An Intro to DifferentialEquations.jl

Chris Rackauckas

July 28, 2020

0.1 Basic Introduction Via Ordinary Differential Equations

This notebook will get you started with DifferentialEquations.jl by introducing you to the functionality for solving ordinary differential equations (ODEs). The corresponding documentation page is the ODE tutorial. While some of the syntax may be different for other types of equations, the same general principles hold in each case. Our goal is to give a gentle and thorough introduction that highlights these principles in a way that will help you generalize what you have learned.

0.1.1 Background

If you are new to the study of differential equations, it can be helpful to do a quick background read on the definition of ordinary differential equations. We define an ordinary differential equation as an equation which describes the way that a variable u changes, that is

$$u' = f(u, p, t)$$

where p are the parameters of the model, t is the time variable, and f is the nonlinear model of how u changes. The initial value problem also includes the information about the starting value:

$$u(t_0) = u_0$$

Together, if you know the starting value and you know how the value will change with time, then you know what the value will be at any time point in the future. This is the intuitive definition of a differential equation.

0.1.2 First Model: Exponential Growth

Our first model will be the canonical exponential growth model. This model says that the rate of change is proportional to the current value, and is this:

$$u' = au$$

where we have a starting value $u(0) = u_0$. Let's say we put 1 dollar into Bitcoin which is increasing at a rate of 98% per year. Then calling now $t = 0$ and measuring time in years, our model is:

$$u' = 0.98u$$

and $u(0) = 1.0$. We encode this into Julia by noticing that, in this setup, we match the general form when

```
f(u,p,t) = 0.98u
```

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

with $ u_0 = 1.0 $. If we want to solve this model on a time span from `t=0.0` to `t=1.0`, then we define an `ODEProblem` by specifying this function `f`, this initial condition `u0`, and this time span as follows:

```
using DifferentialEquations
f(u,p,t) = 0.98u
u0 = 1.0
tspan = (0.0,1.0)
prob = ODEProblem(f,u0,tspan)
```

```
ODEProblem with uType Float64 and tType Float64. In-place: false
timespan: (0.0, 1.0)
u0: 1.0
```

To solve our `ODEProblem` we use the command `solve`.

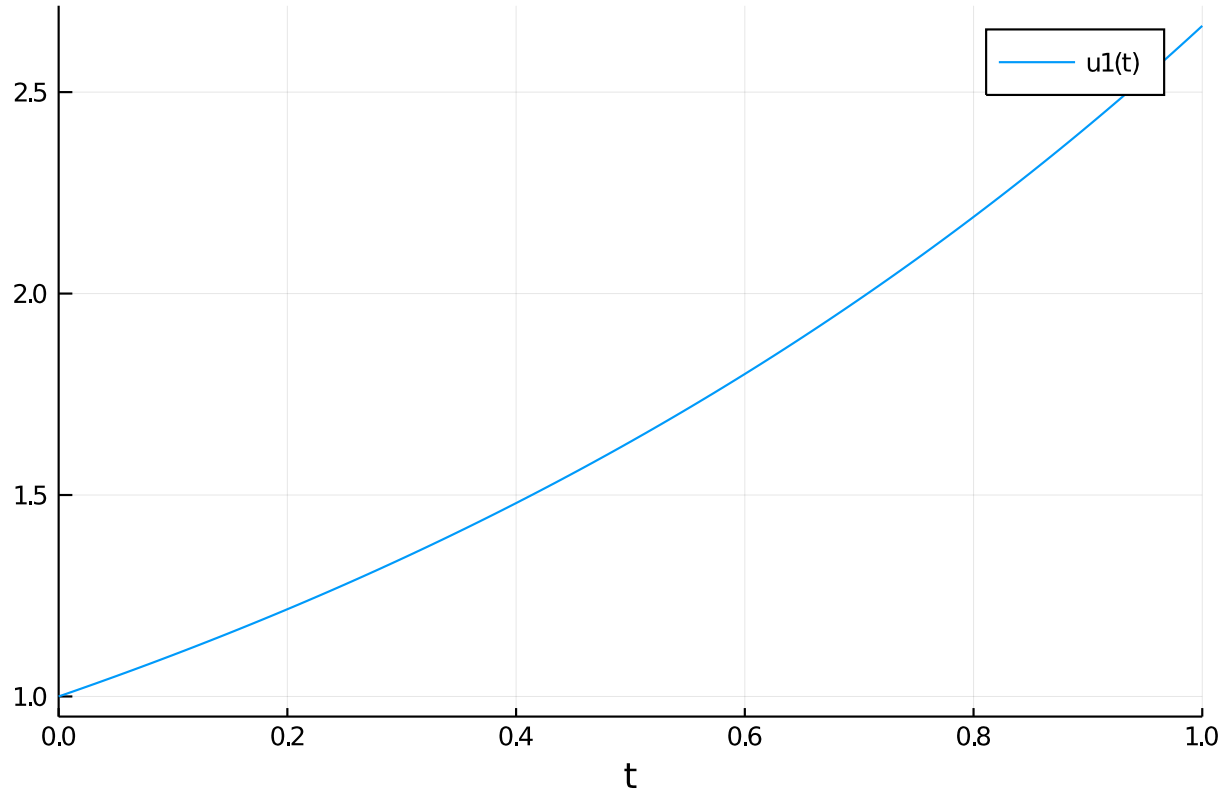
```
sol = solve(prob)
```

```
retcode: Success
Interpolation: automatic order switching interpolation
t: 5-element Array{Float64,1}:
 0.0
 0.10042494449239292
 0.35218603951893646
 0.6934436028208104
 1.0
u: 5-element Array{Float64,1}:
 1.0
 1.1034222047865465
 1.4121908848175448
 1.9730384275622996
 2.664456142481451
```

and that's it: we have succesfully solved our first ODE!

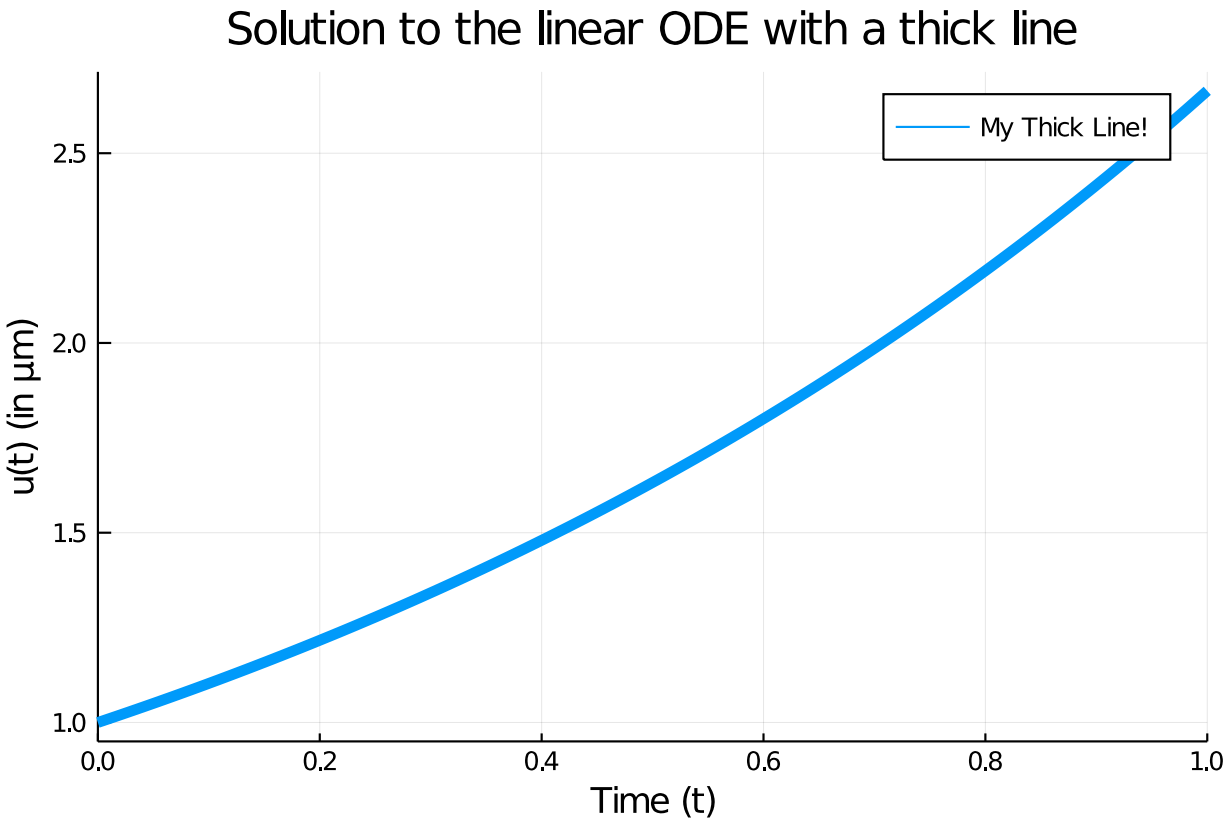
Analyzing the Solution Of course, the solution type is not interesting in and of itself. We want to understand the solution! The documentation page which explains in detail the functions for analyzing the solution is the Solution Handling page. Here we will describe some of the basics. You can plot the solution using the plot recipe provided by Plots.jl:

```
using Plots; gr()
plot(sol)
```



