

# Optimizing DiffEq Code

Chris Rackauckas

February 25, 2019

In this notebook we will walk through some of the main tools for optimizing your code in order to efficiently solve DifferentialEquations.jl. User-side optimizations are important because, for sufficiently difficult problems, most of the time will be spent inside of your `f` function, the function you are trying to solve. "Efficient" integrators are those that reduce the required number of `f` calls to hit the error tolerance. The main ideas for optimizing your DiffEq code, or any Julia function, are the following:

- Make it non-allocating
- Use StaticArrays for small arrays
- Use broadcast fusion
- Make it type-stable
- Reduce redundant calculations
- Make use of BLAS calls
- Optimize algorithm choice

We'll discuss these strategies in the context of small and large systems. Let's start with small systems.

## 0.1 Optimizing Small Systems (<100 DEs)

Let's take the classic Lorenz system from before. Let's start by naively writing the system in its out-of-place form:

```
function lorenz(u,p,t)
    dx = 10.0*(u[2]-u[1])
    dy = u[1]*(28.0-u[3]) - u[2]
    dz = u[1]*u[2] - (8/3)*u[3]
    [dx,dy,dz]
end
```

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

Here, `lorenz` returns an object, `[dx,dy,dz]`, which is created within the body of `lorenz`.

This is a common code pattern from high-level languages like MATLAB, SciPy, or R's `deSolve`. However, the issue with this form is that it allocates a vector, `[dx,dy,dz]`, at each step. Let's benchmark the solution process with this choice of function:

```
using DifferentialEquations, BenchmarkTools
u0 = [1.0;0.0;0.0]
tspan = (0.0,100.0)
prob = ODEProblem(lorenz,u0,tspan)
@benchmark solve(prob,Tsit5())
```

```
BenchmarkTools.Trial:
  memory estimate: 10.94 MiB
  allocs estimate: 102470
  -----
  minimum time:      3.308 ms (0.00% GC)
  median time:       5.788 ms (0.00% GC)
  mean time:         6.046 ms (35.04% GC)
  maximum time:      13.045 ms (55.63% GC)
  -----
  samples:           826
  evals/sample:      1
```

The BenchmarkTools package's `@benchmark` runs the code multiple times to get an accurate measurement. The minimum time is the time it takes when your OS and other background processes aren't getting in the way. Notice that in this case it takes about 5ms to solve and allocates around 11.11 MiB. However, if we were to use this inside of a real user code we'd see a lot of time spent doing garbage collection (GC) to clean up all of the arrays we made. Even if we turn off saving we have these allocations.

```
@benchmark solve(prob,Tsit5(),save_everystep=false)
```

```
BenchmarkTools.Trial:
  memory estimate: 9.57 MiB
  allocs estimate: 89578
  -----
  minimum time:      2.925 ms (0.00% GC)
  median time:       3.852 ms (0.00% GC)
  mean time:         5.425 ms (34.65% GC)
  maximum time:      16.884 ms (64.51% GC)
  -----
  samples:           920
  evals/sample:      1
```

The problem of course is that arrays are created every time our derivative function is called. This function is called multiple times per step and is thus the main source of memory usage. To fix this, we can use the in-place form to `***make our code non-allocating***`:

```
function lorenz!(du,u,p,t)
  du[1] = 10.0*(u[2]-u[1])
```

```

du[2] = u[1]*(28.0-u[3]) - u[2]
du[3] = u[1]*u[2] - (8/3)*u[3]
end

```

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

Here, instead of creating an array each time, we utilized the cache array `du`. When the inplace form is used, `DifferentialEquations.jl` takes a different internal route that minimizes the internal allocations as well. When we benchmark this function, we will see quite a difference.

```

u0 = [1.0;0.0;0.0]
tspan = (0.0,100.0)
prob = ODEProblem(lorenz!,u0,tspan)
@benchmark solve(prob,Tsit5())

```

```

BenchmarkTools.Trial:
  memory estimate:  1.37 MiB
  allocs estimate:  12965
  -----
  minimum time:     754.699 μs (0.00% GC)
  median time:      821.100 μs (0.00% GC)
  mean time:        1.148 ms (23.22% GC)
  maximum time:     11.710 ms (84.47% GC)
  -----
  samples:          4360
  evals/sample:     1

```

```
@benchmark solve(prob,Tsit5(),save_everystep=false)
```

```

BenchmarkTools.Trial:
  memory estimate:  6.91 KiB
  allocs estimate:  93
  -----
  minimum time:     407.901 μs (0.00% GC)
  median time:      413.401 μs (0.00% GC)
  mean time:        438.401 μs (0.47% GC)
  maximum time:     7.547 ms (93.75% GC)
  -----
  samples:          10000
  evals/sample:     1

```

There is a 4x time difference just from that change! Notice there are still some allocations and this is due to the construction of the integration cache. But this doesn't scale with the problem size:

```

tspan = (0.0,500.0) # 5x longer than before
prob = ODEProblem(lorenz!,u0,tspan)
@benchmark solve(prob,Tsit5(),save_everystep=false)

```

```

BenchmarkTools.Trial:
  memory estimate:  6.91 KiB
  allocs estimate:  93
  -----
  minimum time:     2.037 ms (0.00% GC)
  median time:      2.102 ms (0.00% GC)
  mean time:        2.160 ms (0.00% GC)
  maximum time:     4.261 ms (0.00% GC)
  -----
  samples:          2319
  evals/sample:     1

```

since that's all just setup allocations.

