An Intro to DifferentialEquations.jl

Chris Rackauckas

June 15, 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)
```

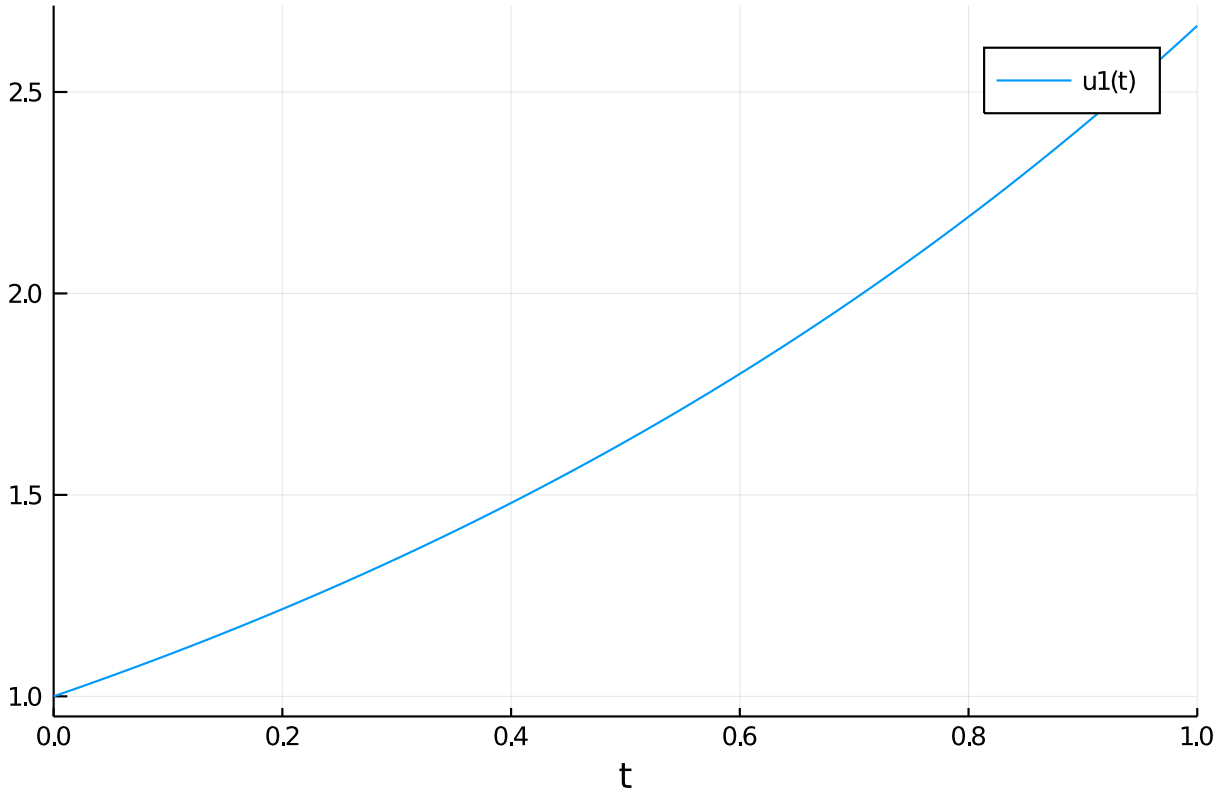
```
retcode: Success
Interpolation: Automatic order switching interpolation
t: 5-element Array{Float64,1}:
 0.0
 0.10042494449239292
 0.35218603951893646
 0.6934436028208104
 1.0
u: 5-element Array{Float64,1}:
 1.0
 1.1034222047865465
 1.4121908848175448
 1.9730384275622996
 2.664456142481451
```

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

