An Intro to DifferentialEquations.jl

Chris Rackauckas

June 14, 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)
```

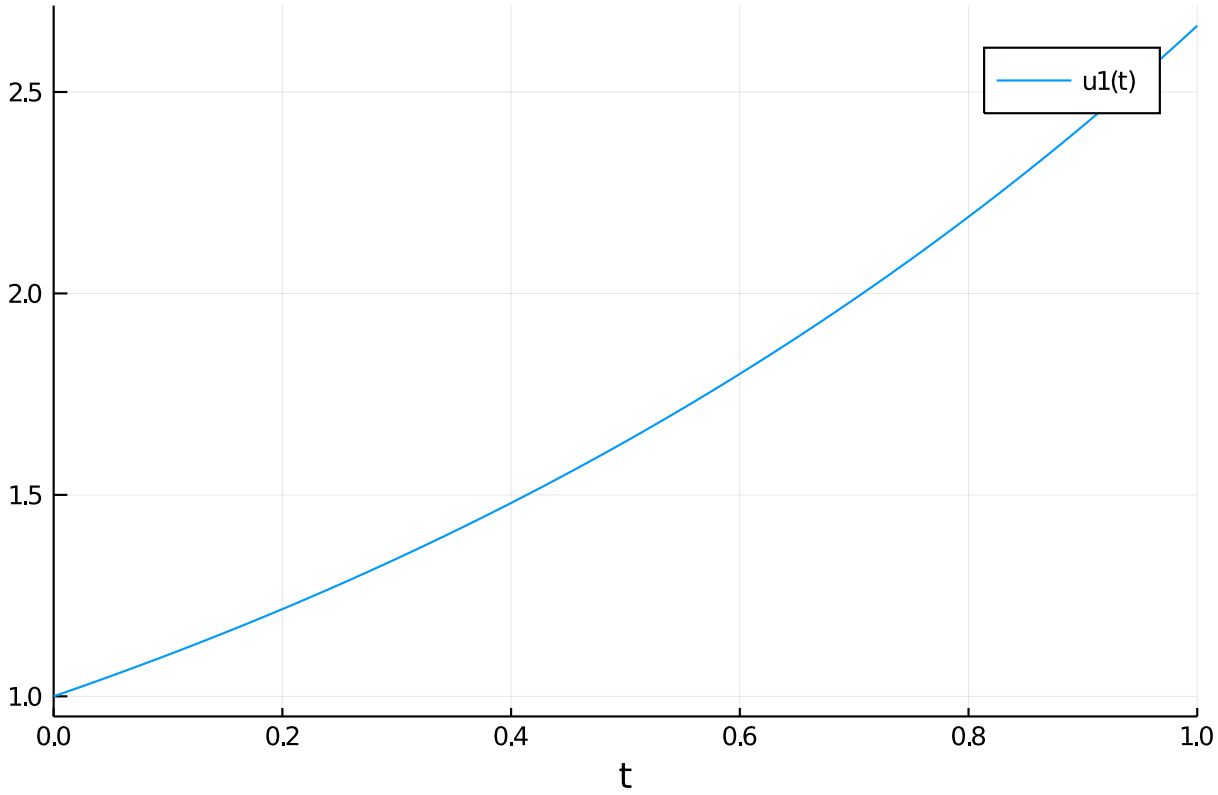
```
retcode: Success
Interpolation: Automatic order switching interpolation
t: 5-element Array{Float64,1}:
 0.0
 0.10042494449239292
 0.35218603951893646
 0.6934436028208104
 1.0
u: 5-element Array{Float64,1}:
 1.0
 1.1034222047865465
 1.4121908848175448
 1.9730384275622996
 2.664456142481451
```

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