**But if the system is small we can optimize even more.** Allocations are only expensive if they are "heap allocations". For a more in-depth definition of heap allocations, [there are a lot of sources online](#). But a good working definition is that heap allocations are variable-sized slabs of memory which have to be pointed to, and this pointer indirection costs time. Additionally, the heap has to be managed and the garbage controllers has to actively keep track of what's on the heap.

However, there's an alternative to heap allocations, known as stack allocations. The stack is statically-sized (known at compile time) and thus its accesses are quick. Additionally, the exact block of memory is known in advance by the compiler, and thus re-using the memory is cheap. This means that allocating on the stack has essentially no cost!

Arrays have to be heap allocated because their size (and thus the amount of memory they take up) is determined at runtime. But there are structures in Julia which are stack-allocated. **structs** for example are stack-allocated "value-type"s. **Tuples** are a stack-allocated collection. The most useful data structure for DiffEq though is the **StaticArray** from the package [StaticArrays.jl](#). These arrays have their length determined at compile-time. They are created using macros attached to normal array expressions, for example:

```

using StaticArrays
A = @SVector [2.0,3.0,5.0]

```

Notice that the 3 after **SVector** gives the size of the **SVector**. It cannot be changed. Additionally, **SVectors** are immutable, so we have to create a new **SVector** to change values. But remember, we don't have to worry about allocations because this data structure is stack-allocated. **SArrays** have a lot of extra optimizations as well: they have fast matrix multiplication, fast QR factorizations, etc. which directly make use of the information about the size of the array. Thus, when possible they should be used.

Unfortunately static arrays can only be used for sufficiently small arrays. After a certain size, they are forced to heap allocate after some instructions and their compile time balloons. Thus static arrays shouldn't be used if your system has more than 100 variables. Additionally, only the native Julia algorithms can fully utilize static arrays.

Let's **\*\*\*optimize `lorenz` using static arrays\*\*\***. Note that in this case, we want to use the out-of-place allocating form, but this time we want to output a static array:

```

function lorenz_static(u,p,t)
    dx = 10.0*(u[2]-u[1])
    dy = u[1]*(28.0-u[3]) - u[2]
    dz = u[1]*u[2] - (8/3)*u[3]
    @SVector [dx,dy,dz]
end

```

lorenz\_static (generic function with 1 method)

To make the solver internally use static arrays, we simply give it a static array as the initial condition:

```

u0 = @SVector [1.0,0.0,0.0]
tspan = (0.0,100.0)
prob = ODEProblem(lorenz_static,u0,tspan)
@benchmark solve(prob,Tsit5())

```

```

BenchmarkTools.Trial:
  memory estimate: 472.06 KiB
  allocs estimate: 2663
  -----
  minimum time:      435.799 μs (0.00% GC)
  median time:       451.300 μs (0.00% GC)
  mean time:         526.177 μs (10.47% GC)
  maximum time:      4.200 ms (88.99% GC)
  -----
  samples:           9465
  evals/sample:      1

```

```

@benchmark solve(prob,Tsit5(),save_everystep=false)

```

```

BenchmarkTools.Trial:
  memory estimate: 6.20 KiB
  allocs estimate: 74
  -----
  minimum time:      339.599 μs (0.00% GC)
  median time:       359.400 μs (0.00% GC)
  mean time:         368.405 μs (0.39% GC)
  maximum time:      9.089 ms (95.64% GC)
  -----
  samples:           10000
  evals/sample:      1

```

And that's pretty much all there is to it. With static arrays you don't have to worry about allocating, so use operations like `*` and don't worry about fusing operations (discussed in the next section). Do "the vectorized code" of R/MATLAB/Python and your code in this case will be fast, or directly use the numbers/values.

**Exercise 1** Implement the out-of-place array, in-place array, and out-of-place static array forms for the [Henon-Heiles System](#) and time the results.

## 0.2 Optimizing Large Systems

### 0.2.1 Interlude: Managing Allocations with Broadcast Fusion

When your system is sufficiently large, or you have to make use of a non-native Julia algorithm, you have to make use of `Arrays`. In order to use arrays in the most efficient manner, you need to be careful about temporary allocations. Vectorized calculations naturally have plenty of temporary array allocations. This is because a vectorized calculation outputs a vector. Thus:

```
A = rand(1000,1000); B = rand(1000,1000); C = rand(1000,1000)
test(A,B,C) = A + B + C
@benchmark test(A,B,C)
```

```
BenchmarkTools.Trial:
  memory estimate:  7.63 MiB
  allocs estimate:  3
  -----
  minimum time:     3.369 ms (0.00% GC)
  median time:      4.138 ms (0.00% GC)
  mean time:        5.564 ms (27.97% GC)
  maximum time:     14.827 ms (46.98% GC)
  -----
  samples:          897
  evals/sample:     1
```