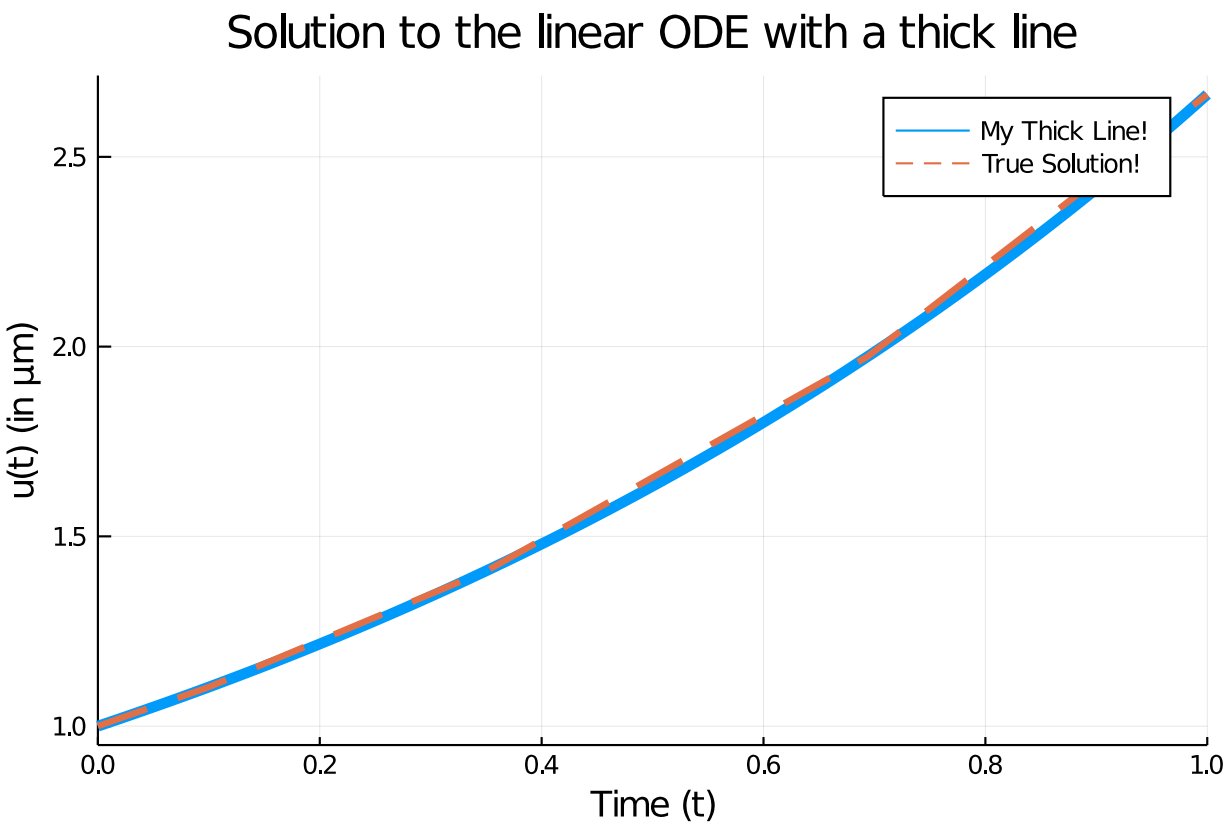
From the picture we see that the solution is an exponential curve, which matches our intuition. As a plot recipe, we can annotate the result using any of the Plots.jl attributes. For example:

```
plot(sol,linewidth=5,title="Solution to the linear ODE with a thick line",
     xaxis="Time (t)",yaxis="u(t) (in μm)",label="My Thick Line!") # legend=false
```



Using the mutating `plot!` command we can add other pieces to our plot. For this ODE we know that the true solution is $u(t) = u_0 exp(at)$, so let's add some of the true solution to our plot:

```
plot!(sol.t, t->1.0*exp(0.98t),lw=3,ls=:dash,label="True Solution!")
```



In the previous command I demonstrated `sol.t`, which grabs the array of time points that the solution was saved at:

```
sol.t
```

```
5-element Array{Float64,1}:
 0.0
 0.10042494449239292
 0.35218603951893646
 0.6934436028208104
 1.0
```

We can get the array of solution values using `sol.u`:

```
sol.u
```

```
5-element Array{Float64,1}:
 1.0
 1.1034222047865465
 1.4121908848175448
 1.9730384275622996
 2.664456142481451
```

`sol.u[i]` is the value of the solution at time `sol.t[i]`. We can compute arrays of functions of the solution values using standard comprehensions, like:

```
[t+u for (u,t) in tuples(sol)]
```

```
5-element Array{Float64,1}:
 1.0
 1.2038471492789395
 1.7643769243364813
 2.66648203038311
 3.664456142481451
```

However, one interesting feature is that, by default, the solution is a continuous function. If we check the print out again:

```
sol
```

```
retcode: Success
Interpolation: automatic order switching interpolation
t: 5-element Array{Float64,1}:
 0.0
 0.10042494449239292
 0.35218603951893646
 0.6934436028208104
 1.0
u: 5-element Array{Float64,1}:
 1.0
 1.1034222047865465
 1.4121908848175448
 1.9730384275622996
 2.664456142481451
```

you see that it says that the solution has a order changing interpolation. The default algorithm automatically switches between methods in order to handle all types of problems. For non-stiff equations (like the one we are solving), it is a continuous function of 4th order accuracy. We can call the solution as a function of time `sol(t)`. For example, to get the value at `t=0.45`, we can use the command:

```
sol(0.45)
```

```
1.554261048055312
```

Controlling the Solver DifferentialEquations.jl has a common set of solver controls among its algorithms which can be found at the Common Solver Options page. We will detail some of the most widely used options.

