An Intro to DifferentialEquations.jl

Chris Rackauckas

September 13, 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`.

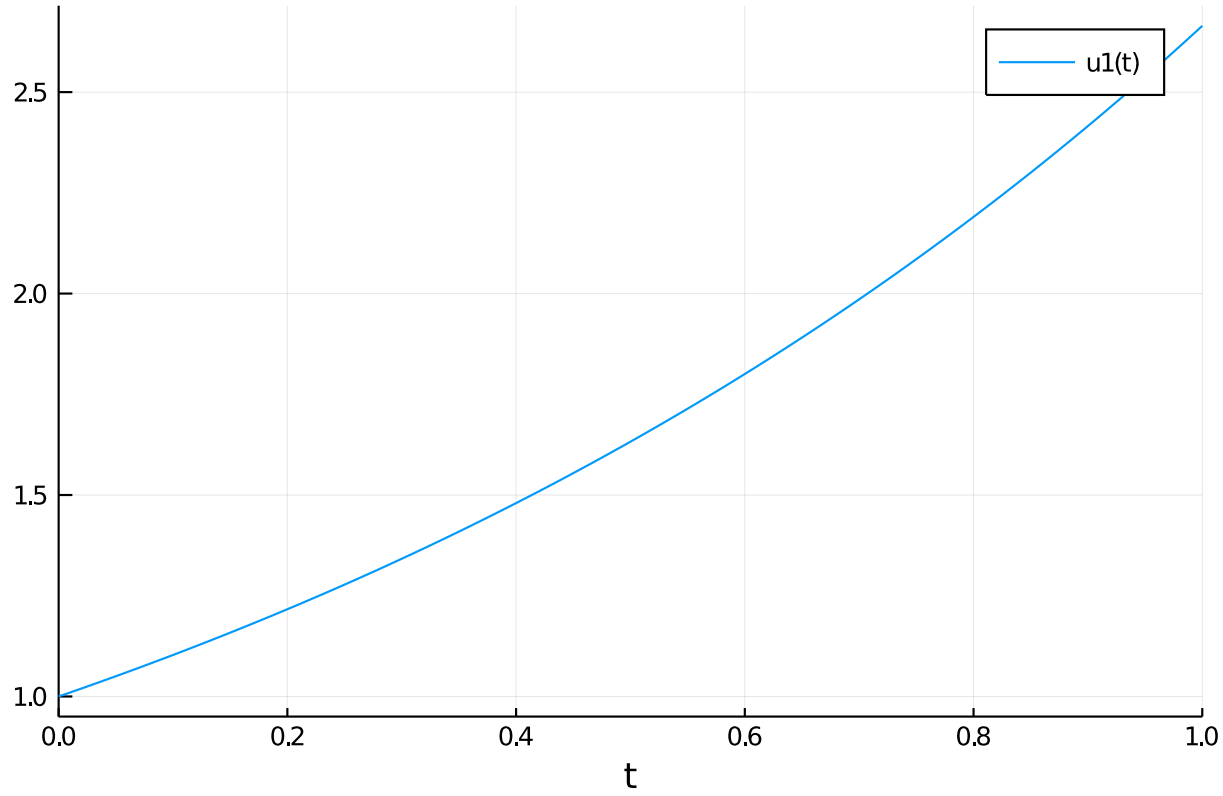
```
sol = solve(prob)
```

```
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!

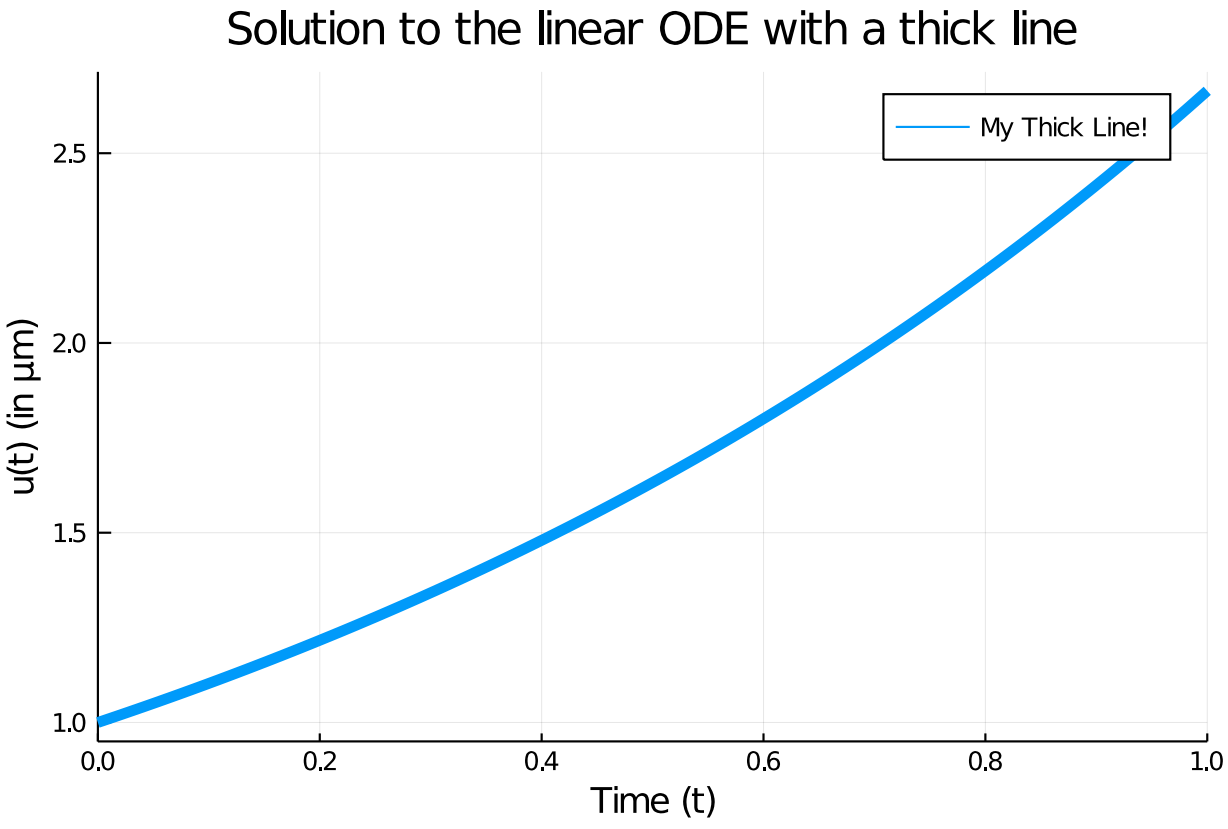
Analyzing the Solution Of course, the solution type is not interesting in and of itself. We want to understand the solution! The documentation page which explains in detail the