That expression `A + B + C` creates 2 arrays. It first creates one for the output of `A + B`, then uses that result array to `+` `C` to get the final result. 2 arrays! We don't want that! The first thing to do to fix this is to use broadcast fusion. [Broadcast fusion](#) puts expressions together. For example, instead of doing the `+` operations separately, if we were to add them all at the same time, then we would only have a single array that's created. For example:

```
test2(A,B,C) = map((a,b,c)->a+b+c,A,B,C)
@benchmark test2(A,B,C)
```

```
BenchmarkTools.Trial:
  memory estimate:  7.63 MiB
  allocs estimate:  5
  -----
  minimum time:     3.987 ms (0.00% GC)
  median time:      4.498 ms (0.00% GC)
  mean time:        5.709 ms (24.30% GC)
  maximum time:     10.332 ms (53.70% GC)
  -----
  samples:          874
  evals/sample:     1
```

Puts the whole expression into a single function call, and thus only one array is required to store output. This is the same as writing the loop:

```
function test3(A,B,C)
    D = similar(A)
    @inbounds for i in eachindex(A)
        D[i] = A[i] + B[i] + C[i]
    end
    D
end
@benchmark test3(A,B,C)
```

```
BenchmarkTools.Trial:
 memory estimate:  7.63 MiB
 allocs estimate:  2
-----
 minimum time:      3.333 ms (0.00% GC)
 median time:      3.753 ms (0.00% GC)
 mean time:        5.059 ms (26.94% GC)
 maximum time:     20.414 ms (59.43% GC)
-----
 samples:          987
 evals/sample:     1
```

However, Julia's broadcast is syntactic sugar for this. If multiple expressions have a `.`, then it will put those vectorized operations together. Thus:

```
test4(A,B,C) = A .+ B .+ C
@benchmark test4(A,B,C)
```

```
BenchmarkTools.Trial:
 memory estimate:  7.63 MiB
 allocs estimate:  2
-----
 minimum time:      3.328 ms (0.00% GC)
 median time:      3.698 ms (0.00% GC)
 mean time:        4.958 ms (26.40% GC)
 maximum time:     10.233 ms (51.57% GC)
-----
 samples:          1008
 evals/sample:     1
```

is a version with only 1 array created (the output). Note that `.s` can be used with function calls as well:

```
sin.(A) .+ sin.(B)
```

```
1000×1000 Array{Float64,2}:
 0.651846  0.641477  1.17453  0.961972  ...  0.423389  1.03704  0.353506
 0.413598  0.967687  1.51906  0.36517   ...  1.62331  0.475489  1.27316
 1.19399   1.54681   1.04661  0.60026   ...  0.646375  1.06193  1.32467
 1.10194   1.22665   0.47539  0.713723  ...  1.54302  0.747526  0.393429
 1.04671   0.437732  1.41167  1.2242    ...  0.74384  0.263407  1.29863
 1.47489   0.659092  1.33652  0.564608  ...  0.788238  0.698497  1.22159
```

1.36098	1.05619	0.752379	1.17676	1.41901	1.02144	0.851331
1.28964	1.3207	0.432539	0.3626	0.173344	0.995353	0.743072
0.989818	0.840008	1.34968	1.04864	0.360081	1.0726	1.06121
0.897717	1.05401	1.1503	0.39823	1.09474	1.21397	0.850303
⋮				⋮		
0.095436	0.756713	1.14063	1.01952	0.246396	1.42154	0.71583
1.02863	0.781688	0.839075	1.51496	1.15294	0.695957	0.76151
0.694103	0.856707	1.34948	0.292275	0.941059	1.48455	1.35747
1.27423	1.05224	1.49442	1.46241	1.59674	1.4533	0.624528
0.833177	1.06741	1.44671	1.49967	... 0.793888	0.97331	1.65561
0.41676	0.78679	0.979136	1.50348	0.893033	1.40242	0.504704
1.06365	0.852093	1.06795	0.838078	0.88784	0.834814	0.769808
0.711148	0.596256	0.679674	1.2792	0.728801	1.05496	0.852655
0.775751	1.32923	1.2483	1.13963	0.868523	0.658992	0.00148049

Also, the @. macro applies a dot to every operator:

```
test5(A,B,C) = @. A + B + C #only one array allocated
@benchmark test5(A,B,C)
```

```
BenchmarkTools.Trial:
  memory estimate:  7.63 MiB
  allocs estimate:  3
  -----
  minimum time:     3.268 ms (0.00% GC)
  median time:      3.651 ms (0.00% GC)
  mean time:        4.914 ms (26.35% GC)
  maximum time:     22.319 ms (83.69% GC)
  -----
  samples:          1016
  evals/sample:     1
```

Using these tools we can get rid of our intermediate array allocations for many vectorized function calls. But we are still allocating the output array. To get rid of that allocation, we can instead use mutation. Mutating broadcast is done via .=. For example, if we pre-allocate the output:

```
D = zeros(1000,1000);
```

Then we can keep re-using this cache for subsequent calculations. The mutating broadcasting form is:

```
test6!(D,A,B,C) = D .= A .+ B .+ C #only one array allocated
@benchmark test6!(D,A,B,C)
```

```
BenchmarkTools.Trial:
  memory estimate:  0 bytes
  allocs estimate:  0
  -----
  minimum time:     1.604 ms (0.00% GC)
```



```

median time:      2.087 ms (0.00% GC)
mean time:        2.262 ms (0.00% GC)
maximum time:     6.563 ms (0.00% GC)
-----
samples:          2198
evals/sample:     1