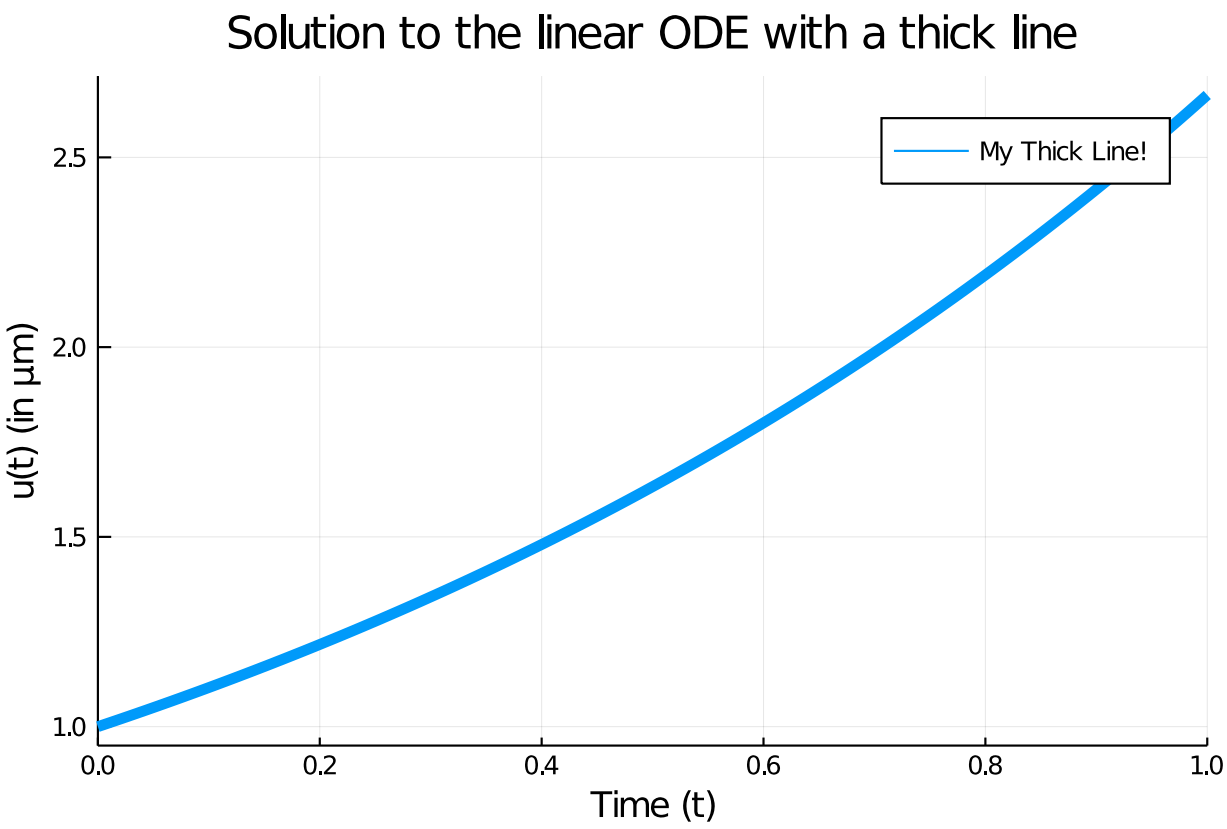
functions for analyzing the solution is the Solution Handling page. Here we will describe some of the basics. You can plot the solution using the plot recipe provided by Plots.jl:

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



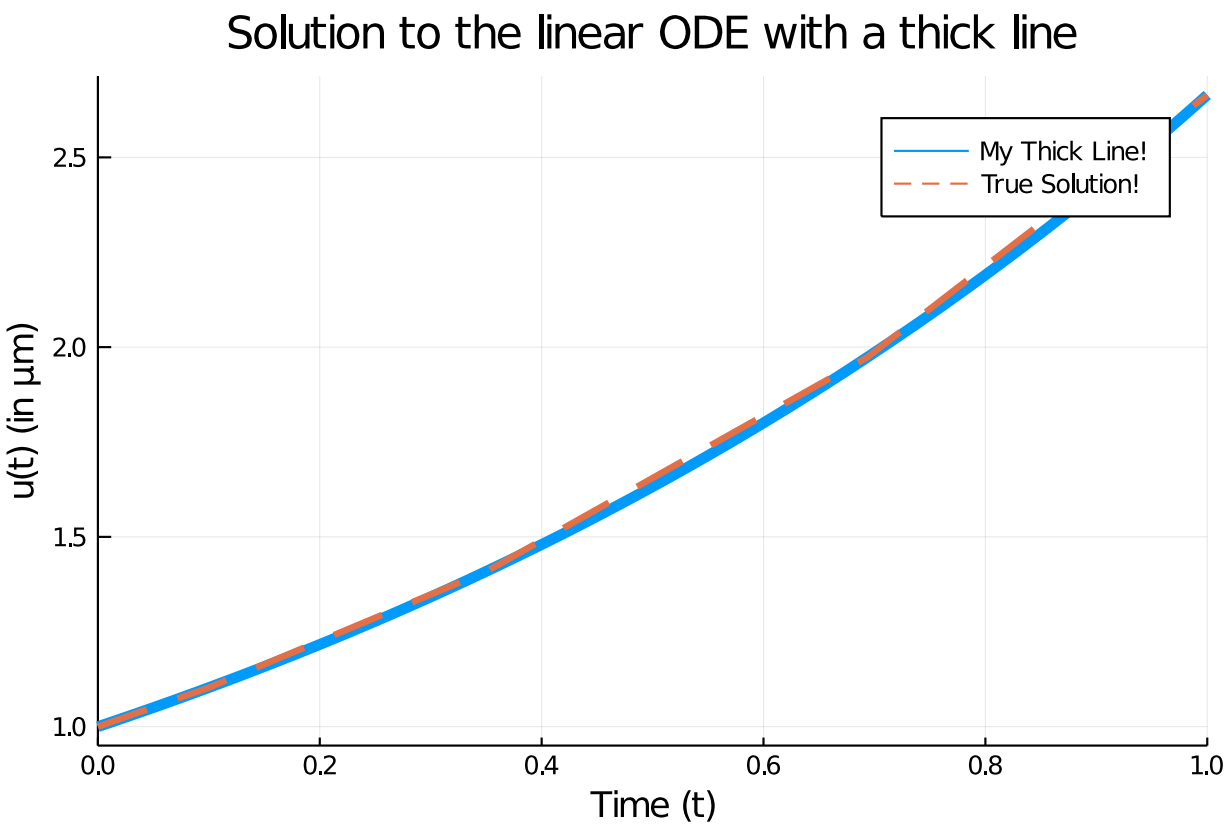
From the picture we see that the solution is an exponential curve, which matches our intuition. As a plot recipe, we can annotate the result using any of the Plots.jl attributes. For example:

```
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

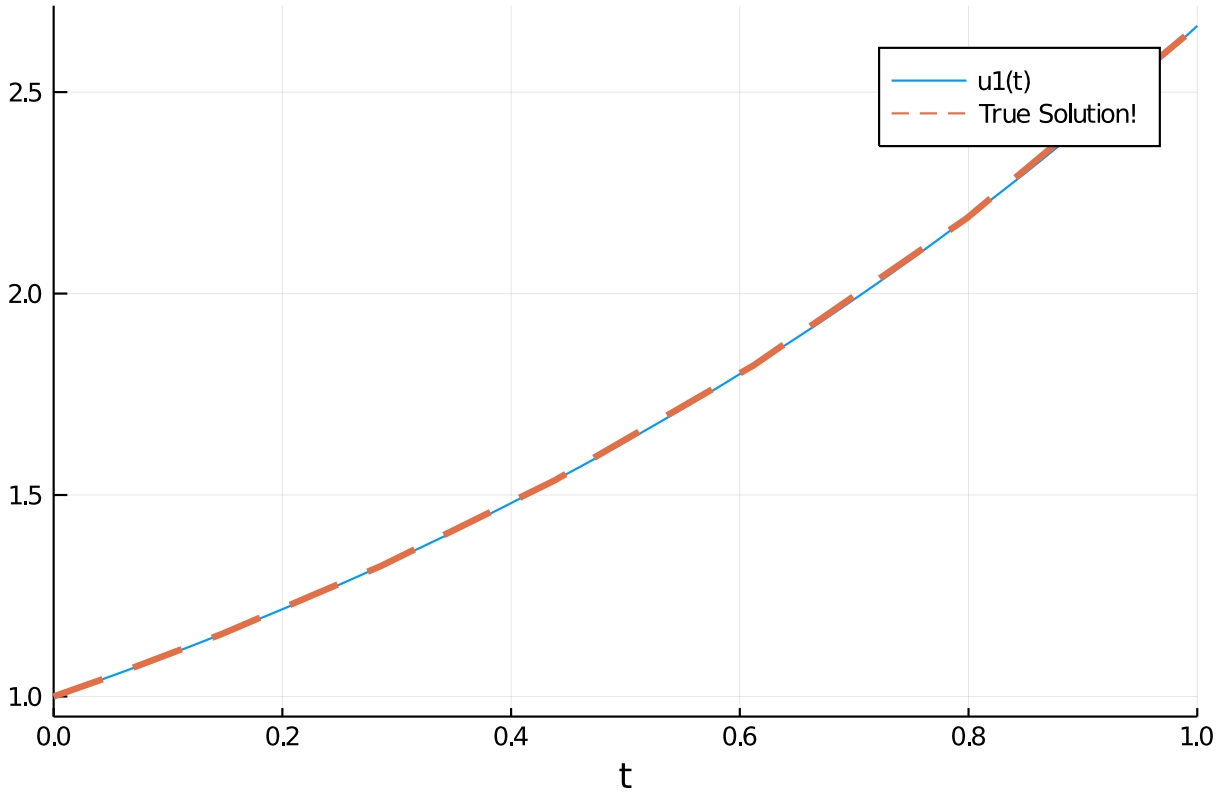
`abstol=1e-6` and `reltol=1e-3`, you can expect a global accuracy of about 1-2 digits. If we want to get around 6 digits of accuracy, we can use the commands:

```
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
```

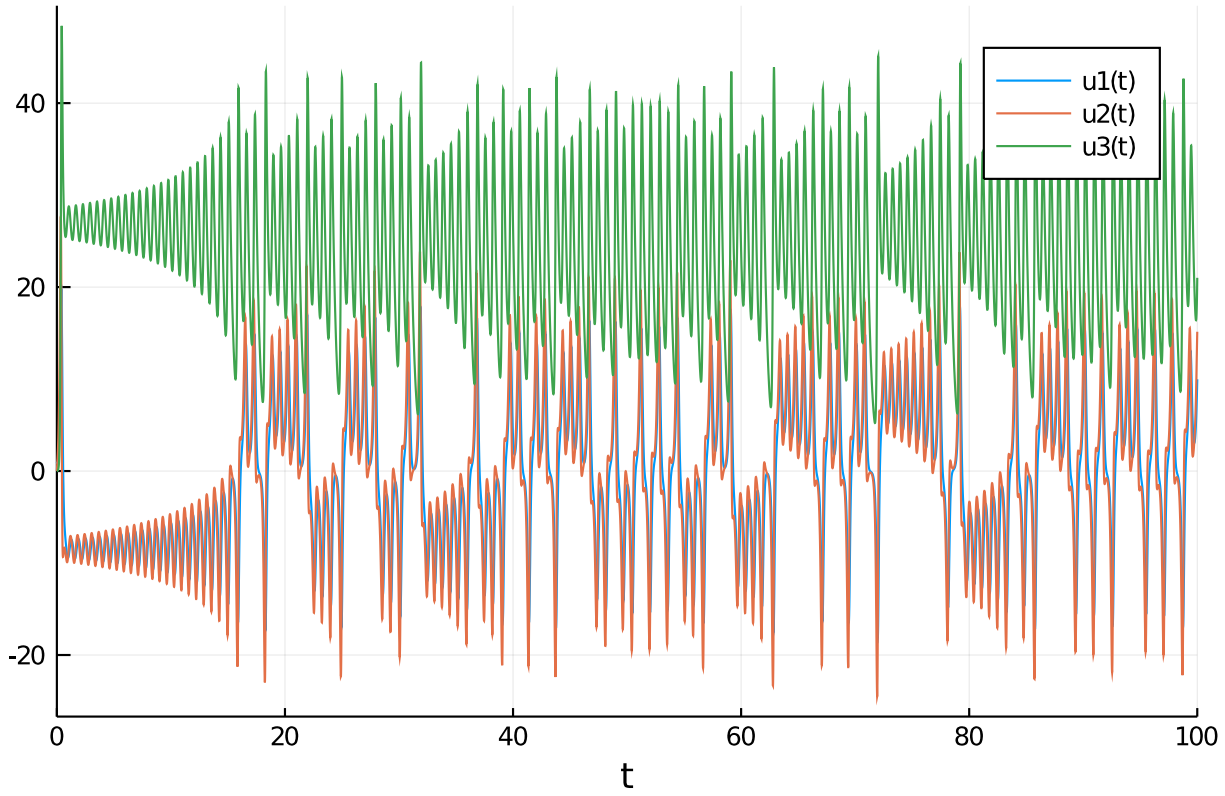
We can get a real matrix by performing a conversion:

```
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
```

