Optimizing DiffEq Code

Chris Rackauckas

February 23, 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:  11.30 MiB
  allocs estimate:  126166
  --------------
  minimum time:     3.855 ms (0.00% GC)
  median time:      6.781 ms (34.35% GC)
  mean time:        6.520 ms (27.77% GC)
  maximum time:     17.276 ms (39.85% GC)
  --------------
  samples:          766
  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:  9.93 MiB
  allocs estimate:  113274
  --------------
  minimum time:     3.443 ms (0.00% GC)
  median time:      3.740 ms (0.00% GC)
  mean time:        4.682 ms (21.84% GC)
  maximum time:     11.116 ms (57.07% GC)
  --------------
  samples:          1067
  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:  1.86 MiB
  allocs estimate:  44799
  --------------
  minimum time:     1.263 ms (0.00% GC)
  median time:      1.286 ms (0.00% GC)
  mean time:        1.511 ms (14.07% GC)
  maximum time:     6.094 ms (65.11% GC)
  --------------
  samples:          3304
  evals/sample:     1
```

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

```
BenchmarkTools.Trial:
  memory estimate:  504.30 KiB
  allocs estimate:  31927
  --------------
  minimum time:     900.199 μs (0.00% GC)
  median time:      912.201 μs (0.00% GC)
  mean time:        949.242 μs (3.22% GC)
  maximum time:     2.860 ms (62.25% GC)
  --------------
  samples:          5258
  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:  2.51 MiB
  allocs estimate:  164407
  --------------
  minimum time:     4.582 ms (0.00% GC)
  median time:      4.620 ms (0.00% GC)
  mean time:        4.794 ms (3.07% GC)
  maximum time:     12.930 ms (0.00% GC)
  --------------
  samples:          1043
  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:

```julia
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:

```julia
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:

```julia
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:  1.30 MiB
  allocs estimate:  42156
  --------------
  minimum time:     1.004 ms (0.00% GC)
  median time:      1.018 ms (0.00% GC)
  mean time:        1.187 ms (13.58% GC)
  maximum time:     4.700 ms (76.54% GC)
  --------------
  samples:          4208
  evals/sample:     1
```

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

```
BenchmarkTools.Trial:
  memory estimate:  869.97 KiB
  allocs estimate:  39567
  --------------
  minimum time:     900.000 μs (0.00% GC)
  median time:      909.800 μs (0.00% GC)
  mean time:        1.035 ms (11.62% GC)
  maximum time:     4.923 ms (79.79% GC)
  --------------
  samples:          4822
  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:     3.262 ms (0.00% GC)
  median time:      3.779 ms (0.00% GC)
  mean time:        4.697 ms (19.25% GC)
  maximum time:     13.252 ms (70.65% GC)
  --------------
  samples:          1063
  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:     3.893 ms (0.00% GC)
  median time:      4.527 ms (0.00% GC)
  mean time:        5.390 ms (17.36% GC)
  maximum time:     10.880 ms (32.80% GC)
  --------------
  samples:          927
  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:

```julia
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:     3.255 ms (0.00% GC)
  median time:      3.784 ms (0.00% GC)
  mean time:        4.706 ms (19.37% GC)
  maximum time:     13.449 ms (41.43% GC)
  --------------
  samples:          1061
  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:

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

```
BenchmarkTools.Trial:
  memory estimate:  7.63 MiB
  allocs estimate:  2
  --------------
  minimum time:     3.245 ms (0.00% GC)
  median time:      3.755 ms (0.00% GC)
  mean time:        4.653 ms (19.05% GC)
  maximum time:     9.599 ms (57.62% GC)
  --------------
  samples:          1073
  evals/sample:     1
```