```

If we use `@.` before the `=`, then it will turn it into `.=`:

```

test7!(D,A,B,C) = @. D = A + B + C #only one array allocated
@benchmark test7!(D,A,B,C)

```

```

BenchmarkTools.Trial:
 memory estimate:  0 bytes
 allocs estimate:  0
-----
minimum time:      1.562 ms (0.00% GC)
median time:       1.818 ms (0.00% GC)
mean time:         1.957 ms (0.00% GC)
maximum time:     6.713 ms (0.00% GC)
-----
samples:           2548
evals/sample:      1

```

Notice that in this case, there is no "output", and instead the values inside of `D` are what are changed (like with the `DiffEq` inplace function). Many Julia functions have a mutating form which is denoted with a `!`. For example, the mutating form of the `map` is `map!`:

```

test8!(D,A,B,C) = map!((a,b,c)->a+b+c,D,A,B,C)
@benchmark test8!(D,A,B,C)

```

```

BenchmarkTools.Trial:
 memory estimate:  32 bytes
 allocs estimate:  1
-----
minimum time:      1.681 ms (0.00% GC)
median time:       1.951 ms (0.00% GC)
mean time:         2.021 ms (0.00% GC)
maximum time:     4.499 ms (0.00% GC)
-----
samples:           2463
evals/sample:      1

```

Some operations require using an alternate mutating form in order to be fast. For example, matrix multiplication via `*` allocates a temporary:

```

@benchmark A*B

```

```

BenchmarkTools.Trial:
 memory estimate:  7.63 MiB

```

```

allocs estimate:  2
-----
minimum time:    12.582 ms (0.00% GC)
median time:     14.868 ms (0.00% GC)
mean time:       16.033 ms (9.67% GC)
maximum time:    33.857 ms (19.79% GC)
-----
samples:         311
evals/sample:    1

```

Instead, we can use the mutating form `mul!` into a cache array to avoid allocating the output:

```

using LinearAlgebra
@benchmark mul!(D,A,B) # same as D = A * B

```

```

BenchmarkTools.Trial:
 memory estimate:  0 bytes
 allocs estimate:  0
-----
minimum time:     11.171 ms (0.00% GC)
median time:      13.597 ms (0.00% GC)
mean time:        14.649 ms (0.00% GC)
maximum time:     26.574 ms (0.00% GC)
-----
samples:          341
evals/sample:     1

```

For repeated calculations this reduced allocation can stop GC cycles and thus lead to more efficient code. Additionally, *\*\*\**we can fuse together higher level linear algebra operations using BLAS*\*\*\**. The package [SugarBLAS.jl](#) makes it easy to write higher level operations like `alpha*B*A + beta*C` as mutating BLAS calls.

## 0.2.2 Example Optimization: Gierer-Meinhardt Reaction-Diffusion PDE Discretization

Let's optimize the solution of a Reaction-Diffusion PDE's discretization. In its discretized form, this is the ODE:

$$du = D_1(A_y u + u A_x) + \frac{au^2}{v} + \bar{u} - \alpha u \quad (1)$$

$$dv = D_2(A_y v + v A_x) + au^2 + \beta v \quad (2)$$

where  $u$ ,  $v$ , and  $A$  are matrices. Here, we will use the simplified version where  $A$  is the tridiagonal stencil  $[1, -2, 1]$ , i.e. it's the 2D discretization of the Laplacian. The native code would be something along the lines of:

```

# Generate the constants
p = (1.0,1.0,1.0,10.0,0.001,100.0) # a,α,ubar,β,D1,D2
N = 100

```

```

Ax = Array{Tridiagonal{Float64,Float64,Float64}}{1:N-1,1:N,1:N-1}
Ay = copy{Array{Float64,2}}{Ax}
Ax[2,1] = 2.0
Ax[end-1,end] = 2.0
Ay[1,2] = 2.0
Ay[end,end-1] = 2.0

function basic_version!(dr,r,p,t)
    a,α,ubar,β,D1,D2 = p
    u = r[:, :, 1]
    v = r[:, :, 2]
    Du = D1*(Ay*u + u*Ax)
    Dv = D2*(Ay*v + v*Ax)
    dr[:, :, 1] = Du .+ a.*u.*u./v .+ ubar .- α*u
    dr[:, :, 2] = Dv .+ a.*u.*u .- β*v
end

a,α,ubar,β,D1,D2 = p
uss = (ubar+β)/α
vss = (a/β)*uss^2
r0 = zeros{Float64,2}{100,100,2}
r0[:, :, 1] .= uss.+0.1.*rand{Float64}()
r0[:, :, 2] .= vss

prob = ODEProblem{basic_version!,r0,(0.0,0.1),p}