The most useful options are the tolerances `abstol` and `reltol`. These tell the internal adaptive time stepping engine how precise of a solution you want. Generally, `reltol` is the relative accuracy while `abstol` is the accuracy when `u` is near zero. These tolerances are local tolerances and thus are not global guarantees. However, a good rule of thumb is that the total solution accuracy is 1-2 digits less than the relative tolerances. Thus for the defaults `abstol=1e-6` and `reltol=1e-3`, you can expect a global accuracy of about 1-2 digits. If we want to get around 6 digits of accuracy, we can use the commands:

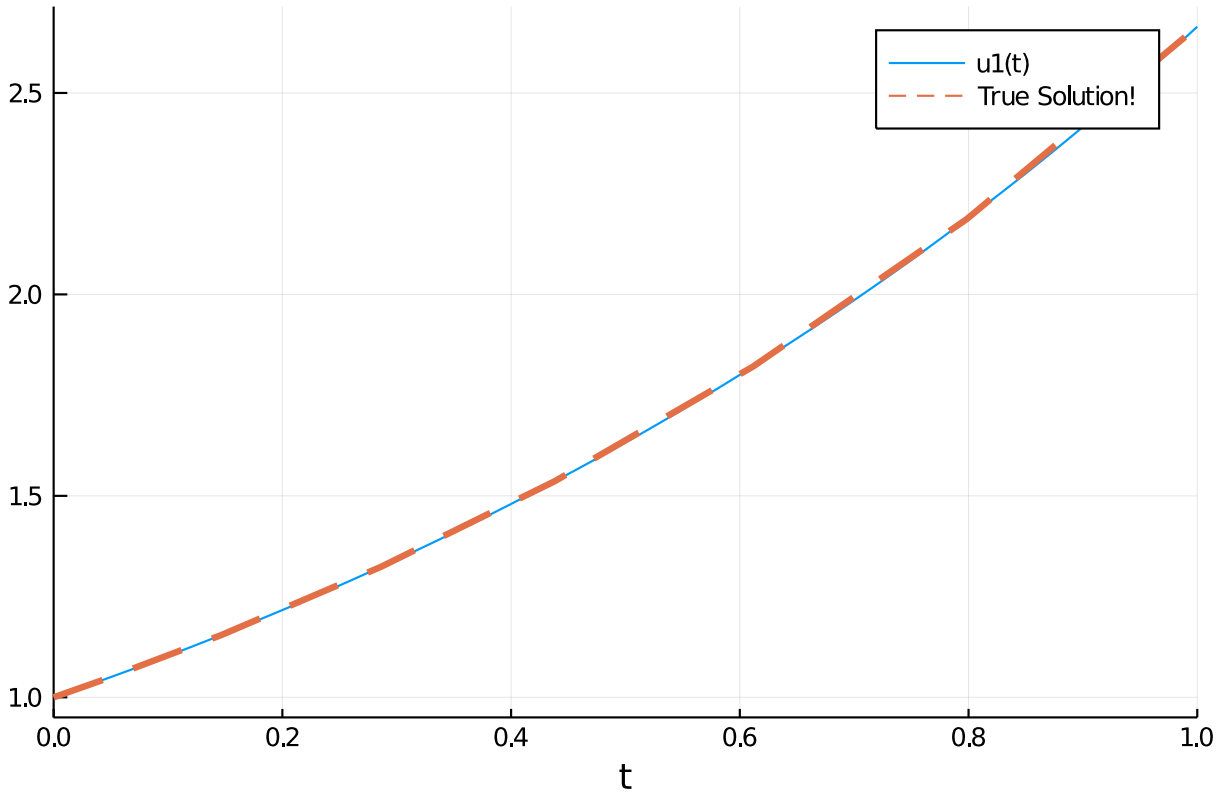
```
sol = solve(prob,abstol=1e-8,reltol=1e-8)
```

```
retcode: Success
Interpolation: automatic order switching interpolation
t: 9-element Array{Float64,1}:
 0.0
 0.04127492324135852
 0.14679917846877366
```

```
 0.28631546412766684
 0.4381941361169628
 0.6118924302028597
 0.7985659100883337
 0.9993516479536952
 1.0
u: 9-element Array{Float64,1}:
 1.0
 1.0412786454705882
 1.1547261252949712
 1.3239095703537043
 1.5363819257509728
 1.8214895157178692
 2.1871396448296223
 2.662763824115295
 2.664456241933517
```

Now we can see no visible difference against the true solution:

```
plot(sol)
plot!(sol.t, t->1.0*exp(0.98t),lw=3,ls=:dash,label="True Solution!")
```



Notice that by decreasing the tolerance, the number of steps the solver had to take was `9` instead of the previous `5`. There is a trade off between accuracy and speed, and it is up to you to determine what is the right balance for your problem.

Another common option is to use `saveat` to make the solver save at specific time points. For example, if we want the solution at an even grid of `t=0.1k` for integers `k`, we would use the command:

```
sol = solve(prob,saveat=0.1)
```

```
retcode: Success
```

```
Interpolation: 1st order linear
t: 11-element Array{Float64,1}:
 0.0
 0.1
 0.2
 0.3
 0.4
 0.5
 0.6
 0.7
 0.8
 0.9
 1.0
u: 11-element Array{Float64,1}:
 1.0
 1.102962785129292
 1.2165269512238264
 1.341783821227542
 1.4799379510586077
 1.632316207054161
 1.8003833264983584
 1.9857565541588758
 2.1902158127997695
 2.415725742084496
 2.664456142481451
```

Notice that when `saveat` is used the continuous output variables are no longer saved and thus `sol(t)`, the interpolation, is only first order. We can save at an uneven grid of points by passing a collection of values to `saveat`. For example:

```
sol = solve(prob,saveat=[0.2,0.7,0.9])
```

```
retcode: Success
Interpolation: 1st order linear
t: 3-element Array{Float64,1}:
 0.2
 0.7
 0.9
u: 3-element Array{Float64,1}:
 1.2165269512238264
 1.9857565541588758
 2.415725742084496
```

If we need to reduce the amount of saving, we can also turn off the continuous output directly via `dense=false`:

```
sol = solve(prob,dense=false)
```

```
retcode: Success
Interpolation: 1st order linear
t: 5-element Array{Float64,1}:
 0.0
 0.10042494449239292
 0.35218603951893646
 0.6934436028208104
 1.0
u: 5-element Array{Float64,1}:
 1.0
 1.1034222047865465
 1.4121908848175448
```

```
 1.9730384275622996
 2.664456142481451
```

and to turn off all intermediate saving we can use `save_everystep=false`:

```
sol = solve(prob,save_everystep=false)
```

```
retcode: Success
Interpolation: 1st order linear
t: 2-element Array{Float64,1}:
 0.0
 1.0
u: 2-element Array{Float64,1}:
 1.0
 2.664456142481451
```

If we want to solve and only save the final value, we can even set `save_start=false`.

```
sol = solve(prob,save_everystep=false,save_start = false)
```

```
retcode: Success
Interpolation: 1st order linear
t: 1-element Array{Float64,1}:
 1.0
u: 1-element Array{Float64,1}:
 2.664456142481451
```

Note that similarly on the other side there is `save_end=false`.

More advanced saving behaviors, such as saving functionals of the solution, are handled via the `SavingCallback` in the Callback Library which will be addressed later in the tutorial.

Choosing Solver Algorithms There is no best algorithm for numerically solving a differential equation. When you call `solve(prob)`, DifferentialEquations.jl makes a guess at a good algorithm for your problem, given the properties that you ask for (the tolerances, the saving information, etc.). However, in many cases you may want more direct control. A later notebook will help introduce the various *algorithms* in DifferentialEquations.jl, but for now let's introduce the *syntax*.