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

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

```
1000×1000 Array{Float64,2}:
 0.275705  0.934648  0.298178  0.914204  …  0.995341  1.35461   0.763932
 1.2256    1.09956   1.02197   0.941697     0.924736  0.706136  1.10077
 1.29598   0.835012  1.61103   1.36711      0.834129  1.04251   0.92353
 0.867255  0.87444   0.676952  1.19853      0.597436  1.37658   1.28946
 1.31984   1.37557   0.455835  0.496353     0.918893  1.00506   1.38571
 1.48529   1.11237   1.20667   1.10481   …  0.860972  1.14452   1.19021
```

```
 1.15797   0.735108  0.897218  0.736023     0.959881  0.318314  0.728505
 0.620286  1.55429   1.06849   1.58117      1.3684    0.628266  0.582272
 0.51857   1.26545   1.54631   1.00339      0.995728  0.960794  1.46078
 0.564607  1.21519   0.342113  1.40072      0.675041  0.848535  1.07118
 ⋮                                       ⋱
 0.439002  0.754796  0.918076  0.936252     0.831746  1.27704   0.277656
 0.3176    1.20863   0.791136  0.830148     0.808349  0.703079  0.786642
 0.893254  1.65426   0.993469  1.09693      1.33081   0.749873  0.481575
 0.854362  0.859931  0.242643  1.36194      1.01201   0.818933  1.01283
 0.446734  1.38122   0.556623  1.45153   …  0.726064  0.662046  0.800921
 0.945935  0.823369  1.29284   0.362687     1.4223    0.710028  1.33903
 0.999638  0.752763  0.410413  0.883474     0.460687  0.932742  0.597102
 0.86352   0.759167  1.04646   0.716764     0.821258  1.45649   1.32533
 0.947768  0.848695  1.15319   0.993951     0.934637  1.33403   0.475995
```

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

```julia
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:     3.237 ms (0.00% GC)
  median time:      3.811 ms (0.00% GC)
  mean time:        4.773 ms (19.17% GC)
  maximum time:     11.407 ms (41.55% GC)
  --------------
  samples:          1046
  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:

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

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

```julia
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:     1.704 ms (0.00% GC)
```

```
  median time:      1.984 ms (0.00% GC)
  mean time:        2.048 ms (0.00% GC)
  maximum time:     3.107 ms (0.00% GC)
  --------------
  samples:          2429
  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:     1.704 ms (0.00% GC)
  median time:      1.971 ms (0.00% GC)
  mean time:        2.045 ms (0.00% GC)
  maximum time:     3.297 ms (0.00% GC)
  --------------
  samples:          2433
  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:     1.881 ms (0.00% GC)
  median time:      2.097 ms (0.00% GC)
  mean time:        2.196 ms (0.00% GC)
  maximum time:     4.868 ms (0.00% GC)
  --------------
  samples:          2269
  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:     13.889 ms (0.00% GC)
  median time:      17.913 ms (0.00% GC)
  mean time:        18.514 ms (6.39% GC)
  maximum time:     29.153 ms (13.38% GC)
  --------------
  samples:          270
  evals/sample:     1
```

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

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

```
BenchmarkTools.Trial:
  memory estimate:  0 bytes
  allocs estimate:  0
  --------------
  minimum time:     14.810 ms (0.00% GC)
  median time:      20.176 ms (0.00% GC)
  mean time:        23.031 ms (0.00% GC)
  maximum time:     71.032 ms (0.00% GC)
  --------------
  samples:          217
  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:

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

$$dv = D_2(A_y v + vA_x) + au^2 + \beta v \tag{2}$$

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:

```julia
# 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:  194.54 MiB
  allocs estimate:  9548
  --------------
  minimum time:     76.815 ms (12.98% GC)
  median time:      180.850 ms (57.21% GC)
  mean time:        153.055 ms (45.14% GC)
  maximum time:     245.897 ms (58.48% GC)
  --------------
  samples:          33
  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 = [0.678519, 0.799805, 0.526267, 0.0397595]
```

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

```
A = [0.678519, 2.0, 0.526267, 0.0397595]
4-element Array{Float64,1}:
 0.6785190086558177
 2.0
 0.5262665389068844
 0.039759467636365065
```

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:  124.46 MiB
  allocs estimate:  8018
  --------------
  minimum time:     88.978 ms (33.65% GC)
  median time:      140.168 ms (33.24% GC)
  mean time:        139.466 ms (44.73% GC)
  maximum time:     211.669 ms (60.07% GC)
  --------------
  samples:          36
  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:  31.00 MiB
  allocs estimate:  6182
  --------------
  minimum time:     51.392 ms (2.21% GC)
  median time:      55.428 ms (2.68% GC)
  mean time:        58.663 ms (5.58% GC)
  maximum time:     139.395 ms (59.69% GC)
  --------------
  samples:          86
  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
```

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

```
BenchmarkTools.Trial:
  memory estimate:  30.89 MiB
  allocs estimate:  1590
  --------------
  minimum time:     45.123 ms (2.28% GC)
  median time:      48.890 ms (2.49% GC)
  mean time:        53.377 ms (5.66% GC)
  maximum time:     152.201 ms (59.39% GC)
  --------------
  samples:          94
  evals/sample:     1
```

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