```

In this version we have encoded our initial condition to be a 3-dimensional array, with `u[:, :, 1]` being the A part and `u[:, :, 2]` being the B part.

```
@benchmark solve(prob,Tsit5())
```

```

BenchmarkTools.Trial:
 memory estimate: 186.88 MiB
 allocs estimate: 8590
-----
 minimum time:      96.116 ms (19.36% GC)
 median time:      232.588 ms (65.51% GC)
 mean time:        194.395 ms (57.40% GC)
 maximum time:     283.688 ms (68.10% GC)
-----
 samples:           27
 evals/sample:      1

```

While this version isn't very efficient,

**We recommend writing the "high-level" code first, and iteratively optimizing it!** The first thing that we can do is get rid of the slicing allocations. The operation `r[:, :, 1]` creates a temporary array instead of a "view", i.e. a pointer to the already existing memory. To make it a view, add `@view`. Note that we have to be careful with views because they point to the same memory, and thus changing a view changes the original values:

```

A = rand{Float64,2}{4}
@show A

```

```
A = [0.0808521, 0.0839342, 0.0638835, 0.693744]
```

```
B = @view A[1:3]
B[2] = 2
@show A
```

```
A = [0.0808521, 2.0, 0.0638835, 0.693744]
4-element Array{Float64,1}:
 0.08085213057920293
 2.0
 0.0638834840667819
 0.6937436324245521
```

Notice that changing B changed A. This is something to be careful of, but at the same time we want to use this since we want to modify the output `dr`. Additionally, the last statement is a purely element-wise operation, and thus we can make use of broadcast fusion there. Let's rewrite `basic_version!` to `***avoid slicing allocations***` and to `***use broadcast fusion***`:

```
function gm2!(dr,r,p,t)
    a,α,ubar,β,D1,D2 = p
    u = @view r[:,1]
    v = @view r[:,2]
    du = @view dr[:,1]
    dv = @view dr[:,2]
    Du = D1*(Ay*u + u*Ax)
    Dv = D2*(Ay*v + v*Ax)
    @. du = Du + a.*u.*u./v + ubar - α*u
    @. dv = Dv + a.*u.*u - β*v
end
prob = ODEProblem(gm2!,r0,(0.0,0.1),p)
@benchmark solve(prob,Tsit5())
```

```
BenchmarkTools.Trial:
 memory estimate: 119.55 MiB
  allocs estimate: 7120
  -----
 minimum time:      71.850 ms (16.54% GC)
 median time:      117.019 ms (41.30% GC)
 mean time:        156.724 ms (57.01% GC)
 maximum time:     248.975 ms (70.25% GC)
  -----
 samples:          32
 evals/sample:     1
```

Now, most of the allocations are taking place in `Du = D1*(Ay*u + u*Ax)` since those operations are vectorized and not mutating. We should instead replace the matrix multiplications with `mul!`. When doing so, we will need to have cache variables to write into. This looks like:

```

Ayu = zeros(N,N)
uAx = zeros(N,N)
Du = zeros(N,N)
Ayv = zeros(N,N)
vAx = zeros(N,N)
Dv = zeros(N,N)
function gm3!(dr,r,p,t)
    a, $\alpha$ ,ubar, $\beta$ ,D1,D2 = p
    u = @view r[:, :, 1]
    v = @view r[:, :, 2]
    du = @view dr[:, :, 1]
    dv = @view dr[:, :, 2]
    mul!(Ayu,Ay,u)
    mul!(uAx,u,Ax)
    mul!(Ayv,Ay,v)
    mul!(vAx,v,Ax)
    @. Du = D1*(Ayu + uAx)
    @. Dv = D2*(Ayv + vAx)
    @. du = Du + a*u*u./v + ubar -  $\alpha$ *u
    @. dv = Dv + a*u*u -  $\beta$ *v
end
prob = ODEProblem(gm3!,r0,(0.0,0.1),p)
@benchmark solve(prob,Tsit5())

```

```

BenchmarkTools.Trial:
  memory estimate: 29.76 MiB
  allocs estimate: 5356
  -----
  minimum time:      49.730 ms (4.46% GC)
  median time:       52.239 ms (4.20% GC)
  mean time:         52.703 ms (5.48% GC)
  maximum time:      63.235 ms (3.19% GC)
  -----
  samples:           95
  evals/sample:      1

```

But our temporary variables are global variables. We need to either declare the caches as `const` or localize them. We can localize them by adding them to the parameters, `p`. It's easier for the compiler to reason about local variables than global variables. **\*\*\*Localizing variables helps to ensure type stability\*\*\***.

```

p = (1.0,1.0,1.0,10.0,0.001,100.0,Ayu,uAx,Du,Ayv,vAx,Dv) # a, $\alpha$ ,ubar, $\beta$ ,D1,D2
function gm4!(dr,r,p,t)
    a, $\alpha$ ,ubar, $\beta$ ,D1,D2,Ayu,uAx,Du,Ayv,vAx,Dv = p
    u = @view r[:, :, 1]
    v = @view r[:, :, 2]
    du = @view dr[:, :, 1]
    dv = @view dr[:, :, 2]
    mul!(Ayu,Ay,u)
    mul!(uAx,u,Ax)
    mul!(Ayv,Ay,v)
    mul!(vAx,v,Ax)
    @. Du = D1*(Ayu + uAx)
    @. Dv = D2*(Ayv + vAx)
    @. du = Du + a*u*u./v + ubar -  $\alpha$ *u
    @. dv = Dv + a*u*u -  $\beta$ *v
end

