeSome\_thing(states, x,index ,j)  esc(quote for $j in 1:$states  $x[$index]+=$x[$j]\*0.5 end  end)  end | function computeSome\_thing(states, x,index )  for j = 1:states  x[index] += x[j] \* 0.5  end  end |

v2 = MVector{2,Float64}(1.0,2.0)

|  |  |
| --- | --- |
| J=1  computeSome\_thing(2,v2,2,j)  v2 becomes (1.0 3.75)  211.711 ns (8 allocations: 128 bytes) | computeSome\_thing(2,v2,2)  v2 becomes (1.0 3.75)  16.299 ns (0 allocations: 0 bytes) |

If benchmark a function that calls them.

|  |  |
| --- | --- |
| function usemacro(v2::MVector{2,Float64})  j=1  for i =1:1e+1  @computeSome\_thing(2,v2,2,j)  end  end | function usefunction(v2::MVector{2,Float64})  for i =1:1e+1  computeSome\_thing(2,v2,2)  end  end |

The results:

|  |  |
| --- | --- |
| v2 becomes (1.0,200.32)  31.468 ns (0 allocations: 0 bytes) | v2 becomes (1.0,200.32)  62.380 ns (0 allocations: 0 bytes) |

**Question2:** the macro performs less if benchmarked directly and better when benchmarking a function that uses the macro?

**2.generated functions:**

They know the symbols (types not the values), and they are executed at compile time and generate/produce a more specialized code based on the types of its arguments to be run at run-time. They are used for example to do multiple dispatch. Another example is they can write a number of nested loops based on N (unknown).

**II.**