The most crucial determining factor in choosing a numerical method is the stiffness of the model. Stiffness is roughly characterized by a Jacobian `f` with large eigenvalues. That's quite mathematical, and we can think of it more intuitively: if you have big numbers in `f` (like parameters of order `1e5`), then it's probably stiff. Or, as the creator of the MATLAB ODE Suite, Lawrence Shampine, likes to define it, if the standard algorithms are slow, then it's stiff. We will go into more depth about diagnosing stiffness in a later tutorial, but for now note that if you believe your model may be stiff, you can hint this to the algorithm chooser via `alg_hints = [:stiff]`.

```
sol = solve(prob,alg_hints=[:stiff])
```

```
retcode: Success
Interpolation: specialized 3rd order "free" stiffness-aware interpolation
t: 8-element Array{Float64,1}:
 0.0
 0.05653299582822294
 0.17270731152826024
 0.3164602871490142
```

```
 0.5057500163821153
 0.7292241858994543
 0.9912975001018789
 1.0
u: 8-element Array{Float64,1}:
 1.0
 1.0569657840332976
 1.1844199383303913
 1.3636037723365293
 1.6415399686182572
 2.0434491434754793
 2.641825616057761
 2.6644526430553817
```

Stiff algorithms have to solve implicit equations and linear systems at each step so they should only be used when required.

If we want to choose an algorithm directly, you can pass the algorithm type after the problem as `solve(prob,alg)`. For example, let's solve this problem using the `Tsit5()` algorithm, and just for show let's change the relative tolerance to `1e-6` at the same time:

```
sol = solve(prob,Tsit5(),reltol=1e-6)
```

```
retcode: Success
Interpolation: specialized 4th order "free" interpolation
t: 10-element Array{Float64,1}:
 0.0
 0.028970819746309166
 0.10049147151547619
 0.19458908698515082
 0.3071725081673423
 0.43945421453622546
 0.5883434923759523
 0.7524873357619015
 0.9293021330536031
 1.0
u: 10-element Array{Float64,1}:
 1.0
 1.0287982807225062
 1.1034941463604806
 1.2100931078233779
 1.351248605624241
 1.538280340326815
 1.7799346012651116
 2.090571742234628
 2.486102171447025
 2.6644562434913377
```

0.1.3 Systems of ODEs: The Lorenz Equation

Now let's move to a system of ODEs. The Lorenz equation is the famous "butterfly attractor" that spawned chaos theory. It is defined by the system of ODEs:

$$\frac{dx}{dt} = \sigma(y - x) \tag{1}$$
$$\frac{dy}{dt} = x(\rho - z) - y \tag{2}$$
$$\frac{dz}{dt} = xy - \beta z \tag{3}$$

To define a system of differential equations in DifferentialEquations.jl, we define our `f` as a vector function with a vector initial condition. Thus, for the vector `u = [x,y,z]'`, we have the derivative function:

```
function lorenz!(du,u,p,t)
    σ,ρ,β = p
    du[1] = σ*(u[2]-u[1])
    du[2] = u[1]*(ρ-u[3]) - u[2]
    du[3] = u[1]*u[2] - β*u[3]
end
```

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

Notice here we used the in-place format which writes the output to the preallocated vector `du`. For systems of equations the in-place format is faster. We use the initial condition $u_0 = [1.0, 0.0, 0.0]$ as follows:

```
u0 = [1.0,0.0,0.0]
```

```
3-element Array{Float64,1}:
 1.0
 0.0
 0.0
```

Lastly, for this model we made use of the parameters `p`. We need to set this value in the `ODEProblem` as well. For our model we want to solve using the parameters $\sigma = 10$, $\rho = 28$, and $\beta = 8/3$, and thus we build the parameter collection:

```
p = (10,28,8/3) # we could also make this an array, or any other type!
```

```
(10, 28, 2.6666666666666665)
```

Now we generate the `ODEProblem` type. In this case, since we have parameters, we add the parameter values to the end of the constructor call. Let's solve this on a time span of `t=0` to `t=100`:

```
tspan = (0.0,100.0)
prob = ODEProblem(lorenz!,u0,tspan,p)
```

```
ODEProblem with uType Array{Float64,1} and tType Float64. In-place: true
timespan: (0.0, 100.0)
u0: [1.0, 0.0, 0.0]
```

Now, just as before, we solve the problem:

```
sol = solve(prob)
```

```
retcode: Success
Interpolation: automatic order switching interpolation
t: 1294-element Array{Float64,1}:
```

```
   0.0
   3.5678604836301404e-5
   0.0003924646531993154
   0.0032624077544510573
   0.009058075635317072
   0.01695646895607931
   0.0276899566248403
   0.041856345938267966
   0.06024040228733675
   0.08368539694547242
   ⋮
  99.39403070915297
  99.47001147494375
  99.54379656909015
  99.614651558349
  99.69093823148101
  99.78733023233721
  99.86114450046736
  99.96115759510786
 100.0
u: 1294-element Array{Array{Float64,1},1}:
 [1.0, 0.0, 0.0]
 [0.9996434557625105, 0.0009988049817849058, 1.781434788799208e-8]
 [0.9961045497425811, 0.010965399721242457, 2.146955365838907e-6]
 [0.9693591634199452, 0.08977060667778931, 0.0001438018342266937]
 [0.9242043615038835, 0.24228912482984957, 0.0010461623302512404]
 [0.8800455868998046, 0.43873645009348244, 0.0034242593451028745]
 [0.8483309877783048, 0.69156288756671, 0.008487623500490047]
 [0.8495036595681027, 1.0145425335433382, 0.01821208597613427]
 [0.9139069079152129, 1.4425597546855036, 0.03669381053327124]
 [1.0888636764765296, 2.052326153029042, 0.07402570506414284]
 ⋮
 [12.999157033749652, 14.10699925404482, 31.74244844521858]
 [11.646131422021162, 7.2855792145502845, 35.365000488215486]
 [7.777555445486692, 2.5166095828739574, 32.030953593541675]
 [4.739741627223412, 1.5919220588229062, 27.249779003951755]
 [3.2351668945618774, 2.3121727966182695, 22.724936101772805]
 [3.310411964698304, 4.28106626744641, 18.435441144016366]
 [4.527117863517627, 6.895878639772805, 16.58544600757436]
 [8.043672261487556, 12.711555298531689, 18.12537420595938]
 [9.97537965430362, 15.143884806010783, 21.00643286956427]
```

The same solution handling features apply to this case. Thus `sol.t` stores the time points and `sol.u` is an array storing the solution at the corresponding time points.