```

```

end
prob = ODEProblem(gm4!,r0,(0.0,0.1),p)
@benchmark solve(prob,Tsit5())

```

```

BenchmarkTools.Trial:
 memory estimate: 29.66 MiB
 allocs estimate: 1091
-----
 minimum time:      42.730 ms (5.07% GC)
 median time:      44.700 ms (5.42% GC)
 mean time:        45.369 ms (6.50% GC)
 maximum time:     64.501 ms (22.62% GC)
-----
 samples:          111
 evals/sample:     1

```

We could then use the BLAS `gemmv` to optimize the matrix multiplications some more, but instead let's devectorize the stencil.

```

p = (1.0,1.0,1.0,10.0,0.001,100.0,N)
function fast_gm!(du,u,p,t)
    a,α,ubar,β,D1,D2,N = p

    @inbounds for j in 2:N-1, i in 2:N-1
        du[i,j,1] = D1*(u[i-1,j,1] + u[i+1,j,1] + u[i,j+1,1] + u[i,j-1,1] - 4u[i,j,1]) +
            a*u[i,j,1]^2/u[i,j,2] + ubar - α*u[i,j,1]
    end

    @inbounds for j in 2:N-1, i in 2:N-1
        du[i,j,2] = D2*(u[i-1,j,2] + u[i+1,j,2] + u[i,j+1,2] + u[i,j-1,2] - 4u[i,j,2]) +
            a*u[i,j,1]^2 - β*u[i,j,2]
    end

    @inbounds for j in 2:N-1
        i = 1
        du[1,j,1] = D1*(2u[i+1,j,1] + u[i,j+1,1] + u[i,j-1,1] - 4u[i,j,1]) +
            a*u[i,j,1]^2/u[i,j,2] + ubar - α*u[i,j,1]
    end

    @inbounds for j in 2:N-1
        i = 1
        du[1,j,2] = D2*(2u[i+1,j,2] + u[i,j+1,2] + u[i,j-1,2] - 4u[i,j,2]) +
            a*u[i,j,1]^2 - β*u[i,j,2]
    end

    @inbounds for j in 2:N-1
        i = N
        du[end,j,1] = D1*(2u[i-1,j,1] + u[i,j+1,1] + u[i,j-1,1] - 4u[i,j,1]) +
            a*u[i,j,1]^2/u[i,j,2] + ubar - α*u[i,j,1]
    end

    @inbounds for j in 2:N-1
        i = N
        du[end,j,2] = D2*(2u[i-1,j,2] + u[i,j+1,2] + u[i,j-1,2] - 4u[i,j,2]) +
            a*u[i,j,1]^2 - β*u[i,j,2]
    end

    @inbounds for i in 2:N-1
        j = 1
    end

```

```

    du[i,1,1] = D1*(u[i-1,j,1] + u[i+1,j,1] + 2u[i,j+1,1] - 4u[i,j,1]) +
        a*u[i,j,1]^2/u[i,j,2] + ubar - α*u[i,j,1]
end
@inbounds for i in 2:N-1
    j = 1
    du[i,1,2] = D2*(u[i-1,j,2] + u[i+1,j,2] + 2u[i,j+1,2] - 4u[i,j,2]) +
        a*u[i,j,1]^2 - β*u[i,j,2]
end
@inbounds for i in 2:N-1
    j = N
    du[i,end,1] = D1*(u[i-1,j,1] + u[i+1,j,1] + 2u[i,j-1,1] - 4u[i,j,1]) +
        a*u[i,j,1]^2/u[i,j,2] + ubar - α*u[i,j,1]
end
@inbounds for i in 2:N-1
    j = N
    du[i,end,2] = D2*(u[i-1,j,2] + u[i+1,j,2] + 2u[i,j-1,2] - 4u[i,j,2]) +
        a*u[i,j,1]^2 - β*u[i,j,2]
end

@inbounds begin
    i = 1; j = 1
    du[1,1,1] = D1*(2u[i+1,j,1] + 2u[i,j+1,1] - 4u[i,j,1]) +
        a*u[i,j,1]^2/u[i,j,2] + ubar - α*u[i,j,1]
    du[1,1,2] = D2*(2u[i+1,j,2] + 2u[i,j+1,2] - 4u[i,j,2]) +
        a*u[i,j,1]^2 - β*u[i,j,2]

    i = 1; j = N
    du[1,N,1] = D1*(2u[i+1,j,1] + 2u[i,j-1,1] - 4u[i,j,1]) +
        a*u[i,j,1]^2/u[i,j,2] + ubar - α*u[i,j,1]
    du[1,N,2] = D2*(2u[i+1,j,2] + 2u[i,j-1,2] - 4u[i,j,2]) +
        a*u[i,j,1]^2 - β*u[i,j,2]

    i = N; j = 1
    du[N,1,1] = D1*(2u[i-1,j,1] + 2u[i,j+1,1] - 4u[i,j,1]) +
        a*u[i,j,1]^2/u[i,j,2] + ubar - α*u[i,j,1]
    du[N,1,2] = D2*(2u[i-1,j,2] + 2u[i,j+1,2] - 4u[i,j,2]) +
        a*u[i,j,1]^2 - β*u[i,j,2]

    i = N; j = N
    du[end,end,1] = D1*(2u[i-1,j,1] + 2u[i,j-1,1] - 4u[i,j,1]) +
        a*u[i,j,1]^2/u[i,j,2] + ubar - α*u[i,j,1]
    du[end,end,2] = D2*(2u[i-1,j,2] + 2u[i,j-1,2] - 4u[i,j,2]) +
        a*u[i,j,1]^2 - β*u[i,j,2]
end
end
prob = ODEProblem(fast_gm!,r0,(0.0,0.1),p)
@benchmark solve(prob,Tsit5())