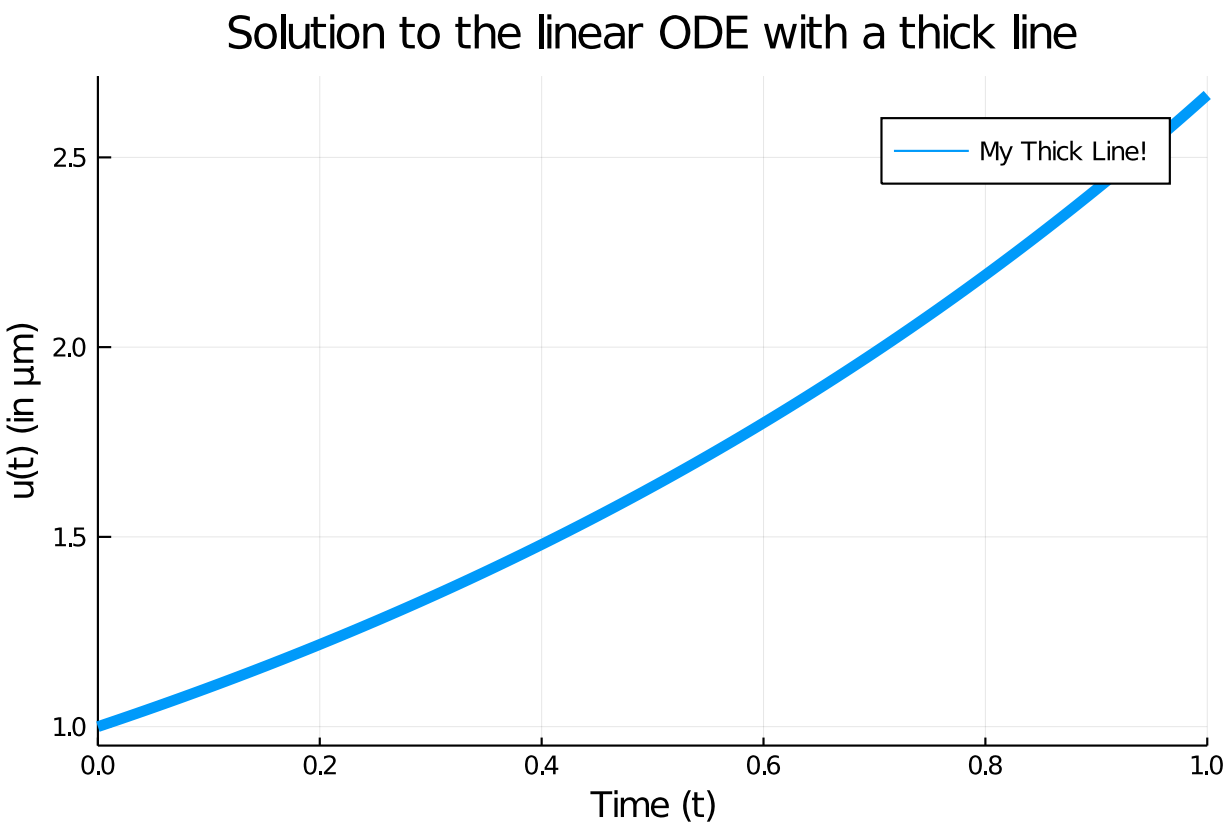
functions for analyzing the solution is the Solution Handling page. Here we will describe some of the basics. You can plot the solution using the plot recipe provided by Plots.jl:

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



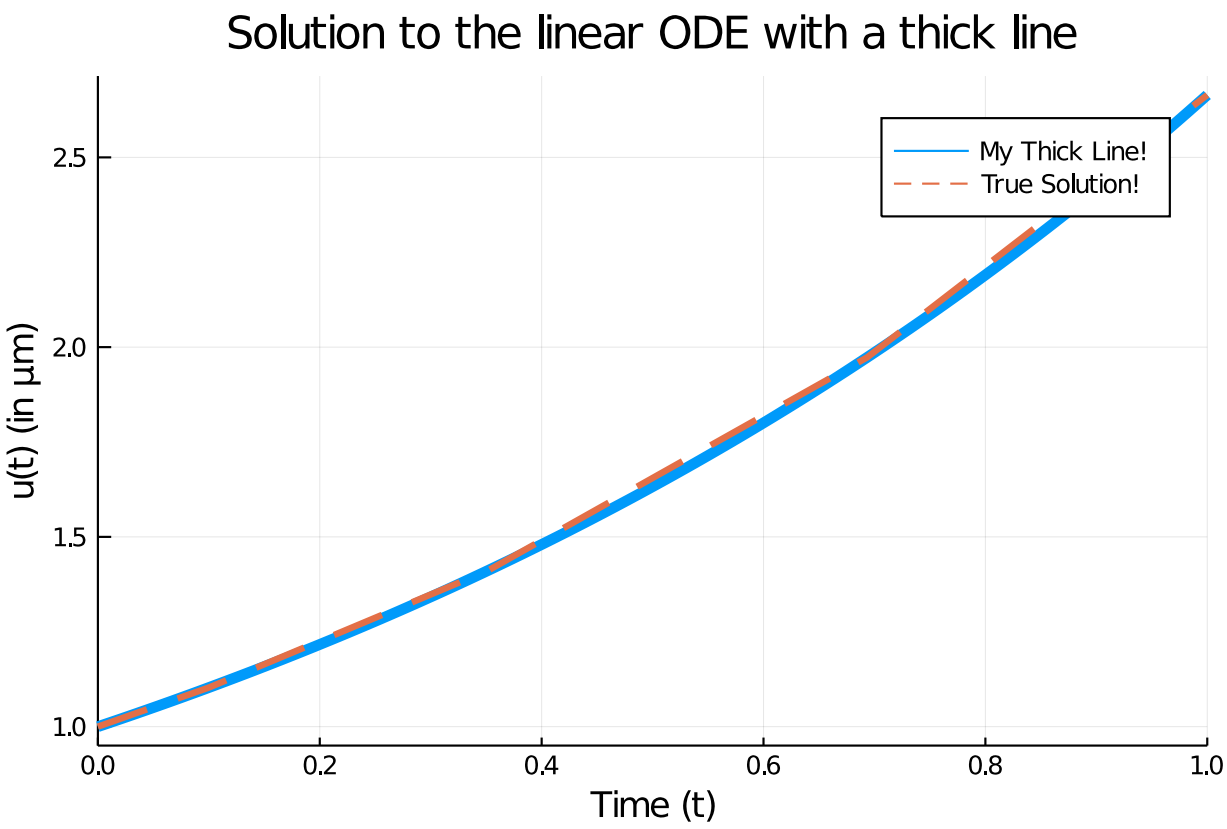
From the picture we see that the solution is an exponential curve, which matches our intuition. As a plot recipe, we can annotate the result using any of the Plots.jl attributes. For example:

```
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

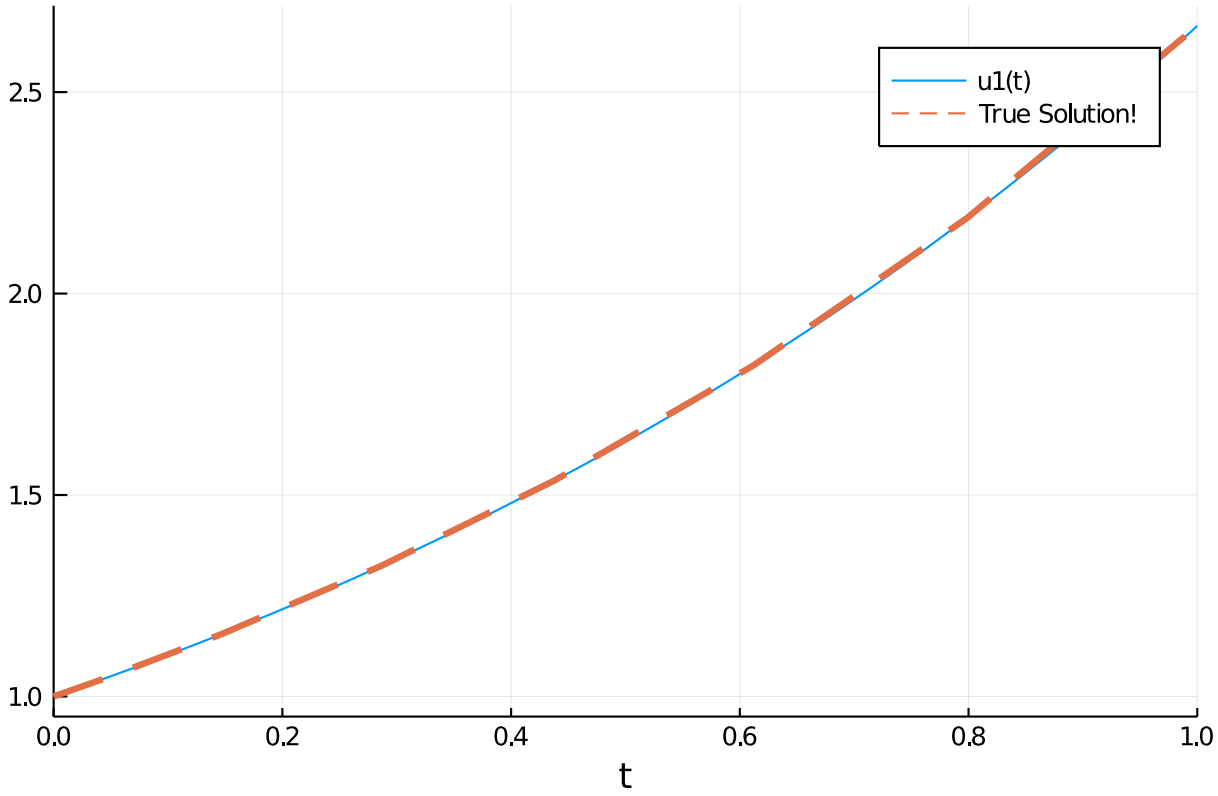
`abstol=1e-6` and `reltol=1e-3`, you can expect a global accuracy of about 1-2 digits. If we want to get around 6 digits of accuracy, we can use the commands:

```
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
```