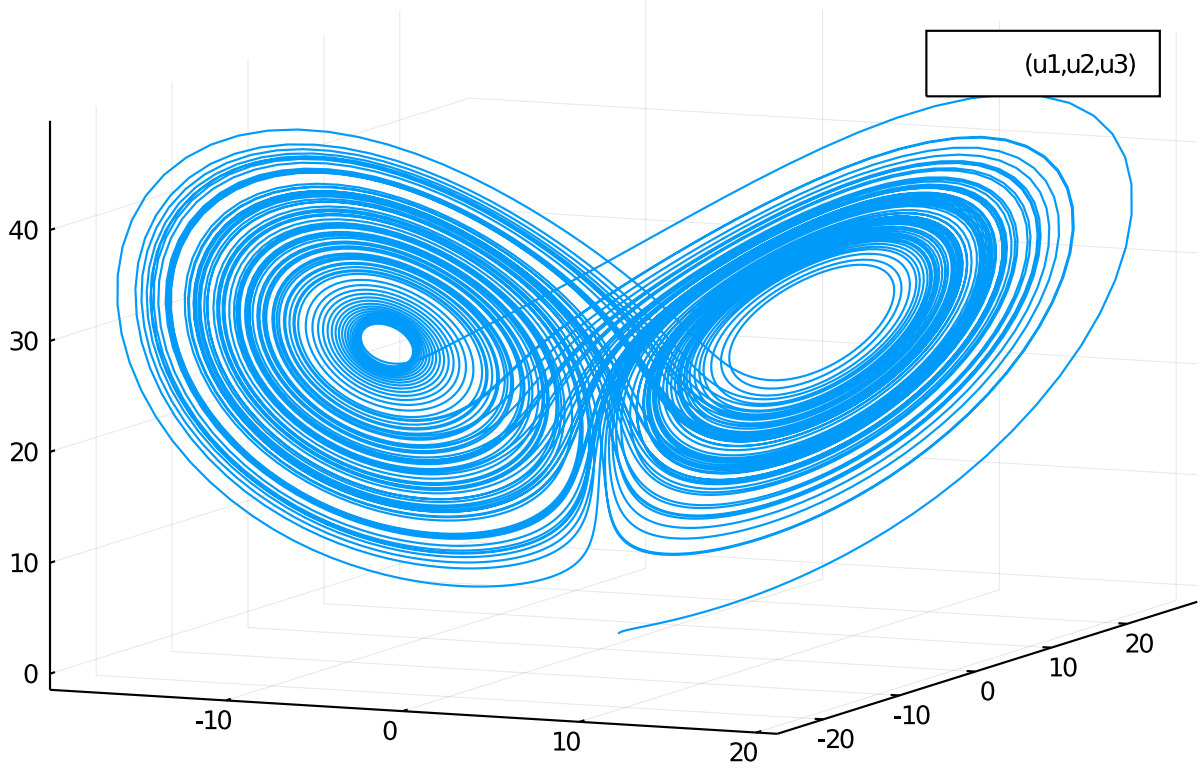
This is the same as sol, i.e. `sol[i,j] = A[i,j]`, but now it's a true matrix. Plotting will by default show the time series for each variable:

```
plot(sol)
```



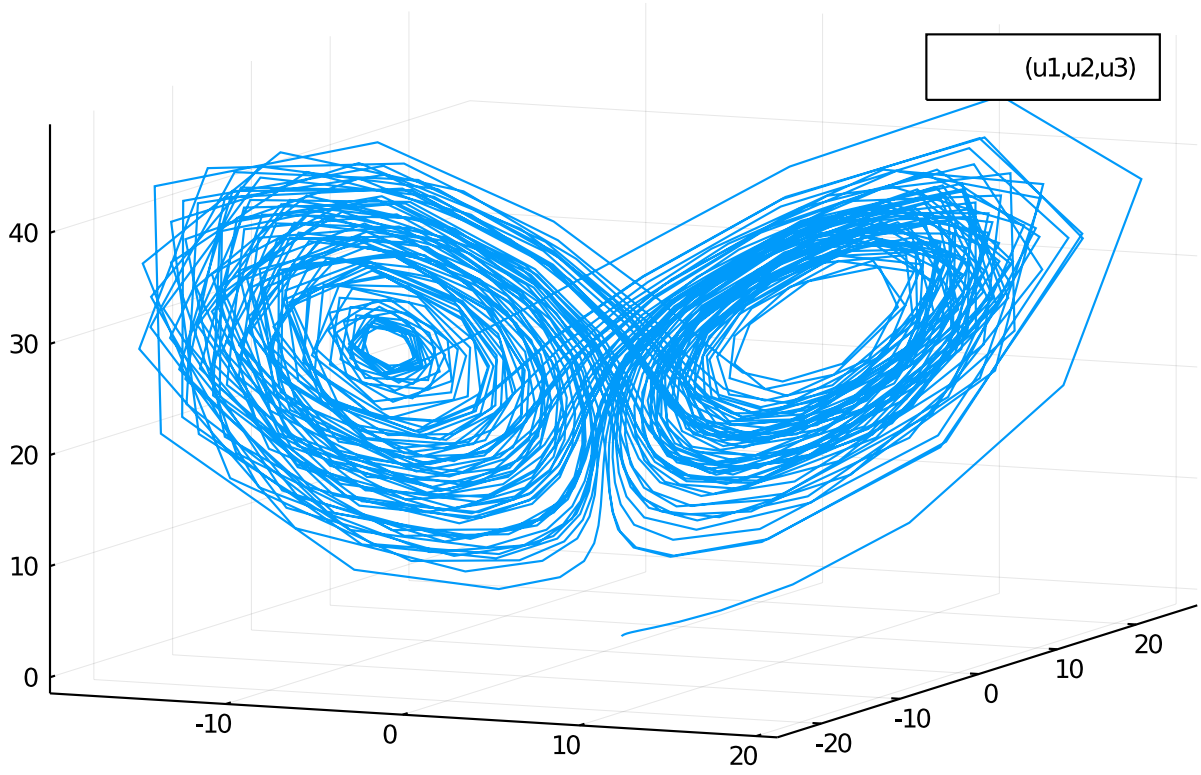
If we instead want to plot values against each other, we can use the `vars` command. Let's plot variable 1 against variable 2 against variable 3:

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



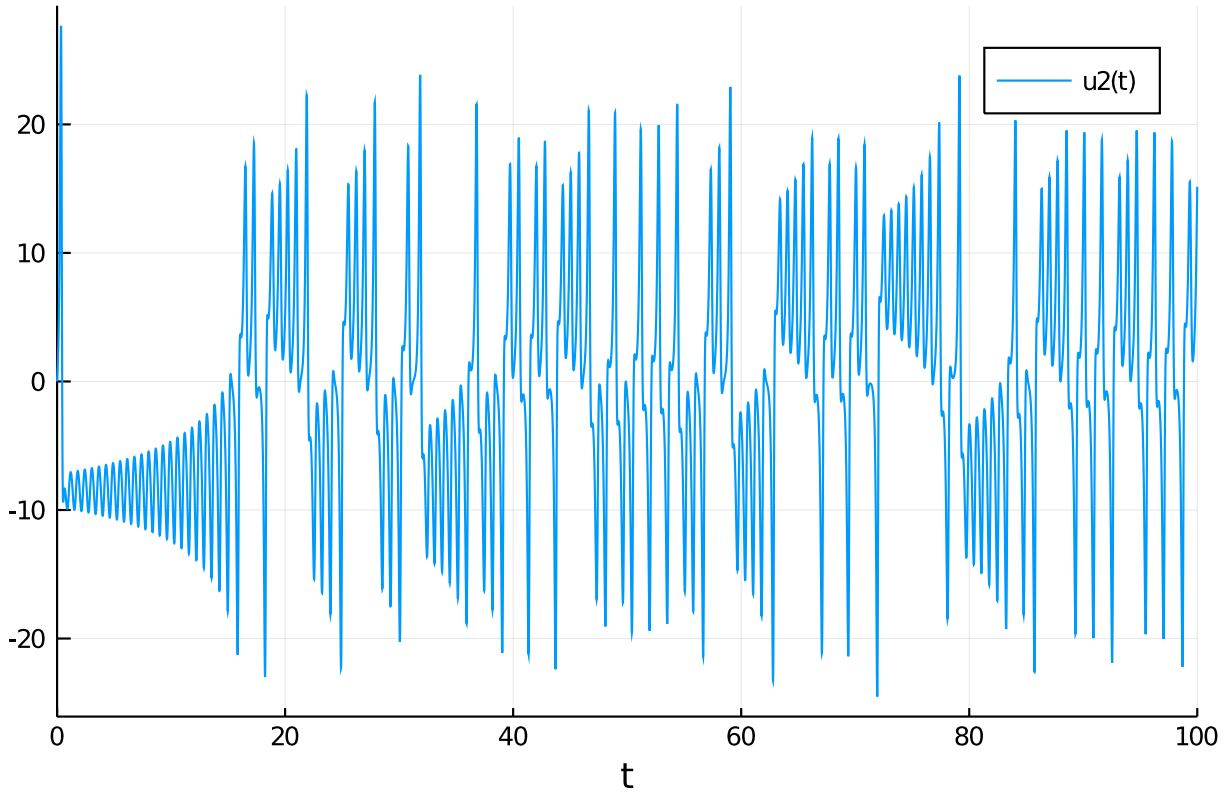
This is the classic Lorenz attractor plot, where the `x` axis is `u[1]`, the `y` axis is `u[2]`, and the `z` axis is `u[3]`. Note that the plot recipe by default uses the interpolation, but we can turn this off:

```
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
 0.03972310783882107
```

```
 0.099368494979485
 0.17320817466172284
 0.27109948837460085
 0.3893846960643187
 0.5209741318104791
 0.6666773711679637
 0.8314974106205894
 1.0
u: 10-element Array{Array{Float64,2},1}:
 [0.08881214343921484 0.28519654401856664; 0.8573406950859672 0.30510158738
9911; 0.8217928766694909 0.09324198113373083; 0.6049763120457494 0.87000506
15340407]
 [-0.0384819787829116 0.10640937107599599; 0.8486292479951969 0.21844507977
814534; 0.8431884918705893 0.09900956298249874; 0.6853870556229418 1.007843
2324096118]
 [-0.2665819844993368 -0.22525724152486018; 0.7926393536897747 0.0246369202
2072117; 0.9225961943560335 0.17741388200917235; 0.7788317294189188 1.18945
37710000708]
 [-0.5992526182731293 -0.7327038399027899; 0.6741198687091151 -0.2959374304
813885; 1.1094892661528089 0.41094991296509764; 0.8410541157010191 1.359923
3962025565]
 [-1.092564342104016 -1.5405545114569648; 0.4924759344528773 -0.78595431648
80876; 1.521702221025913 0.9912725577430487; 0.8154018720957178 1.464962788
1838487]
 [-1.6848394935425723 -2.634968610752533; 0.36488584656193934 -1.3065329220
603714; 2.2666442274244347 2.146127490219454; 0.5927618429215089 1.35467382
67890892]
 [-2.1766626567187197 -3.810863286450063; 0.5526959516238098 -1.48656461138
7934; 3.3420369989844616 4.000168331459474; 0.07128676205432372 0.851879382
1782525]
 [-2.2456100034860835 -4.694525604414771; 1.4406228461800739 -0.73659089715
52473; 4.610408222187757 6.562489146335712; -0.845270896550113 -0.249155320
11815278]
 [-1.3082717088844626 -4.513491550822032; 3.5688672303820983 1.900000615620
8429; 5.603675070387456 9.495602489872784; -2.2320305347716722 -2.207466475
574227]
 [1.1981855369970358 -2.180836507922414; 6.973063668179599 6.95794392421986
3; 5.230515658785164 11.311491958663536; -3.7901193303808407 -4.83249968786
5761]
```