```

```

BenchmarkTools.Trial:
 memory estimate: 29.63 MiB
 allocs estimate: 506
-----
 minimum time:      10.032 ms (15.64% GC)
 median time:       10.835 ms (17.24% GC)
 mean time:         11.198 ms (20.62% GC)
 maximum time:      21.637 ms (35.17% GC)
-----
 samples:           446

```

```
evals/sample:      1
```

Lastly, we can do other things like multithread the main loops, but these optimizations get the last 2x-3x out. The main optimizations which apply everywhere are the ones we just performed (though the last one only works if your matrix is a stencil. This is known as a matrix-free implementation of the PDE discretization).

This gets us to about 8x faster than our original MATLAB/SciPy/R vectorized style code!

The last thing to do is then `***optimize our algorithm choice***`. We have been using `Tsit5()` as our test algorithm, but in reality this problem is a stiff PDE discretization and thus one recommendation is to use `CVODE_BDF()`. However, instead of using the default dense Jacobian, we should make use of the sparse Jacobian afforded by the problem. The Jacobian is the matrix  $\frac{df_i}{dr_j}$ , where  $r$  is read by the linear index (i.e. down columns). But since the  $u$  variables depend on the  $v$ , the band size here is large, and thus this will not do well with a Banded Jacobian solver. Instead, we utilize sparse Jacobian algorithms. `CVODE_BDF` allows us to use a sparse Newton-Krylov solver by setting `linear_solver = :GMRES` (see [the solver documentation](#), and thus we can solve this problem efficiently. Let's see how this scales as we increase the integration time.

```
prob = ODEProblem(fast_gm!,r0,(0.0,10.0),p)
@benchmark solve(prob,Tsit5())
```

```
BenchmarkTools.Trial:
  memory estimate:  2.76 GiB
  allocs estimate:  41690
  -----
  minimum time:     2.243 s (31.51% GC)
  median time:      4.049 s (13.25% GC)
  mean time:        4.049 s (13.25% GC)
  maximum time:     5.855 s (6.26% GC)
  -----
  samples:          2
  evals/sample:     1
```

```
using Sundials
@benchmark solve(prob,CVODE_BDF(linear_solver=:GMRES))
```

```
BenchmarkTools.Trial:
  memory estimate:  117.28 MiB
  allocs estimate:  32905
  -----
  minimum time:     549.482 ms (2.10% GC)
  median time:      703.733 ms (24.11% GC)
  mean time:        684.796 ms (21.98% GC)
  maximum time:     743.778 ms (27.93% GC)
  -----
  samples:          8
  evals/sample:     1
```



```

prob = ODEProblem(fast_gm!,r0,(0.0,100.0),p)
# Will go out of memory if we don't turn off `save_everystep`!
@benchmark solve(prob,Tsit5(),save_everystep=false)

```

```

BenchmarkTools.Trial:
  memory estimate:  2.91 MiB
  allocs estimate:  114
  -----
  minimum time:     4.828 s (0.00% GC)
  median time:      4.932 s (0.00% GC)
  mean time:        4.932 s (0.00% GC)
  maximum time:     5.036 s (0.00% GC)
  -----
  samples:          2
  evals/sample:     1

```

```

@benchmark solve(prob,CVODE_BDF(linear_solver=:GMRES))

```

```

BenchmarkTools.Trial:
  memory estimate:  317.88 MiB
  allocs estimate:  90975
  -----
  minimum time:     1.603 s (0.00% GC)
  median time:      1.718 s (9.35% GC)
  mean time:        1.758 s (10.11% GC)
  maximum time:     1.953 s (19.07% GC)
  -----
  samples:          3
  evals/sample:     1

```

Now let's check the allocation growth.

```

@benchmark solve(prob,CVODE_BDF(linear_solver=:GMRES),save_everystep=false)

```

```

BenchmarkTools.Trial:
  memory estimate:  4.21 MiB
  allocs estimate:  81365
  -----
  minimum time:     1.525 s (0.00% GC)
  median time:      1.529 s (0.00% GC)
  mean time:        1.529 s (0.00% GC)
  maximum time:     1.534 s (0.00% GC)
  -----
  samples:          4
  evals/sample:     1

```

```

prob = ODEProblem(fast_gm!,r0,(0.0,500.0),p)
@benchmark solve(prob,CVODE_BDF(linear_solver=:GMRES),save_everystep=false)

```

```

BenchmarkTools.Trial:
  memory estimate:  5.51 MiB
  allocs estimate: 113492
  -----
  minimum time:     2.148 s (0.00% GC)
  median time:      2.151 s (0.00% GC)
  mean time:        2.150 s (0.00% GC)
  maximum time:     2.152 s (0.00% GC)
  -----
  samples:          3
  evals/sample:     1

```

Notice that we've eliminated almost all allocations, allowing the code to grow without hitting garbage collection and slowing down.

