Optimizing DiffEq Code

Chris Rackauckas

February 26, 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:     3.432 ms (0.00% GC)
  median time:      7.508 ms (0.00% GC)
  mean time:        6.054 ms (32.65% GC)
  maximum time:     10.902 ms (53.43% GC)
  --------------
  samples:          825
  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:     3.043 ms (0.00% GC)
  median time:      4.049 ms (0.00% GC)
  mean time:        5.580 ms (32.84% GC)
  maximum time:     16.875 ms (54.95% GC)
  --------------
  samples:          895
  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:     750.500 μs (0.00% GC)
  median time:      788.100 μs (0.00% GC)
  mean time:        1.073 ms (23.04% GC)
  maximum time:     7.777 ms (81.01% GC)
  --------------
  samples:          4646
  evals/sample:     1
```

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

```
BenchmarkTools.Trial:
  memory estimate:  6.91 KiB
  allocs estimate:  93
  --------------
  minimum time:     408.400 μs (0.00% GC)
  median time:      413.000 μs (0.00% GC)
  mean time:        422.758 μs (0.46% GC)
  maximum time:     7.579 ms (93.83% 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:     2.040 ms (0.00% GC)
  median time:      2.102 ms (0.00% GC)
  mean time:        2.137 ms (0.00% GC)
  maximum time:     2.800 ms (0.00% GC)
  --------------
  samples:          2337
  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. `struct`s for example are stack-allocated "value-type"s. `Tuple`s 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, `SVector`s 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. `SArray`s 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:     446.000 μs (0.00% GC)
  median time:      456.499 μs (0.00% GC)
  mean time:        519.058 μs (10.23% GC)
  maximum time:     4.370 ms (88.33% GC)
  --------------
  samples:          9595
  evals/sample:     1
```

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

```
BenchmarkTools.Trial:
  memory estimate:  6.20 KiB
  allocs estimate:  74
  --------------
  minimum time:     354.500 μs (0.00% GC)
  median time:      359.699 μs (0.00% GC)
  mean time:        366.869 μs (0.41% GC)
  maximum time:     9.197 ms (95.56% 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:     3.314 ms (0.00% GC)
  median time:      3.829 ms (0.00% GC)
  mean time:        5.205 ms (26.52% GC)
  maximum time:     16.879 ms (62.13% GC)
  --------------
  samples:          959
  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:     3.912 ms (0.00% GC)
  median time:      4.468 ms (0.00% GC)
  mean time:        5.725 ms (23.97% GC)
  maximum time:     15.726 ms (55.42% GC)
  --------------
  samples:          873
  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:     3.295 ms (0.00% GC)
  median time:      3.658 ms (0.00% GC)
  mean time:        4.911 ms (26.09% GC)
  maximum time:     27.593 ms (87.80% GC)
  --------------
  samples:          1017
  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:     3.282 ms (0.00% GC)
  median time:      3.727 ms (0.00% GC)
  mean time:        5.031 ms (26.44% GC)
  maximum time:     19.949 ms (80.20% GC)
  --------------
  samples:          993
  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}:
 0.714224  0.932906  0.687867  0.333103  …  0.65617   0.620043  0.680045
 0.442348  1.22639   0.619741  0.699785     0.79441   0.205799  1.41574
 0.614579  1.24894   1.15599   0.383281     0.561799  0.905566  1.0797
 0.976279  0.795133  0.967587  0.346901     0.427098  0.622939  0.738418
 0.662795  1.00588   1.23635   0.488709     0.412131  0.979077  1.06539
 0.527144  0.643899  0.821488  0.760522  …  0.734829  0.637803  0.760276
```