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}:
 -0.266582  -0.225257
  0.792639   0.0246369
  0.922596   0.177414
  0.778832   1.18945
```

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}:
 0.0888121  0.285197
 0.857341   0.305102
 0.821793   0.093242
 0.604976   0.870005
```

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: 6-element Array{Float64,1}:
 0.0
 0.07693858714903604
 0.283363620019056
 0.5767077727693567
 0.8954871812915701
 1.0
u: 6-element Array{Array{BigFloat,2},1}:
 [0.088812143439214841222906215989496558904647827148437 5 0.2851965440185666
44496104345307685434818267822265625; 0.85734069508596721931326101184822618 9
61334228515625 0.30510158738991099980353283172007650136947631835937 5; 0.821
79287666949085711110001284396275877952575683593 75 0.09324198113373083351973
49151712842285633087158203125; 0.60497631204574942032081708020996302366256 7
138671875 0.87000506153404066900236557557946071028709411621093 75]
 [-0.17597200962325970001401436552847507445841842628926678002580438831897 56
755058755 -0.09187827204135348639641567229465896063390126461717497287074659
293838516467441702; 0.81896302499622087289641505259108165638623347898438659
44066755739560978174429 07 0.10563033779464006002557402993262676194068087751
70153056218751507396433221794 82; 0.88565307904535861699354674579227387156 16
66462748121125578114168236471603740843 1 0.13729523521143505340343655994297 5
86820831594236789661466371281175749901700849 44; 0.74784573891728881966487 98
70270660338394170761483697826268102848576087871994 8992 1.125183896383568104
53536532710402014351633480876510077977982000330602047679134 4]
 [-1.15597020245698488874624694919315533606280676854875898259102639922556 87
27281776 -1.65001477711500052093463125784963974590314225537714561641957469 6
592686046682041; 0.47196048230630555525979508477539569967656750410556795511
18689834403395180904319 -0.84714863601207091200439528027007901044303066544 1
96095077525760722722338834893 13; 1.58677853423860841683191041214726644159 30
78408763477355178827715670867528926852 1.08750529404332194679010516438978 54
12317432157346849657693277383884805573540733; 0.80252255903437962589700902 4
52204073266099324308528247203427588002747061631578 3 1.46659798364974418204
56186083510933315683497448329338139864170331883432214051 79]
 [-2.27909173298914807579421635987932988881297043333118064886776438423579 64
38012949 -4.22569943967002356789833955550917499281654217473097030860262394 8
289389881811732; 0.79378204117188419607511161180400842138295000736060955749
042924325430699818736 63 -1.34416659121232554557663717089231086090035347742 1
41579165705659050408879215117 5; 3.83578504759266463024083600564276636545593
53250034291934837641932670654152708 6 4.93547644542899782561777640484907233 4
41186454828713288085700911392023865536818 1; -0.23870137860733289487454249 80
```

```
4386343624314307905983427127220722147085564855516 3 0.501281336913918967761962168456119755412593233532923122766259430163506492769584 3]
 [-0.560357448297953118267618964740412279889317166673205796045945918728102160705292 7 -3.9310667830889126677062506452623749022749661919073466872345395475921832548521 71; 4.730607859968620742821663560623651608558792345011003504140193740238487769184059 3.533157075324052870677688546339724577566648619643337723069736319847174922337868; 5.686116618846144261116583743909399112913105157286433880151396392645637534950099 10.412680598374884162735787389955035471594437826356743658904877829322495104182 82; -2.827006011922466242003802889505119664914551528813282411305179335359968489492673 7; -3.1477703590621502074222797774682262444337911224081110418007330353427028193948 46]
 [1.19820387729643749368812800034791958583703905954848466150953922783023531836948 6; -2.180822594550738974995651995664110524187803648731813075392368611939719929382924; 6.97309463546482896014087043670312347577594147774558255426345876429415639287991 2 6.957986548701535306743377484886982310396448948328335066460391943226235645503145; 5.230516647420349973124422810115672760295696334884239391958106007424813575529439 11.311513851135571995086086714430376263130449836995280003687705247065719297161 09; -3.790132823508951598512434859445207701278862798655713026188326056598621949641908 -4.83252131562349372512086426631141490315753577063686566100392842015550744341351 8]
```