Why is `CVODE_BDF` doing well? What's happening is that, because the problem is stiff, the number of steps required by the explicit Runge-Kutta method grows rapidly, whereas `CVODE_BDF` is taking large steps. Additionally, the `GMRES` linear solver form is quite an efficient way to solve the implicit system in this case. This is problem-dependent, and in many cases using a Krylov method effectively requires a preconditioner, so you need to play around with testing other algorithms and linear solvers to find out what works best with your problem.

### 0.3 Conclusion

Julia gives you the tools to optimize the solver "all the way", but you need to make use of it. The main thing to avoid is temporary allocations. For small systems, this is effectively done via static arrays. For large systems, this is done via in-place operations and cache arrays. Either way, the resulting solution can be immensely sped up over vectorized formulations by using these principles.

### 0.4 Appendix

```

using DiffEqTutorials
DiffEqTutorials.tutorial_footer(WEAVE_ARGS[:folder],WEAVE_ARGS[:file])

```

These benchmarks are part of the `DiffEqTutorials.jl` repository, found at:

<https://github.com/JuliaDiffEq/DiffEqTutorials.jl>

To locally run this tutorial, do the following commands:

```

using DiffEqTutorials
DiffEqTutorials.weave_file("introduction","optimizing_diffeq_code.jmd")

```

Computer Information:

```

Julia Version 1.1.0
Commit 80516ca202 (2019-01-21 21:24 UTC)
Platform Info:
  OS: Windows (x86_64-w64-mingw32)
  CPU: Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz

```

```

WORD_SIZE: 64
LIBM: libopenlibm
LLVM: libLLVM-6.0.1 (ORCJIT, skylake)
Environment:
  JULIA_EDITOR = "C:\Users\accou\AppData\Local\atom\app-1.34.0\atom.exe" -a
  JULIA_NUM_THREADS = 6

```

Package Information:

```

Status `C:\Users\accou\.julia\environments\v1.1\Project.toml`
[7e558dbc] ArbNumerics v0.3.6
[c52e3926] Atom v0.7.14
[6e4b80f9] BenchmarkTools v0.4.2
[336ed68f] CSV v0.4.3
[3895d2a7] CUDAapi v0.5.4
[be33ccc6] CUDAnative v1.0.1
[3a865a2d] CuArrays v0.9.1
[a93c6f00] DataFrames v0.17.1
[55939f99] DecFP v0.4.8
[abce61dc] Decimals v0.4.0
[39dd38d3] Dierckx v0.4.1
[459566f4] DiffEqCallbacks v2.5.2
[f3b72e0c] DiffEqDevTools v2.6.1
[aae7a2af] DiffEqFlux v0.2.0
[c894b116] DiffEqJump v6.1.0+ [~C:\Users\accou\.julia\dev\DiffEqJump`]
[1130ab10] DiffEqParamEstim v1.6.0+ [~C:\Users\accou\.julia\dev\DiffEqParamEstim`]
[055956cb] DiffEqPhysics v3.1.0
[225cb15b] DiffEqTutorials v0.0.0 [~C:\Users\accou\.julia\external\DiffEqTutorials.jl`]
[0c46a032] DifferentialEquations v6.3.0
[497a8b3b] DoubleFloats v0.7.5
[587475ba] Flux v0.7.3
[f6369f11] ForwardDiff v0.10.3+ [~C:\Users\accou\.julia\dev\ForwardDiff`]
[28b8d3ca] GR v0.38.1
[7073ff75] IJulia v1.17.0
[c601a237] Interact v0.9.1
[b6b21f68] Ipopt v0.5.4
[4076af6c] JuMP v0.19.0
[e5e0dc1b] Juno v0.5.4
[eff96d63] Measurements v2.0.0
[76087f3c] NLOpt v0.5.1
[429524aa] Optim v0.17.2
[1dea7af3] OrdinaryDiffEq v5.2.1+ [~C:\Users\accou\.julia\dev\OrdinaryDiffEq`]
[65888b18] ParameterizedFunctions v4.1.1
[91a5bcd] Plots v0.23.0
[71ad9d73] PuMaS v0.0.0 [~C:\Users\accou\.julia\dev\PuMaS`]
[d330b81b] PyPlot v2.7.0
[731186ca] RecursiveArrayTools v0.20.0
[90137ffa] StaticArrays v0.10.2
[789caeaf] StochasticDiffEq v6.1.1+ [~C:\Users\accou\.julia\dev\StochasticDiffEq`]
[c3572dad] Sundials v3.0.0
[1986cc42] Unitful v0.14.0
[2a06ce6d] UnitfulPlots v0.0.0 #master (https://github.com/ajkeller34/UnitfulPlots.jl)
[44d3d7a6] Weave v0.7.1 [~C:\Users\accou\.julia\dev\Weave`]

```