However, there are a few extra features which are good to know when dealing with systems of equations. First of all, `sol` also acts like an array. `sol[i]` returns the solution at the `i`th time point.

```
sol.t[10],sol[10]
```

```
(0.08368539694547242, [1.0888636764765296, 2.052326153029042, 0.07402570506
414284])
```

Additionally, the solution acts like a matrix where `sol[j,i]` is the value of the `j`th variable at time `i`:

```
sol[2,10]
```

```
2.052326153029042
```

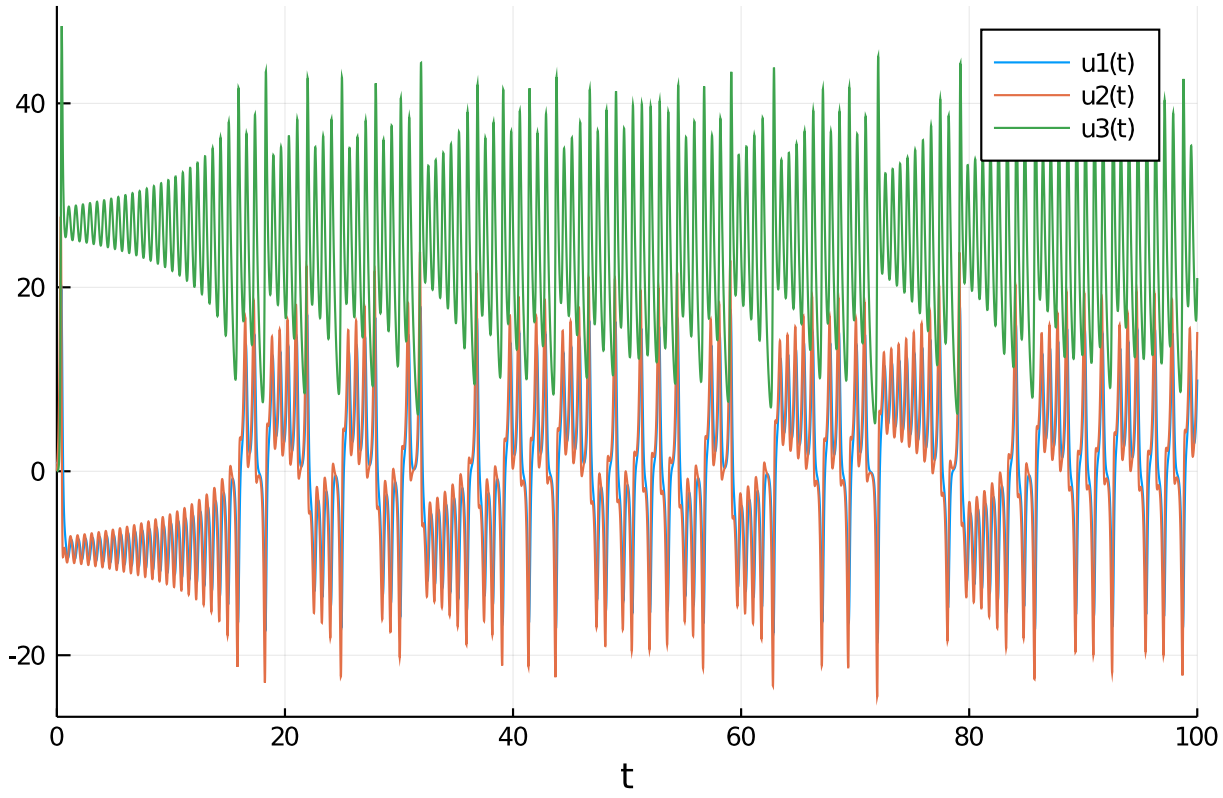
We can get a real matrix by performing a conversion:

```
A = Array(sol)
```

```
3×1294 Array{Float64,2}:
 1.0  0.999643     0.996105    0.969359     ...   4.52712   8.04367   9.97538
 0.0  0.000998805  0.0109654   0.0897706          6.89588  12.7116   15.1439
 0.0  1.78143e-8   2.14696e-6  0.000143802       16.5854   18.1254   21.0064
```

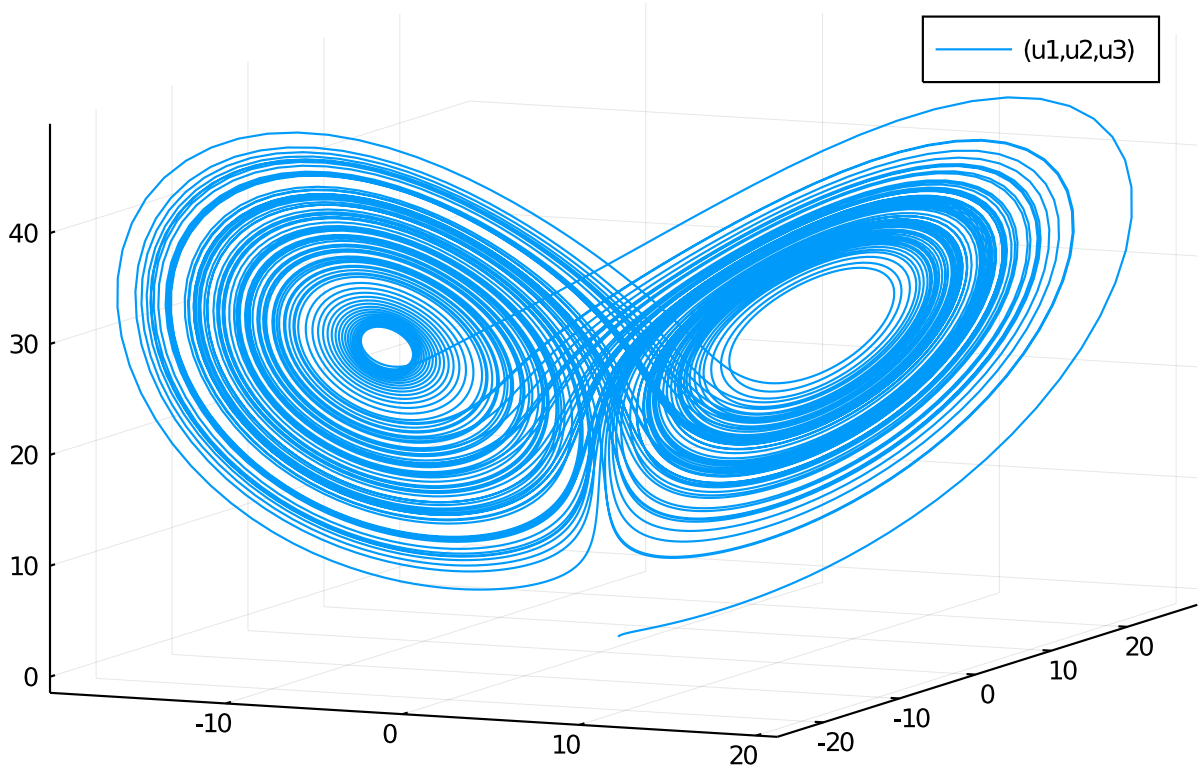
This is the same as sol, i.e. `sol[i,j] = A[i,j]`, but now it's a true matrix. Plotting will by default show the time series for each variable:

```
plot(sol)
```