```
sol[1,3]
```

```
-1.155970202456984888746246949193155336062806768548758982591026399225568727281776
```

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: 6-element Array{BigFloat,1}:
 0.0
 0.07693858714903603814983080487442526642671633941705844002291016804951119212437842
 0.28336362001905603654914717160961285527644798799587275744013451955474752578676 28
 0.57670777276935676675054913042983329589647118411637351218287026807334651131789 09
 0.89548718129157024996477455195203020272769344669146972837918262626573189400790 65
 1.0
u: 6-element Array{Array{BigFloat,2},1}:
 [0.088812143439214841222906215989496558904647827148437 5 0.2851965440185666444961043453076854348182678222656 25; 0.85734069508596721931326101184822618961334228515625 0.30510158738991099980353283172007650136947631835937 5; 0.82179287666949085711110001284396275877952575683593 75 0.09324198113373083351973491517128422856330871582031 25; 0.60497631204574942032081708020996302366256
```

```
138671875 0.8700050615340406690023655755794607102870941162109375]
 [-0.17597200962325970749520323137497252352079674635004073254476283031960416
769861023 -0.09187827204135349732201773608933850902351967615290597094055065
400030187919385918; 0.8189630249962208709039431707501134970172969097671146155
551065214183512797630782 23 0.10563033779464005351900357932428223358743210240
058717442132296969202822591185417; 0.8856530790453586197675283095109006741 8
819986114570710418961497145007094189502 15 0.13729523521143505625569086427202
367628017317563691148679414019245320211368536 09; 0.74784573891728882252546
584692165549116391902137172020838167117152616917027150541.1251838963835681
102285928328965814803563267891966142654999993508125037175382551
 [-1.15597020245698498842868075653661526815694258807645013579211407551253754
8170002 -1.650014777115000694182844903596802926892404276309879462075980141
212104153972813; 0.47196048230630552385690345055474623425819906376578997648
458524040530641090881960 -0.84714863601207100757793065964195517276908072414 3
200539737227794671272247291381; 1.5867785342386085214867249352619477841020
905746724444729182094102435336157848351.0875052940433221023654365753849648
794244346126083649656583824665882200398604270.80252255903437960385966196 2
845967762104433358056502986771223220996250489797821 1.466597983649744182412
595304630978175761798740776309474487414562157299222930452]
 [-2.279091732989148151663567763500803487247377902516051426228543601567256 1
39204276 -4.225699439670024038391864029417390699175319933635960020317251466
412825697613398; 0.79378204117188457034736188773682136230887684538642453282
328278461823766431861220 -1.344166591212325264375369501636090737736155032321
468378810790756013017067370568; 3.835785047592665250683262564501825829571501
600883805457177012583473076854927706 4.935476445428999041954350233900612541
248269198265407696343193354425405677488947; -0.23870137860733331612854131 9
839510800237474807943258417546372318221060012251548 0.501281336913918473977
449706818065217676777180751975638149451657011175228020893 4]
 [-0.560357448297951386000247299065178288226867340128106749864129161992147 2
913964769 -3.931066783088911160911752335280643209351268867385920803296520141
6140227848392637; 4.730607859968623234135890455632751576829342644770190430 8
374114441344765413130990 3.533157075324056482488194785675556404236068577625 9
854924847257704914323015423330; 5.686116618846144186390872778952739065852494
941036497756219732746119627460838709 10.412680598374885767625364499078856 30
219288216744734624114730398107861750851537; -2.827006011922467437939888336 9
145458331678733601148349542052335253591685732506740 -3.147770359062152164912
101096354576950497186253010116583418631670331475146553505]
 [1.198203877296437493688154543431504889081387778987255689187702147290606799
260885 -2.180822594550738974995526149858061198551817266599002690679888869 0
22231529717559; 6.973094635464828960140742936771239175140758697345127961963
864754447522734503861 6.957986548701535306743301854821343261369260619569906
344840991786353871495502412; 5.230516647420349973124226244472413245402712 2 8
974880004853498023690032799101403711.311513851135571995085764117611639321 7
244300742440253732515358792987337232327 5; -3.790132823508951598512353264267
652434759043022606254833142089218082051965681744 -4.832521315623493725120778
124687987500170087540796177953472198126086714845818162]
```

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: 10-element Array{Float64,1}:
 0.0
 0.04438031062146118
 0.11785072300365426
 0.20597743299973356
 0.3100082234651565
 0.43612443413315105
 0.5741047484766288
 0.7169027208336972
 0.8785311587397756
 1.0
u: 10-element Array{StaticArrays.SArray{Tuple{4,2},Float64,2,8},1}:
 [0.21103097539005167 0.22597377980580236; 0.6031581091615956 0.34683768005
30608; 0.4752549669090169 0.7616691531076119; 0.29741073427704157 0.8934420
975961352]
 [0.14495853720549684 0.010986904464440533; 0.6188196756566442 0.3418785196
6017485; 0.4581882647990857 0.7839373625914294; 0.3685740901681188 1.090220
744459473]
 [-0.003723393072314979 -0.4579506312015146; 0.5931770281425873 0.214019922
68841583; 0.4671207609095637 0.9369534546929223; 0.46603453300615794 1.3662
99582767819]
 [-0.23763871884898005 -1.1850948036766034; 0.49816615239100315 -0.08220522
614764922; 0.552822140401531 1.352967781285802; 0.5406156438234239 1.588246
3671726177]
 [-0.5677549765090889 -2.2129554472552644; 0.3428995215972567 -0.5051255276
027085; 0.7766665121815393 2.2256265977948093; 0.5542225370113982 1.6478057
205420145]
 [-0.9876295646614429 -3.5439329507262167; 0.19051862359091257 -0.849066148
0899777; 1.2340861327293187 3.8721204802340234; 0.44006613780128756 1.34626
21390065306]
 [-1.3590927818776375 -4.780124938604513; 0.22603851553984916 -0.5575597852
891078; 1.9307446396507069 6.321003466716155; 0.12810075844440394 0.4530775
6676987754]
 [-1.4787574608759335 -5.30536923534868; 0.6691634936432309 1.0371107905296
402; 2.7411526339717334 9.210727087558285; -0.40779091648073296 -1.15019935
75990692]
 [-1.0403684622304108 -4.175211239629608; 1.8485566962878555 5.017333404512
595; 3.465117399411872 12.01267044980029; -1.2386495547917922 -3.7402035418
23556]
 [-0.15098325576865368 -1.577128474793121; 3.2409176552086736 9.68322049522
9644; 3.583029336659146 12.886927036451508; -1.9514180207506389 -6.05472383
8737258]
```