```
 0.955341  0.586305  0.839382  1.22949      1.01606   0.581158  1.42245
 0.937299  0.608347  1.32198   0.764027     1.14296   1.28124   1.04175
 0.290494  1.02107   1.43987   1.07987      1.01713   0.512618  1.33427
 1.35285   1.52333   0.611285  1.06191      0.839572  0.937584  1.61233
 ⋮                                    ⋱
 0.297008  0.760293  0.586464  1.49456      0.587711  0.974146  1.19358
 0.949055  0.401827  0.951537  1.28537      0.901635  1.29708   1.1673
 0.705743  0.878169  0.590304  0.85596      1.05523   1.55694   1.31649
 1.07392   0.595717  0.353329  0.79761      0.42773   1.17119   1.35263
 0.976411  0.933483  0.89384   1.2152   …   0.218495  0.341001  0.935505
 0.760544  0.62652   1.08118   0.459699     0.336436  1.44743   1.09185
 1.12708   0.825045  0.200494  1.34024      0.513717  0.674917  0.873277
 1.21654   1.48754   0.998283  1.51498      0.921581  0.459839  1.1186
 0.718792  0.535398  1.27996   0.842256     1.20256   0.862517  1.34342
```

Also, the `@.` macro applys 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:     3.313 ms (0.00% GC)
  median time:      3.947 ms (0.00% GC)
  mean time:        5.308 ms (27.39% GC)
  maximum time:     25.335 ms (78.13% GC)
  --------------
  samples:          940
  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:     1.599 ms (0.00% GC)
```

```
  median time:      2.076 ms (0.00% GC)
  mean time:        2.221 ms (0.00% GC)
  maximum time:     6.187 ms (0.00% GC)
  --------------
  samples:          2239
  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:     1.593 ms (0.00% GC)
  median time:      1.950 ms (0.00% GC)
  mean time:        2.054 ms (0.00% GC)
  maximum time:     5.612 ms (0.00% GC)
  --------------
  samples:          2422
  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:     1.727 ms (0.00% GC)
  median time:      1.910 ms (0.00% GC)
  mean time:        1.971 ms (0.00% GC)
  maximum time:     7.242 ms (0.00% GC)
  --------------
  samples:          2525
  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:     12.495 ms (0.00% GC)
  median time:      16.926 ms (0.00% GC)
  mean time:        17.553 ms (9.26% GC)
  maximum time:     29.295 ms (20.42% GC)
  --------------
  samples:          285
  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:     11.765 ms (0.00% GC)
  median time:      15.602 ms (0.00% GC)
  mean time:        17.136 ms (0.00% GC)
  maximum time:     37.235 ms (0.00% GC)
  --------------
  samples:          292
  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:

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

$$dv = D_2(A_y v + v A_x) + a u^2 + \beta v \tag{2}$$

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([1.0 for i in 1:N-1],[-2.0 for i in 1:N],[1.0 for i in 1:N-1]))
Ay = copy(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(100,100,2)
r0[:,:,1] .= uss.+0.1.*rand.()
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:     92.036 ms (20.94% GC)
  median time:      239.035 ms (64.96% GC)
  mean time:        201.085 ms (57.86% GC)
  maximum time:     344.738 ms (68.73% GC)
  --------------
  samples:          25
  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(4)
@show A
```

```
A = [0.357901, 0.758099, 0.604851, 0.67412]
```

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

```
A = [0.357901, 2.0, 0.604851, 0.67412]
4-element Array{Float64,1}:
 0.3579013406073024
 2.0
 0.604850581872213
 0.6741203339744615
```

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:     67.013 ms (17.12% GC)
  median time:      116.285 ms (43.61% GC)
  mean time:        157.402 ms (58.08% GC)
  maximum time:     266.432 ms (72.23% 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,α,ubar,β,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 - α*u
  @. dv = Dv + a*u*u - β*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:     49.627 ms (4.61% GC)
  median time:      51.254 ms (4.17% GC)
  mean time:        51.783 ms (5.36% GC)
  maximum time:     54.931 ms (3.91% GC)
  --------------
  samples:          97
  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,α,ubar,β,D1,D2
function gm4!(dr,r,p,t)
  a,α,ubar,β,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 - α*u
  @. dv = Dv + a*u*u - β*v
```