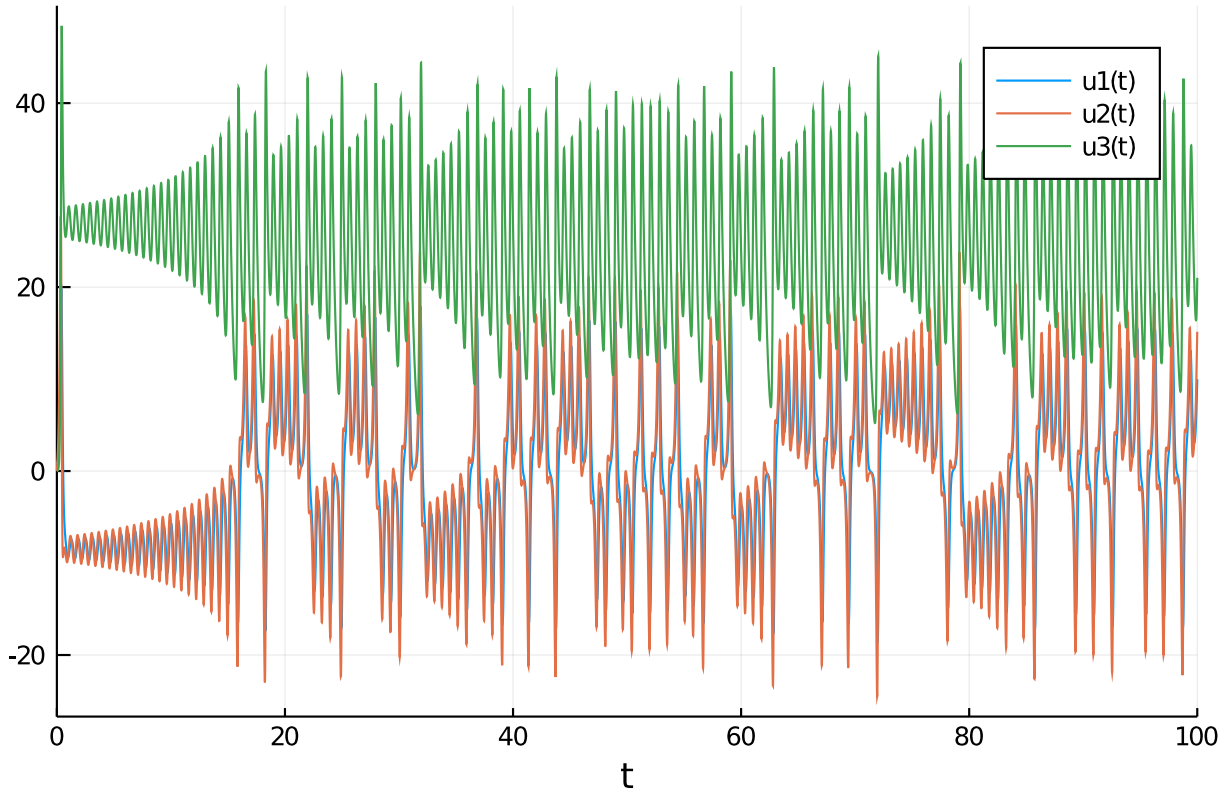
We can get a real matrix by performing a conversion:

```
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
```

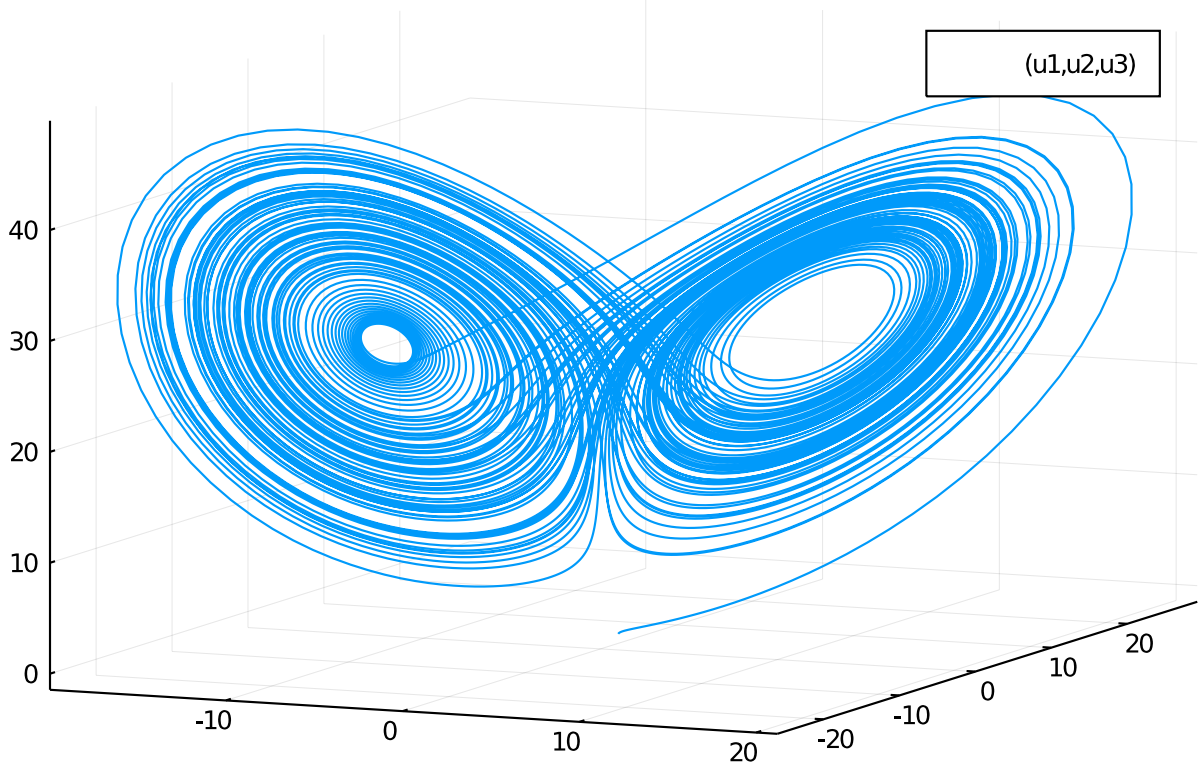
This is the same as sol, i.e. `sol[i,j] = A[i,j]`, but now it's a true matrix. Plotting will by default show the time series for each variable:

```
plot(sol)
```