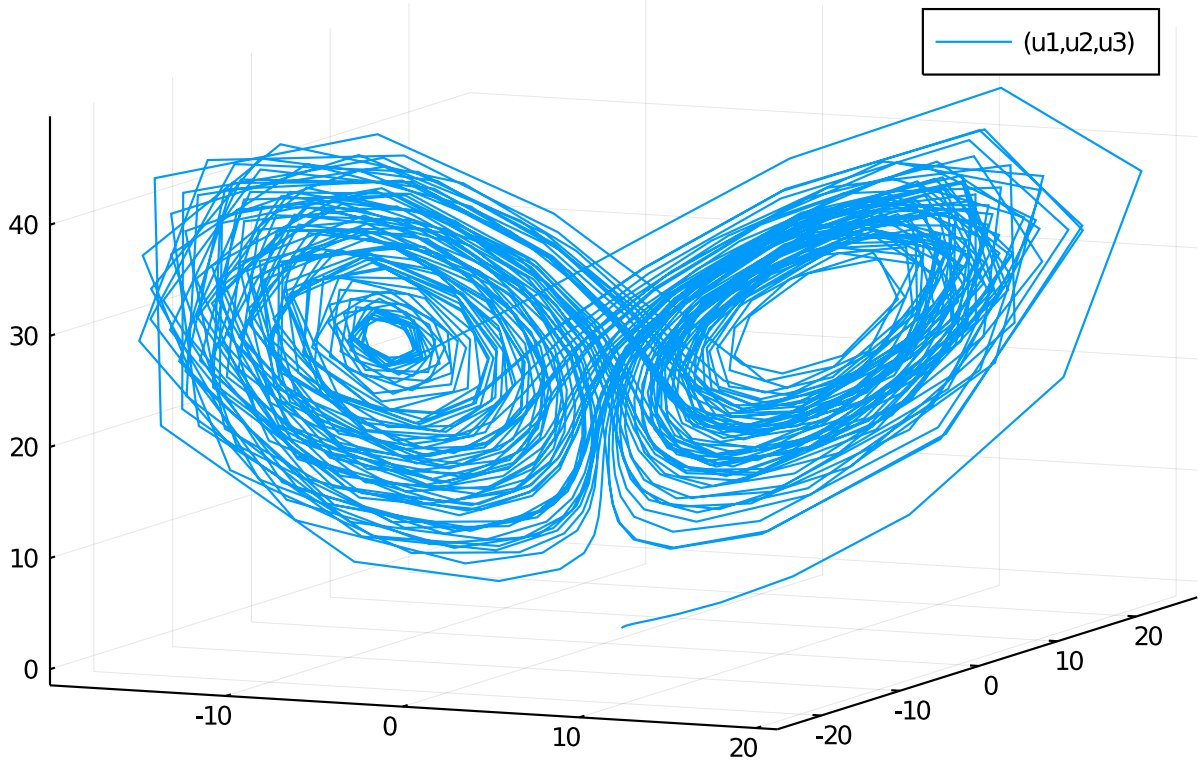
If we instead want to plot values against each other, we can use the `vars` command. Let's plot variable 1 against variable 2 against variable 3:

```
plot(sol,vars=(1,2,3))
```



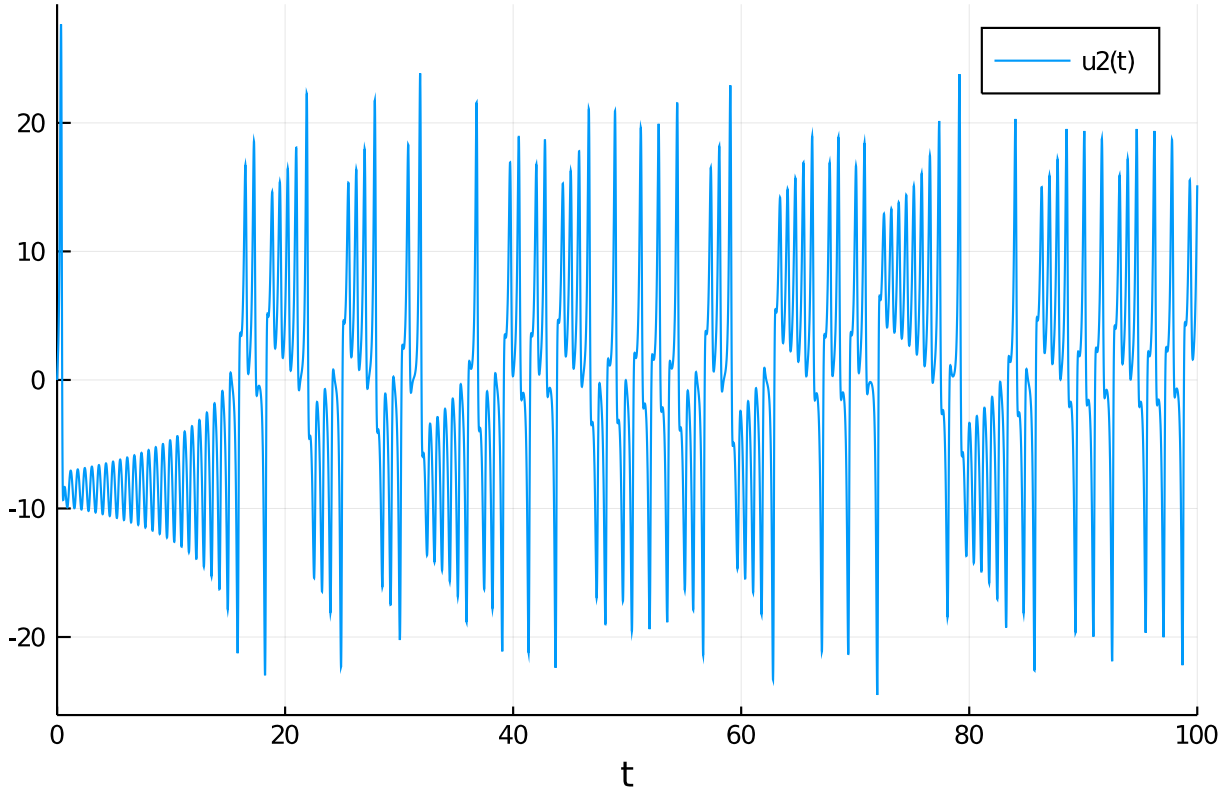
This is the classic Lorenz attractor plot, where the `x` axis is `u[1]`, the `y` axis is `u[2]`, and the `z` axis is `u[3]`. Note that the plot recipe by default uses the interpolation, but we can turn this off:

```
plot(sol,vars=(1,2,3),denseplot=false)
```



Yikes! This shows how calculating the continuous solution has saved a lot of computational effort by computing only a sparse solution and filling in the values! Note that in vars, `0=time`, and thus we can plot the time series of a single component like:

```
plot(sol,vars=(0,2))
```



0.2 Internal Types

The last basic user-interface feature to explore is the choice of types. DifferentialEquations.jl respects your input types to determine the internal types that are used. Thus since in the previous cases, when we used `Float64` values for the initial condition, this meant that the internal values would be solved using `Float64`. We made sure that time was specified via `Float64` values, meaning that time steps would utilize 64-bit floats as well. But, by simply changing these types we can change what is used internally.

As a quick example, let's say we want to solve an ODE defined by a matrix. To do this, we can simply use a matrix as input.

```
A  = [1. 0  0 -5
      4 -2  4 -3
     -4  0  0  1
      5 -2  2  3]
u0 = rand(4,2)
tspan = (0.0,1.0)
f(u,p,t) = A*u
prob = ODEProblem(f,u0,tspan)
sol = solve(prob)
```

```
retcode: Success
Interpolation: automatic order switching interpolation
```

```
t: 10-element Array{Float64,1}:
 0.0
 0.0431583364054055
 0.1153143854351763
 0.19765891776727906
 0.30243489494902526
 0.42252610292262877
 0.5574409597445212
 0.7002286201489554
 0.8534607321245409
 1.0
u: 10-element Array{Array{Float64,2},1}:
 [0.3984819530794539 0.9637556818431008; 0.9147620825925182 0.3617522521140
6525; 0.6267976723471336 0.8271326495338405; 0.8092339752422932 0.955089117
4747071]
 [0.22001705043861136 0.7554289557732575; 0.8819793562014258 0.461039941018
5265; 0.6112020571939 0.7266240213731241; 0.9671432081880452 1.319994570880
869]
 [-0.16809430936651382 0.2115347979002603; 0.7235703793558566 0.39397555237
553544; 0.6792390606369912 0.6972745062519699; 1.188444298085305 1.88520855
26201756]
 [-0.7318934827269516 -0.6982058934855276; 0.4255744626283403 0.01107806018
4708305; 0.9297752436191355 0.9476196735462834; 1.3574911015821431 2.423677
918513722]
 [-1.585613689553726 -2.248520937952204; -0.029183271841557168 -0.796385494
9827429; 1.5580103553669167 1.8311908363596512; 1.408027829478332 2.8655823
25376089]
 [-2.6340523430881877 -4.406959946722383; -0.4470237581814744 -1.8201229602
413465; 2.730774354780132 3.7705886009900373; 1.1852770914783142 2.90727113
03547463]
 [-3.6615802691423376 -6.941053615430349; -0.4371406305842422 -2.4506613625
61908; 4.563499899394147 7.192553793559531; 0.5125173036142324 2.1795193612
067223]
 [-4.186840426001786 -9.062005469554194; 0.5694137887890667 -1.586245349492
7955; 6.83444470063553 12.004139127139837; -0.7290037820542522 0.3307350388
000838]
 [-3.5460304772885167 -9.592000913584338; 3.232909106762804 2.2217307543097
58; 9.033119728808751 17.660432918400907; -2.6303279546178624 -2.9885088035
790863]
 [-1.2003357638717973 -7.141462398459231; 7.498435076722952 9.4767166751888
1; 9.976173736001996 21.993644097140702; -4.824170381902354 -7.327832194928
521]
```