```
sol[3]
```

```
4×2 StaticArrays.SArray{Tuple{4,2},Float64,2,8} with indices SOneTo(4)×SOne
To(2):
 -0.00372339  -0.457951
```

```
  0.593177     0.21402
  0.467121     0.936953
  0.466035     1.3663
```

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 DiffEqTutorials.jl repository, found at: https://github.com/JuliaDiffEq/DiffEq

To locally run this tutorial, do the following commands:

```
using DiffEqTutorials
DiffEqTutorials.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) Xeon(R) CPU @ 2.30GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-8.0.1 (ORCJIT, haswell)
Environment:
  JULIA_CUDA_MEMORY_LIMIT = 536870912
  JULIA_DEPOT_PATH = /builds/JuliaGPU/DiffEqTutorials.jl/.julia
  JULIA_PROJECT = @.
  JULIA_NUM_THREADS = 4
```

Package Information:

```
Status `/builds/JuliaGPU/DiffEqTutorials.jl/Project.toml`
[2169fc97-5a83-5252-b627-83903c6c433c] AlgebraicMultigrid 0.3.0
[7e558dbc-694d-5a72-987c-6f4ebed21442] ArbNumerics 1.0.5
```

```
[6e4b80f9-dd63-53aa-95a3-0cdb28fa8baf] BenchmarkTools 0.5.0
[be33ccc6-a3ff-5ff2-a52e-74243cff1e17] CUDAnative 3.1.0
[159f3aea-2a34-519c-b102-8c37f9878175] Cairo 1.0.3
[3a865a2d-5b23-5a0f-bc46-62713ec82fae] CuArrays 2.2.1
[55939f99-70c6-5e9b-8bb0-5071ed7d61fd] DecFP 0.4.10
[abce61dc-4473-55a0-ba07-351d65e31d42] Decimals 0.4.1
[ebbdde9d-f333-5424-9be2-dbf1e9acfb5e] DiffEqBayes 2.15.0
[eb300fae-53e8-50a0-950c-e21f52c2b7e0] DiffEqBiological 4.3.0
[459566f4-90b8-5000-8ac3-15dfb0a30def] DiffEqCallbacks 2.13.3
[f3b72e0c-5b89-59e1-b016-84e28bfd966d] DiffEqDevTools 2.21.0
[77a26b50-5914-5dd7-bc55-306e6241c503] DiffEqNoiseProcess 4.2.0
[9fdde737-9c7f-55bf-ade8-46b3f136cc48] DiffEqOperators 4.10.0
[1130ab10-4a5a-5621-a13d-e4788d82bd4c] DiffEqParamEstim 1.14.1
[055956cb-9e8b-5191-98cc-73ae4a59e68a] DiffEqPhysics 3.2.0
[0c46a032-eb83-5123-abaf-570d42b7fbaa] DifferentialEquations 6.14.0
[31c24e10-a181-5473-b8eb-7969acd0382f] Distributions 0.23.4
[497a8b3b-efae-58df-a0af-a86822472b78] DoubleFloats 1.1.12
[587475ba-b771-5e3f-ad9e-33799f191a9c] Flux 0.10.4
[f6369f11-7733-5829-9624-2563aa707210] ForwardDiff 0.10.10
[7073ff75-c697-5162-941a-fcdaad2a7d2a] IJulia 1.21.2
[23fbe1c1-3f47-55db-b15f-69d7ec21a316] Latexify 0.13.5
[c7f686f2-ff18-58e9-bc7b-31028e88f75d] MCMCChains 3.0.12
[eff96d63-e80a-5855-80a2-b1b0885c5ab7] Measurements 2.2.1
[961ee093-0014-501f-94e3-6117800e7a78] ModelingToolkit 3.1.1
[2774e3e8-f4cf-5e23-947b-6d7e65073b56] NLsolve 4.4.0
[8faf48c0-8b73-11e9-0e63-2155955bfa4d] NeuralNetDiffEq 1.5.0
[429524aa-4258-5aef-a3af-852621145aeb] Optim 0.21.0
[1dea7af3-3e70-54e6-95c3-0bf5283fa5ed] OrdinaryDiffEq 5.41.0
[65888b18-ceab-5e60-b2b9-181511a3b968] ParameterizedFunctions 5.3.0
[91a5bcdd-55d7-5caf-9e0b-520d859cae80] Plots 1.4.0
[d330b81b-6aea-500a-939a-2ce795aea3ee] PyPlot 2.9.0
[731186ca-8d62-57ce-b412-fbd966d074cd] RecursiveArrayTools 2.4.4
[47a9eef4-7e08-11e9-0b38-333d64bd3804] SparseDiffTools 1.8.0
[684fba80-ace3-11e9-3d08-3bc7ed6f96df] SparsityDetection 0.3.2
[90137ffa-7385-5640-81b9-e52037218182] StaticArrays 0.12.3
[f3b207a7-027a-5e70-b257-86293d7955fd] StatsPlots 0.14.6
[789caeaf-c7a9-5a7d-9973-96adeb23e2a0] StochasticDiffEq 6.23.1
[c3572dad-4567-51f8-b174-8c6c989267f4] Sundials 4.2.3
[1986cc42-f94f-5a68-af5c-568840ba703d] Unitful 1.2.1
[44d3d7a6-8a23-5bf8-98c5-b353f8df5ec9] Weave 0.10.2
[b77e0a4c-d291-57a0-90e8-8db25a27a240] InteractiveUtils
[37e2e46d-f89d-539d-b4ee-838fcccc9c8e] LinearAlgebra
[44cfe95a-1eb2-52ea-b672-e2afdf69b78f] Pkg
```