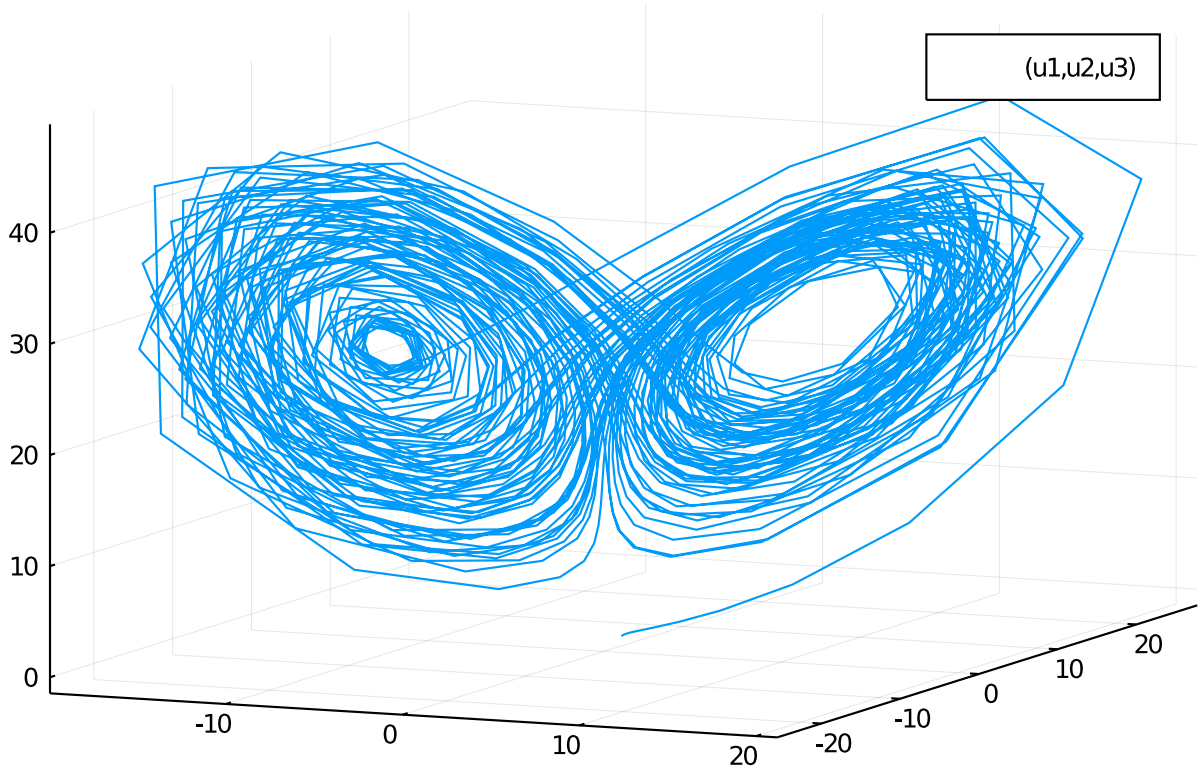
If we instead want to plot values against each other, we can use the `vars` command. Let's plot variable 1 against variable 2 against variable 3:

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



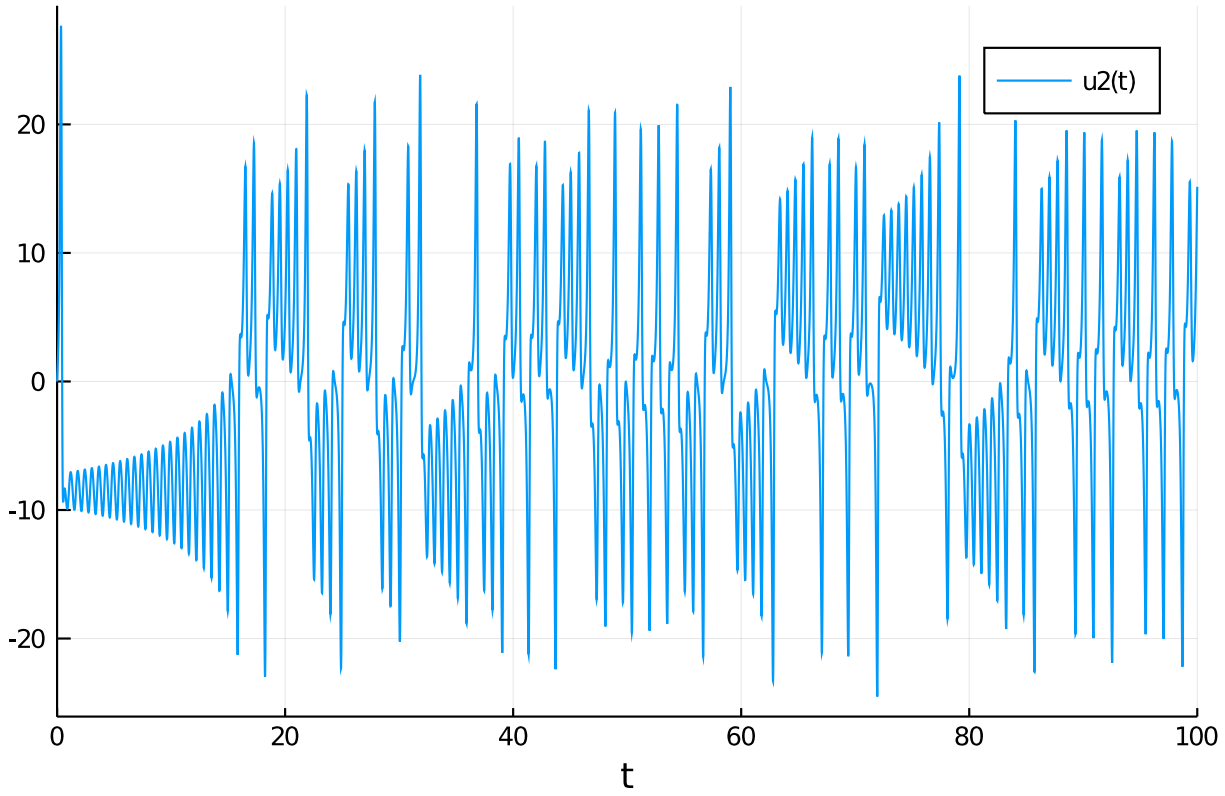
This is the classic Lorenz attractor plot, where the `x` axis is `u[1]`, the `y` axis is `u[2]`, and the `z` axis is `u[3]`. Note that the plot recipe by default uses the interpolation, but we can turn this off:

```
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
 0.05392648622307931
```

```
 0.1406132536939904
 0.23261557152239454
 0.3516952255025277
 0.4823733912189384
 0.6275835875214623
 0.796759097145997
 0.9831278804494583
 1.0
u: 10-element Array{Array{Float64,2},1}:
 [0.20075484072637195 0.7757405105775659; 0.8967433436433823 0.967117294956
2656; 0.980525927678076 0.3303948714318077; 0.1677649362171667 0.0703911672
6426942]
 [0.15360775609339183 0.7784140739473572; 1.006607155225334 1.0559363342164
354; 0.9532188047086694 0.16992661752982702; 0.2513257214925577 0.221682566
79909618]
 [0.02951434485930901 0.6955199499599316; 1.1040695908117368 1.058141269438
6352; 0.9463791460600056 -0.05925537968655875; 0.35210816419319 0.456759024
15339353]
 [-0.15194608228725986 0.4876549356117511; 1.1272944821492565 0.88450309624
10229; 1.0028989801660435 -0.22809030934046706; 0.40313529490278777 0.68230
23464595378]
 [-0.4212652951959783 0.042442049171868834; 1.1006611985833963 0.4418198174
0930915; 1.1860511674708485 -0.2661342597463349; 0.36664639193607607 0.9110
584902252213]
 [-0.6807811668698157 -0.640006799759824; 1.1112545192193721 -0.20922446784
223064; 1.5152771620987866 0.011554127654654689; 0.17753773665045988 1.0386
239131781112]
 [-0.7873540985438721 -1.5415177748390512; 1.3333350295151487 -0.9108926163
871542; 1.9541352687341735 0.7899997988154843; -0.21787518187205232 0.96855
29498617972]
 [-0.44798557154448604 -2.551289533555641; 2.0583729428090534 -1.2935544052
94493; 2.3165339317463496 2.318525328714581; -0.8812434028949278 0.52098350
74470793]
 [0.7756638100834994 -3.1458661250822226; 3.506235710565667 -0.577580643490
3984; 2.015026048712961 4.5048288006668695; -1.7092420295582131 -0.51004388
26147548]
 [0.9372824658462704 -3.1508581381092617; 3.6644387978403876 -0.43325426865
127825; 1.9278367149014712 4.70773601023575; -1.77977550335718 -0.632061999
471221]
```

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}:
 0.0295143   0.69552
 1.10407     1.05814
 0.946379   -0.0592554
 0.352108    0.456759
```

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}:
 0.200755  0.775741
 0.896743  0.967117
 0.980526  0.330395
 0.167765  0.0703912
```