There is no real difference from what we did before, but now in this case `u0` is a `4x2` matrix. Because of that, the solution at each time point is matrix:

```
sol[3]
```

```
4×2 Array{Float64,2}:
 -0.168094  0.211535
  0.72357   0.393976
  0.679239  0.697275
  1.18844   1.88521
```

In DifferentialEquations.jl, you can use any type that defines `+`, `-`, `*`, `/`, and has an appropriate `norm`. For example, if we want arbitrary precision floating point numbers, we can change the input to be a matrix of `BigFloat`:

```
big_u0 = big.(u0)
```

```
4×2 Array{BigFloat,2}:
 0.398482  0.963756
 0.914762  0.361752
 0.626798  0.827133
 0.809234  0.955089
```

and we can solve the `ODEProblem` with arbitrary precision numbers by using that initial condition:

```
prob = ODEProblem(f,big_u0,tspan)
sol = solve(prob)
```

```
retcode: Success
Interpolation: automatic order switching interpolation
t: 5-element Array{Float64,1}:
 0.0
 0.15945699885551542
 0.42119133406156783
 0.7216495970126032
 1.0
u: 5-element Array{Array{BigFloat,2},1}:
 [0.39848195307945388776715844869613647460937 5 0.9637556818431007865655146815697662532329559326171875; 0.914762082592518233070677524665370583534240722
65625 0.36175225211406525360757768794428557157516479492187 5; 0.626797672347
13357204611838824348524212837219238281 25 0.82713264953384046940243479184573
51624965667724609375; 0.80923397524229323529709745344007387757301330566406 2
5 0.95508911747470714992402918142033740878105163574218 75]
 [-0.45591885212764269225572966168770064607752744364675493594411188586541089
60309203 -0.23925412414778976888604263078459408735189049808722876519896737
92749897577090273; 0.57568070800547095036006153894872899781582763974737449 7
31882033938743501463376 65 0.22436991924335785137793240243117366222573359599
82229576395335922475501361542198; 0.78864464676737598575545150397602396670 7
92946245384410255588233473416511458664 38 0.78853508225456409849798808480603
21774699109379376158852697107099043622962496344; 1.29154231733348365696144 1
90106117802624908105477398956574425429132119622288961 2.19100867583406217321
51673284734873053193549689196903413042620626411370691829232]
 [-2.62261972515230850093840947846558561882919008406712206909826526416022636
77383755 -4.38168344200213084143504297663206203215811701738019039348766865 4
70991763579304; -0.44396293808566942978278119303153645546431749619491129674
15261542302583438074556 -1.80991728651671186118556635940299153739403821065 0
31563701584969223287248326382; 2.71515735163651600667205982451357549417107 3
96449181269145099166522153186981791 3.74324621797609851718739226290997786 7
52318400103534601829448181480835594381392; 1.18960488984923579231845232529 8
78012745239629705630742003981400744986823623229 6 2.91007643810965467291900 5
68764392422009106226278230709866285053585897605139 2051]
 [-4.18612530017745638337939505212036416639894284914392094829888188404166691
28136406 -9.27373429242803612111862585579551601959478878510323459802151152 1
845441909424841; 0.83506057611031913375931628649143840598748170631048698679
06482720847223415604546 -1.25777932786415917945634413665548125422817781127 8
77363083566224449004217907921 9; 7.17525503088152160742854614009519606729644
41081267698680341741472073193003465 56 12.79300489363077550969337471126600 14
31644085448038582349828520373068756446757537; -0.96173388623208434034515947 8
19446164373418122688036102989550813475885742899235 61 -0.0500723641690716471
20238383273898031472801024383020086724761916461883777237056 73]
 [-1.20032360556571634029020912668472763605385157876750733075459961405866808
9409039 -7.14145111105911317845295072273456109692622913116387906949306673 9
74359901464314; 7.49846173390919422861639650966357051288800320656290042514 8
571916254283480153233 9.47676289102591055389671898425963188551599613039327 8
28905368751136663516878438 7; 9.97618115678731426970515029422331603701058582
32244101788384638649174403993741 08 21.99367341626346221746286889079600495 12
```

```
9850870937463864938968901708055930702034; -4.824183122371212072227360332052
4666691346231240962648694151040392053878042168 2 -7.327856418070343007524586
3847492150335240385965037332666246711494567396804262 99]

sol[1,3]

-2.622619725152308500938409478465585618829190084067122069098265264160226377
383755
```

To really make use of this, we would want to change `abstol` and `reltol` to be small! Notice that the type for "time" is different than the type for the dependent variables, and this can be used to optimize the algorithm via keeping multiple precisions. We can convert time to be arbitrary precision as well by defining our time span with `BigFloat` variables:

```
prob = ODEProblem(f,big_u0,big.(tspan))
sol = solve(prob)

retcode: Success
Interpolation: automatic order switching interpolation
t: 5-element Array{BigFloat,1}:
 0.0
 0.159456998855515426709261384172414149860661134579339478744415384034460600
2778608
 0.421191334061567844963495054061999650603920184392629107978780523761969483
5843882
 0.721649597012603220345202322675873150865147589172971373295652499255410274
0336311
 1.0
u: 5-element Array{Array{BigFloat,2},1}:
 [0.39848195307945388776715844869613647460937 5 0.963755681843100786565514681
569766253232955932617187 5; 0.914762082592518233070677524665370583534240722
65625 0.361752252114065253607577687944285571575164794921875; 0.626797672347
133572046118388243485242128372192382812 5 0.827132649533840469402434791845735
516249656677246093 75; 0.809233975242293235297097453440073877573013305664062
5 0.955089117474707149924029181420337408781051635742187 5]
 [-0.455918852127642704335008587724700623372863841999295329146632275621075716
9659049 -0.23925412414778978844436920353090492924642690660131104956243681
29169838672690548; 0.575680708005470943904124236429961281039199658941607028
39509109846593634325494 96 0.224369919243357842948468703168115903712130796785
104586070734530538808784543129 2; 0.788644646767375991198262507974780657916
329457941303844044136155689394231306439 9 0.788535082254564103998119234934992
102257792888977436060428428699585823587386263 3; 1.291542317333483660492401
015135380051822011550541205256990711906753387340738451 2.191008675834062184
617155674946831285941193815996142638905265636076320282061506]
 [-2.622619725152308350819416668846640215725423734135839935310416322547484614545146 -4.381683442002130509830722414171852300176636045983037838290572305
43392124969933; -0.443962938085669389309081574675566266155579284398225626284084466018780350992552 5 -1.809917286516711726944376045847950456384973605028
477968037777264386758724255558; 2.715157351636515802089720546222019834982935344845888198045046179773499455339415 3.743246217976098159226735604460237369178260806734825369832371545109872575795371; 1.189604889849235848824169776451409082891104405578647439457368192403923338004215 2.910076438109654709208657440596381942459281298127450159570280037203074223257 01]
 [-4.186125300177456405283688186803626872676754056935339341750554719621136264397857 -9.273734292428035803630479868738691364749289897784424652157584494669149055985657; 0.835060576110318670315370860302872216690315249723578655736902649051390230 19 -1.257779327864159768555379730930248584960214176719269301734507968531831888559079; 7.175255030881521052110458181480522402729295347251237464430077715176603086044876 12.79300489363077420626645055384150838
```

```
971626173359759852898690780536123583378086; -0.961733886232083948543709453091426982135800188140346374228933382561651728593972
9 -0.05007236416907099910513736450156282467410357841653594899655130638662799033116438]
 [-1.200323605565716976464445605453444964185546863931585045140714013127372514347568 -7.1414511110591139971822661544495810474774575796756219744292801715534566173987
76; 7.498461733909193268854806141739759814758758899530854424408712552013123642771748 9.476762891025908820868209771595847126364186809031305131839443038255481030395209; 9.976181156787314270340480207107317916743025641391229614680709427566701408774721.9936734162634616279914771239299574670941443103695507891454041556823202861224 4; -4.824183122371211641496129891885934614869949633338509233639636010815287788431156 -7.32785641807034210111066748069435513496803104403149552280794000060251344993339 3]
```