```julia
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
```

```julia
    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:  30.87 MiB
  allocs estimate:  1287
  --------------
  minimum time:     9.242 ms (7.65% GC)
  median time:      10.742 ms (10.51% GC)
  mean time:        11.765 ms (14.34% GC)
  maximum time:     90.043 ms (88.00% GC)
  --------------
  samples:          425
```

```
  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 80x 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:  110985
  --------------
  minimum time:     2.426 s (23.30% GC)
  median time:      13.637 s (33.49% GC)
  mean time:        13.637 s (33.49% GC)
  maximum time:     24.847 s (34.49% GC)
  --------------
  samples:          2
  evals/sample:     1
```

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

```
BenchmarkTools.Trial:
  memory estimate:  116.36 MiB
  allocs estimate:  34146
  --------------
  minimum time:     583.201 ms (5.47% GC)
  median time:      690.583 ms (14.42% GC)
  mean time:        684.408 ms (15.17% GC)
  maximum time:     764.132 ms (20.60% GC)
  --------------
  samples:          8
  evals/sample:     1
```

```julia
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:  19.82 MiB
  allocs estimate:  693067
  --------------
  minimum time:     4.945 s (0.00% GC)
  median time:      5.053 s (0.02% GC)
  mean time:        5.053 s (0.02% GC)
  maximum time:     5.162 s (0.04% GC)
  --------------
  samples:          2
  evals/sample:     1
```

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

```
BenchmarkTools.Trial:
  memory estimate:  306.71 MiB
  allocs estimate:  90649
  --------------
  minimum time:     1.613 s (0.00% GC)
  median time:      1.741 s (6.26% GC)
  mean time:        1.732 s (7.03% GC)
  maximum time:     1.842 s (13.91% GC)
  --------------
  samples:          3
  evals/sample:     1
```

Now let's check the allocation growth.

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

```
BenchmarkTools.Trial:
  memory estimate:  4.04 MiB
  allocs estimate:  81374
  --------------
  minimum time:     1.456 s (0.00% GC)
  median time:      1.601 s (0.00% GC)
  mean time:        1.588 s (0.00% GC)
  maximum time:     1.693 s (0.00% GC)
  --------------
  samples:          4
  evals/sample:     1
```

```julia
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:  5.55 MiB
  allocs estimate:  120769
  --------------
  minimum time:     2.150 s (0.00% GC)
  median time:      2.221 s (0.00% GC)
  mean time:        2.236 s (0.00% GC)
  maximum time:     2.336 s (0.00% GC)
  --------------
  samples:          3
  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

```julia
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`
  [c52e3926] Atom v0.7.14
  [6e4b80f9] BenchmarkTools v0.4.2
  [336ed68f] CSV v0.4.3
  [be33ccc6] CUDAnative v1.0.1
  [3a865a2d] CuArrays v0.9.1
  [a93c6f00] DataFrames v0.17.1
  [39dd38d3] Dierckx v0.4.1
  [aae7a2af] DiffEqFlux v0.2.0
  [c894b116] DiffEqJump v6.1.0+ [`C:\Users\accou\.julia\dev\DiffEqJump`]
  [1130ab10] DiffEqParamEstim v1.5.1
  [225cb15b] DiffEqTutorials v0.0.0 [`C:\Users\accou\.julia\external\DiffEq
Tutorials.jl`]
  [0c46a032] DifferentialEquations v6.3.0
  [587475ba] Flux v0.7.3
  [f6369f11] ForwardDiff v0.10.3+ [`C:\Users\accou\.julia\dev\ForwardDiff`]
  [7073ff75] IJulia v1.17.0
  [c601a237] Interact v0.9.1
  [b6b21f68] Ipopt v0.5.4
  [4076af6c] JuMP v0.18.5
  [e5e0dc1b] Juno v0.5.4
  [76087f3c] NLopt v0.5.1
  [429524aa] Optim v0.17.2
  [1dea7af3] OrdinaryDiffEq v5.1.4+ [`C:\Users\accou\.julia\dev\OrdinaryDif
fEq`]
  [65888b18] ParameterizedFunctions v4.1.0
  [91a5bcdd] Plots v0.23.0
  [71ad9d73] PuMaS v0.0.0 [`C:\Users\accou\.julia\dev\PuMaS`]
  [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
  [44d3d7a6] Weave v0.7.1
```