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
 0.06710711556243842
 0.28768651229642395
 0.5831183483910013
 0.9086442367054589
 1.0
u: 6-element Array{Array{BigFloat,2},1}:
 [0.200754840726371952541740029118955135345458984375 0.77574051057756587823
63279897253960371017456054687 5; 0.896743343643382262442287355952430516481 39
95361328125 0.967117294956265638461445632856339216232299804687 5; 0.98052592
76780760036018591563333757221698760986328125 0.330394871431807679584835568 6
757713556289672851562 5; 0.167764936217166704679470967676024883985519409179 6
875 0.070391167264269416392608036403544247150421142578125]
 [0.138365924368441596721896994972261860708423236843306172289327509969934 47
98364662 0.772822981373959335948499860269920147688469066386760523240702538 7
114413087713372; 1.027334846499251080989895224517272807055660226787009235 00
942316718623174780836 7 1.067315654296136358961445604920127393853065389044 83
617163737698394449204733306 5; 0.948949376610011876757273408893688571298456 6
614271602238143560309675037421889613 0.132186526396001034573670718935402142
467645619665664953011110537650682028902259 7; 0.269510745944762107500675837 4
707234482880801259011223892499500204476265664586959 0.258287779695249366152
063984477324131023644177588761308262541783057045381379360 1]
 [-0.275053974865865550518911557973824525679493409739538749452662408882393 2
96471318 0.305387352130809245167342621013926095377547348897870558341008787
1259543987491882; 1.117859528904977601503887483262135452951698040303434545 1
970045679583470339314 27 0.705705578333208158768453309513651157549629249251 2
233377458583573901065372161238; 1.072088519966089551170890548745728391698 35
433951498673454373322278876723838035 3 -0.275351595399439136142082545279086 9
341199263673577029674401735034612353360721932; 0.401604117291500746050047 82
891696555961927192274047067926985279457166605576873 85 0.798844931885021095 5
537769851220064772643829564988609695366221070432540511816958]
 [-0.784770369336626846072703750845952210844585021127122205602007444681124 3
828621879 -1.258014185596069155422799201940449648637955849904952814239714 92
1473916534734309; 1.232196209894328972848664801851851053833927586852435678 4
034840111790625863325 57 -0.715417723683433010756988245264023278979171257383
939366532767983361476815005265 6; 1.820361265740220721377456392283967257735 5
149741218388039198907163133246472660 63 0.496809103339840727241948298670192 9
212781875193685168509869508953140844315696385; -0.077090514832188654054108 7
```

```
52387633681425834695177520850040219185985448272092078 4 1.017598766563264849
06009889853920605476383080995206446525103405783492845275846 7]
 [0.16539373471855043235269990616314167267561648071405024983323074603349333
38170379 -3.012724179940403476541315694134726655752203181759187696181593844
746228164546646; 2.85428469479678773960257067108243022564615065209268018511
391411515134319557385 7 -1.046203962469471340624565080250533298540331866448 0
381048040247557637346689729 62; 2.266313493148240471976257897193309155759051
090079254238063271139205050076195 45 3.602662976039042705969459145408495664 7
129331750995586392187425099610798620762 26; -1.382252899276956473312409462 44
030552478369659050483452322765450728354115693376 8 -0.027734004878170336773 5
746729083512458041402690049747777211901473831323689408227 2]
 [0.937295655901917723868229379078492499271600035516140995757619504481381 89
43330874 -3.150867381144996182004750209236630684158204794066582530662474788
18717481015792; 3.664460098368130854025896462345709559380591309280388130206
767836067883054579 66 -0.433245272557746429190031899107521904504141721007496
676539217048634824003826020 3; 1.927835330537225261331128417855674130551357 8
411618446057500661920847434067868 83 4.707761356490605968924452447222042551 5
701025983177815499564962936468622326280 21; -1.779784611902473211214738996 01
580509619236568727402309528796416466093273072137 5 -0.632070945224491626356 8
030768271175565166180888262826559269062146386574813749507]
```

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