Let's end by showing a more complicated use of types. For small arrays, it's usually faster to do operations on static arrays via the package StaticArrays.jl. The syntax is similar to that of normal arrays, but for these special arrays we utilize the `@SMatrix` macro to indicate we want to create a static array.

```
using StaticArrays
A  = @SMatrix [ 1.0  0.0 0.0 -5.0
                4.0 -2.0 4.0 -3.0
               -4.0  0.0 0.0  1.0
                5.0 -2.0 2.0  3.0]
u0 = @SMatrix rand(4,2)
tspan = (0.0,1.0)
f(u,p,t) = A*u
prob = ODEProblem(f,u0,tspan)
sol = solve(prob)

retcode: Success
Interpolation: automatic order switching interpolation
t: 11-element Array{Float64,1}:
 0.0
 0.007163927323932619
 0.04615872311533139
 0.11560313880988551
 0.19730087428935955
 0.29141428170770495
 0.4020236545779373
 0.5329722383938044
 0.6718148245305243
 0.8396162958261761
 1.0
u: 11-element Array{StaticArrays.SArray{Tuple{4,2},Float64,2,8},1}:
 [0.010464809441216527 0.9352435235736136; 0.9228135670166842 0.4997511896357640
4; 0.25187072683703304 0.5209977130569305; 0.8953651538109129 0.4072902249431578]
 [-0.021823451543931616 0.9265627065592806; 0.8975715311192906 0.5244701530463395
; 0.25848166888748125 0.49738837603506664; 0.9051253450889294 0.4498048962770293]
 [-0.20727560906409667 0.851266068473453; 0.7510290472074644 0.6207653806299068
; 0.31235022214676467 0.3801400524464554; 0.9493515805551066 0.6789706197675234]
 [-0.5719543793632117 0.5980433770002167; 0.46495118838857386 0.6367196678385356
; 0.4872511896220083 0.23618313730943863; 0.987532015816827 1.069784384265273]
 [-1.037698652148758 0.10524211021905411; 0.12567951938435745 0.4230544112887340
6; 0.8295910969715014 0.22020152396923182; 0.9573434431961764 1.4791583951772 86]
```

```
 [-1.581660821715615 -0.7083564551546928; -0.1967274635514562 -0.0698937928
9983344; 1.407318651635367 0.4833618615068218; 0.8086781833186691 1.8435113
091016242]
 [-2.1493504019812484 -1.944239057896934; -0.36134490167315436 -0.835074032
8184774; 2.309470758683357 1.278996043289152; 0.4615498439843619 2.05946841
6464487]
 [-2.561452699875834 -3.6353405906170813; -0.0650855888861212 -1.6699773509
167564; 3.579756544163601 3.0001673931104116; -0.20371156834908782 1.911856
1388323214]
 [-2.447007197602169 -5.365124275834457; 1.0450947549699208 -1.935677165253
8507; 4.9091225070456606 5.733120257263439; -1.1969707674270826 1.147291203
2540945]
 [-1.1555184292555893 -6.633300097217812; 3.655133504009725 -0.497832090239
8792; 5.877520119948311 9.885452566028606; -2.68804579751542 -0.75576336517
16178]
 [1.6144357555953899 -5.998624122824556; 7.315665827543399 3.54165628381118
67; 5.266445686065155 13.728275763610508; -4.159512101820095 -3.58613086138
64566]

sol[3]

4×2 StaticArrays.SArray{Tuple{4,2},Float64,2,8} with indices SOneTo(4)×SOne
To(2):
 -0.207276  0.851266
  0.751029  0.620765
  0.31235   0.38014
  0.949352  0.678971
```

0.3 Conclusion

These are the basic controls in DifferentialEquations.jl. All equations are defined via a problem type, and the `solve` command is used with an algorithm choice (or the default) to get a solution. Every solution acts the same, like an array `sol[i]` with `sol.t[i]`, and also like a continuous function `sol(t)` with a nice plot command `plot(sol)`. The Common Solver Options can be used to control the solver for any equation type. Lastly, the types used in the numerical solving are determined by the input types, and this can be used to solve with arbitrary precision and add additional optimizations (this can be used to solve via GPUs for example!). While this was shown on ODEs, these techniques generalize to other types of equations as well.

0.4 Appendix

This tutorial is part of the SciMLTutorials.jl repository, found at: https://github.com/SciMLTutorials/Sci
For more information on doing scientific machine learning (SciML) with open source software, check out https://sciml.ai/.

To locally run this tutorial, do the following commands:

```
using SciMLTutorials
SciMLTutorials.weave_file("introduction","01-ode_introduction.jmd")
```

Computer Information:

```
Julia Version 1.4.2
Commit 44fa15b150* (2020-05-23 18:35 UTC)
Platform Info:
  OS: Linux (x86_64-pc-linux-gnu)
  CPU: Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-8.0.1 (ORCJIT, skylake)
Environment:
  JULIA_LOAD_PATH = /builds/JuliaGPU/DiffEqTutorials.jl:
  JULIA_DEPOT_PATH = /builds/JuliaGPU/DiffEqTutorials.jl/.julia
  JULIA_CUDA_MEMORY_LIMIT = 2147483648
  JULIA_NUM_THREADS = 8
```

Package Information:

```
Status `/builds/JuliaGPU/DiffEqTutorials.jl/tutorials/introduction/Project.toml`
[6e4b80f9-dd63-53aa-95a3-0cdb28fa8baf] BenchmarkTools 0.5.0
[0c46a032-eb83-5123-abaf-570d42b7fbaa] DifferentialEquations 6.15.0
[65888b18-ceab-5e60-b2b9-181511a3b968] ParameterizedFunctions 5.4.0
[91a5bcdd-55d7-5caf-9e0b-520d859cae80] Plots 1.5.6
[90137ffa-7385-5640-81b9-e52037218182] StaticArrays 0.12.4
[c3572dad-4567-51f8-b174-8c6c989267f4] Sundials 4.2.5
[37e2e46d-f89d-539d-b4ee-838fcccc9c8e] LinearAlgebra
```