```
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:     42.962 ms (4.71% GC)
  median time:      44.314 ms (4.55% GC)
  mean time:        45.113 ms (6.05% GC)
  maximum time:     63.534 ms (21.57% 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
```

```
    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:     9.833 ms (14.95% GC)
  median time:      10.715 ms (17.68% GC)
  mean time:        11.013 ms (20.68% GC)
  maximum time:     16.579 ms (11.46% GC)
  --------------
  samples:          454
```

```
  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:  41672
  --------------
  minimum time:     2.209 s (32.64% GC)
  median time:      5.801 s (26.55% GC)
  mean time:        5.801 s (26.55% GC)
  maximum time:     9.394 s (25.12% GC)
  --------------
  samples:          2
  evals/sample:     1
```

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

```
BenchmarkTools.Trial:
  memory estimate:  114.48 MiB
  allocs estimate:  31563
  --------------
  minimum time:     521.555 ms (2.19% GC)
  median time:      673.733 ms (24.39% GC)
  mean time:        654.493 ms (22.19% GC)
  maximum time:     712.025 ms (28.14% 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:     4.734 s (0.00% GC)
  median time:      4.795 s (0.00% GC)
  mean time:        4.795 s (0.00% GC)
  maximum time:     4.857 s (0.00% GC)
  --------------
  samples:          2
  evals/sample:     1
```

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

```
BenchmarkTools.Trial:
  memory estimate:  385.20 MiB
  allocs estimate:  103303
  --------------
  minimum time:     1.861 s (0.00% GC)
  median time:      1.933 s (8.23% GC)
  mean time:        1.979 s (8.77% GC)
  maximum time:     2.144 s (16.87% 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:  4.62 MiB
  allocs estimate:  91649
  --------------
  minimum time:     1.728 s (0.00% GC)
  median time:      1.728 s (0.00% GC)
  mean time:        1.728 s (0.00% GC)
  maximum time:     1.728 s (0.00% GC)
  --------------
  samples:          3
  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:  6.34 MiB
  allocs estimate:  133976
  --------------
  minimum time:     2.532 s (0.00% GC)
  median time:      2.538 s (0.00% GC)
  mean time:        2.538 s (0.00% GC)
  maximum time:     2.544 s (0.00% GC)
  --------------
  samples:          2
  evals/sample:     1
```

Notice that we've elimated 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
  [bb2cbb15] DiffEqBenchmarks v0.0.0 [`C:\Users\accou\.julia\external\DiffE
qBenchmarks.jl`]
  [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\DiffEqPar
amEstim`]
  [055956cb] DiffEqPhysics v3.1.0
  [a077e3f3] DiffEqProblemLibrary v4.1.0
  [225cb15b] DiffEqTutorials v0.0.0 [`C:\Users\accou\.julia\external\DiffEq
Tutorials.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
  [7f56f5a3] LSODA v0.4.0
  [eff96d63] Measurements v2.0.0
  [76087f3c] NLopt v0.5.1
  [c030b06c] ODE v2.4.0
  [54ca160b] ODEInterface v0.4.5+ [`C:\Users\accou\.julia\dev\ODEInterface`
]
  [09606e27] ODEInterfaceDiffEq v3.0.0
  [429524aa] Optim v0.17.2
  [1dea7af3] OrdinaryDiffEq v5.2.1+ [`C:\Users\accou\.julia\dev\OrdinaryDif
fEq`]
  [65888b18] ParameterizedFunctions v4.1.1
  [91a5bcdd] 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\Stochasti
cDiffEq`]
  [c3572dad] Sundials v3.0.0
  [1986cc42] Unitful v0.14.0
  [2a06ce6d] UnitfulPlots v0.0.0 #master (https://github.com/ajkeller34/Uni
tfulPlots.jl)
  [44d3d7a6] Weave v0.7.1 [`C:\Users\accou\.julia\dev\Weave`]
```