```
-0.275053974865865550518911557973824525679493409739538749452662408882393229
6471318
```

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
 0.067107115562438418377697552621102545570540441832212517454262105079696189
49759242
 0.287686512296423947506690006844863599463955345609664823575147787622855867
5940776
 0.583118348391001237223137562711281971719124172681811699301835255327879129
7493794
 0.908644236705458802225160205670866220622479461008606364552372147659270654
6806487
 1.0
u: 6-element Array{Array{BigFloat,2},1}:
 [0.200754840726371952541740029118955135345458984375 0.77574051057756587823
63279897253960371017456054687 5; 0.896743343643382262442287355952430516481 39
95361328125 0.967117294956265638461445632856339216232299804687 5; 0.98052592
76780760036018591563333757221698760986328125 0.330394871431807679584835568 6
757713556289672851562 5; 0.167764936217166704679470967676024883985519409179 6
```

```
875 0.07039116726426941639260803640354424715042114257812
5]
 [0.13836592436844160367489127851113916678416026527029526355140858673807329
41241007 0.77282298137395933893061189530226834836417840845698045074303396
99882695271173377; 1.02733484649925107244485235774929366819580047260049886985
730824985174856888097 9 1.06731565429613635487572454395633941827347436491014
667093054899973832682654348 6; 0.94894937661001187839004148679811145987812
55891668659714322902078631061966671051 0.13218652639600105086382973621519477
3597633837587054200799683361507958066211651 9; 0.26951074594476209977484537
16888514238992475296087808533740391732251125678500 78 0.25828777969524935023
15476023150140912742533800966009341819344611792211311349785]
 [-0.27505397486586548993133785592912962847059867938029959024114240094578
33832601109 0.30538735213080934306064255716735657422465078056081648158952365
12164143390820049; 1.11785952890497760820182588267412527002461612103096104
39426796280145849972548 17 0.70570557833320825663430287237162557956568961581
42665088553103942636145109180665; 1.07208851996608951131602444482730261458
56895424560589775723214354532236755888 66 -0.27535159539943912492447325713
56700836252559279369040403857800732318800569262257; 0.40160411729150075300
29160431962070178186166895516082449090655329231546506637422 0.79884493188502
10435038454144001704570034874956649562373828384830294620391600754]
 [-0.78477036933662681173402093974664391560803438280995433204686424277674
10707830353 -1.25801418559606860970797553528012902390378859586637511382936
3718158726036875872; 1.23219620989432880866637418270907969712745968389211
40589090083464098379173706 4 -0.71541772368343260944460854219422830736017299
93430135844492780505542680546674509; 1.82036126574022045806644721185702786
89817182027842210457248863295975256726170 88 0.49680910333984020701147424811
54064223454487856297059218880620575848238278690338; -0.07709051483218839789
75346961896983419626857169418906400014528958165246003738687 9 1.01759876656
32649189573478875682849602502447506548690535034231146142996087791 42]
 [0.16539373471854949063273759595861178454360989988605843843163921745451003
08033735 -3.01272417994040309407922351104904600298146512610985295145465826
4587588652834985; 2.85428469479678665305014713090025803185832044655639189955
158101877172436288295 -1.04620396246947194416427344306949458712220635510277
701520880809262607280656 2005; 2.26631349314824074395691132160238109725711
863146140706733127516552469580612637 1 3.60266297603904110599796492298944732
720550421712042975818111630507585212970040 1; -1.38225289927695587504688940
8399586579079679433483263411507493121529576673868884 -0.02773400487816955841
129652313872193642082440535032074159318737809387866042673811]
 [0.93729565590191745085831100267504022254411801334522701725378637208323945
6528 7039 -3.15086738114499618226816975225236601488689103331673433060440462
5881618066161608; 3.66446009836813059115409234016169713168496659926410267812
937072428002687246 1144 -0.43324527255774667872046006094867287374953998243
671116191586256262173689935147 98; 1.92783533053722541479089770109675652377
96418922631213425820106201001091156002637 4 4.70776135649060563665151347493
33367693423295171889471158728746403321484693858 22; -1.77978461190247309669
18079198576791533317110816450425170191411016790697984543 1 -0.63207094522
4491421838834066358528836589395323010878674009910269011525937558268]
```

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
 0.04077144418220069
 0.09797709764732387
 0.16362990401992172
 0.240726389022935
 0.34029638631605874
 0.4648937736100185
 0.6137018349351122
 0.7787988400313253
 0.9490115787984593
 1.0
u: 11-element Array{StaticArrays.SArray{Tuple{4,2},Float64,2,8},1}:
 [0.7896883235841079 0.8758256609227859; 0.5833739882166069 0.4868377261122
385; 0.07468513004306354 0.304186835584229; 0.724374236741433 0.59605244090
00487]
 [0.6505800318720438 0.7640022115816141; 0.5582144852360046 0.5326217850716
795; -0.00967569863595065 0.19884722701311416; 0.9286526781551622 0.8296379
226578134]
 [0.3756769761169915 0.5186400567498405; 0.42683310691591175 0.485894516724
80516; -0.06792049082502699 0.10663271837846786; 1.1978921924988644 1.14237
37364024698]
 [-0.05231673049658053 0.1104917349828276; 0.15382395646550784 0.2866393663
772453; -0.025106198510803625 0.10700753436916774; 1.470779308514076 1.4673
195009726654]
 [-0.6985712890715667 -0.5335724659877886; -0.29663654829678743 -0.11097755
773369194; 0.21054507733987743 0.293624432100265; 1.7227396507593182 1.7810
91564811748]
 [-1.727477698531247 -1.5956935028197958; -0.9912204935568418 -0.7911935221
111379; 0.8692593093672187 0.9015308561704477; 1.9002065375394221 2.0349052
297360117]
 [-3.210006709696324 -3.177500958116653; -1.802443211652017 -1.660864210594
3447; 2.329487132886209 2.3402479798652185; 1.8108858088146558 2.0260953866
84216]
 [-4.955060440262974 -5.115636236851844; -2.175327004357237 -2.179290667651
5864; 4.998834783140636 5.0810744816139035; 1.1215648897015242 1.3931318749
124386]
 [-6.20395187591461 -6.637629387710871; -0.9995599022573427 -1.177873925535
9167; 8.809311137921831 9.13084901201097; -0.5296334165138357 -0.2733579036
3937984]
 [-5.716435661146319 -6.4192274339595565; 2.8630730835783207 2.622460845339
1243; 12.69573871328679 13.442158625471304; -3.2410596132734986 -3.11614375
62839547]
 [-5.041512489387573 -5.805494137957469; 4.636997407095338 4.42355030945225
; 13.607572524377087 14.50854858009318; -4.225855877356378 -4.1666621846576
45]
```

```
sol[3]
```

```
4×2 StaticArrays.SArray{Tuple{4,2},Float64,2,8} with indices SOneTo(4)×SOne
To(2):
  0.375677   0.51864
  0.426833   0.485895
 -0.0679205  0.106633
  1.19789    1.14237
```

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
[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
[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
[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
[c3572dad-4567-51f8-b174-8c6c989267f4] Sundials 4.2.2
[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
```