functions for analyzing the solution is the Solution Handling page. Here we will describe some of the basics. You can plot the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

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



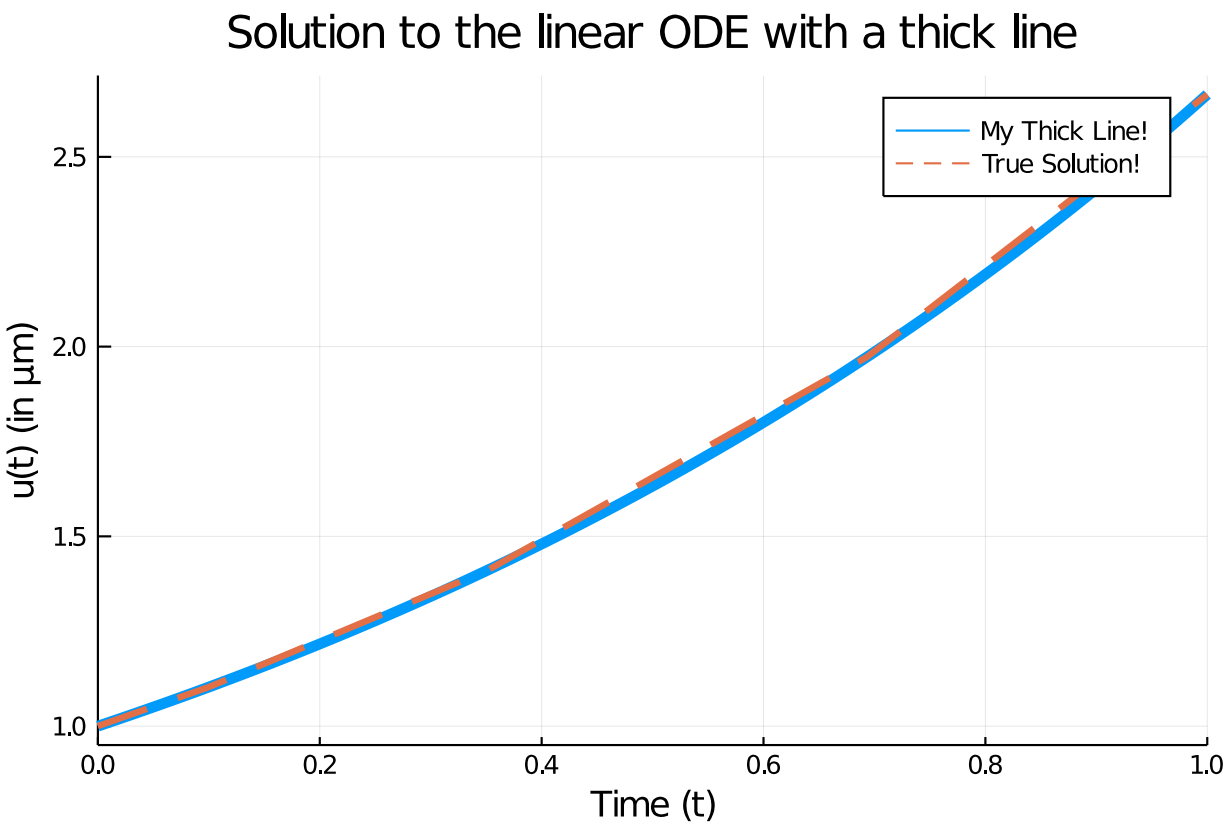
From the picture we see that the solution is an exponential curve, which matches our intuition. As a plot recipe, we can annotate the result using any of the Plots.jl attributes. For example:

```
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 . 5 5 4 2 6 1 0 4 8 0 5 5 3 1 2

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:

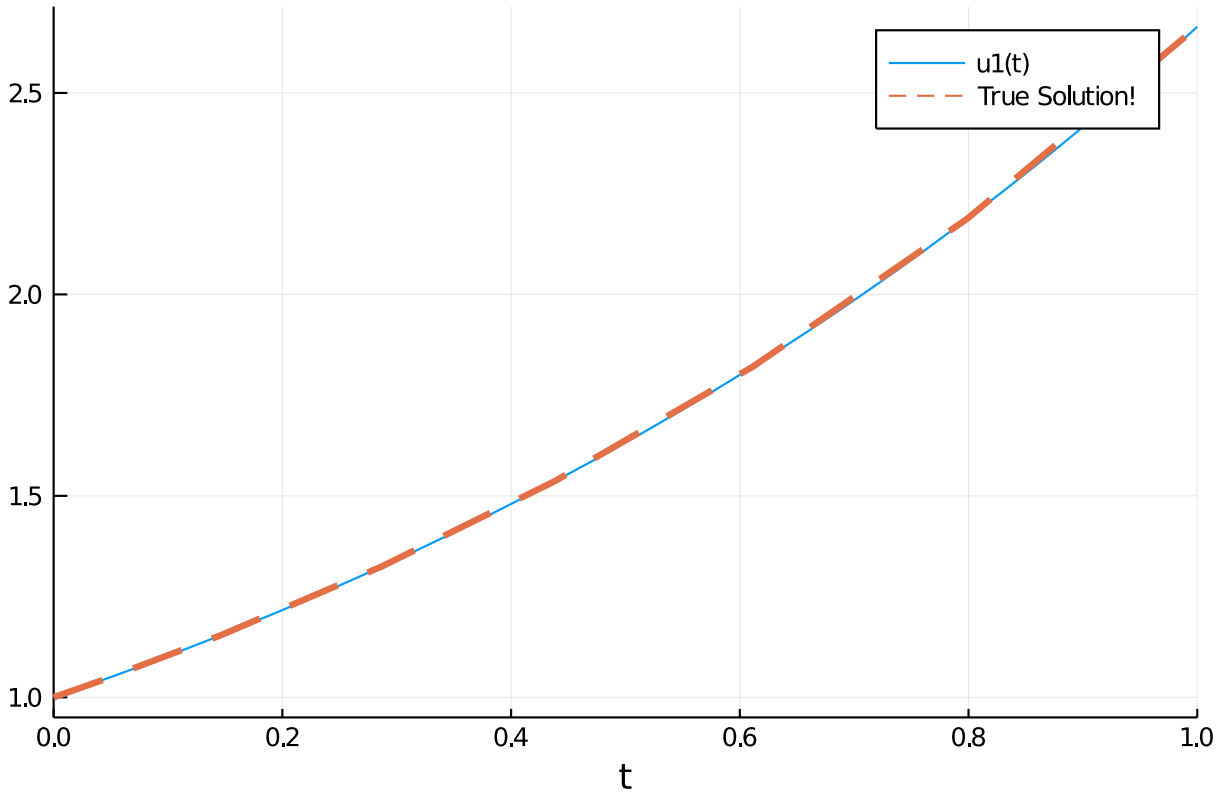
```
sol = solve(prob,abstol=1e-8,reltol=1e-8)
```

```
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


⋮@*(99.3940307091529799.4700114749437599.5437965690901599.61465155834999.6909382314810199.787330232337
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 . 0 5 2 3 2 6 1 5 3 0 2 9 0 4 2

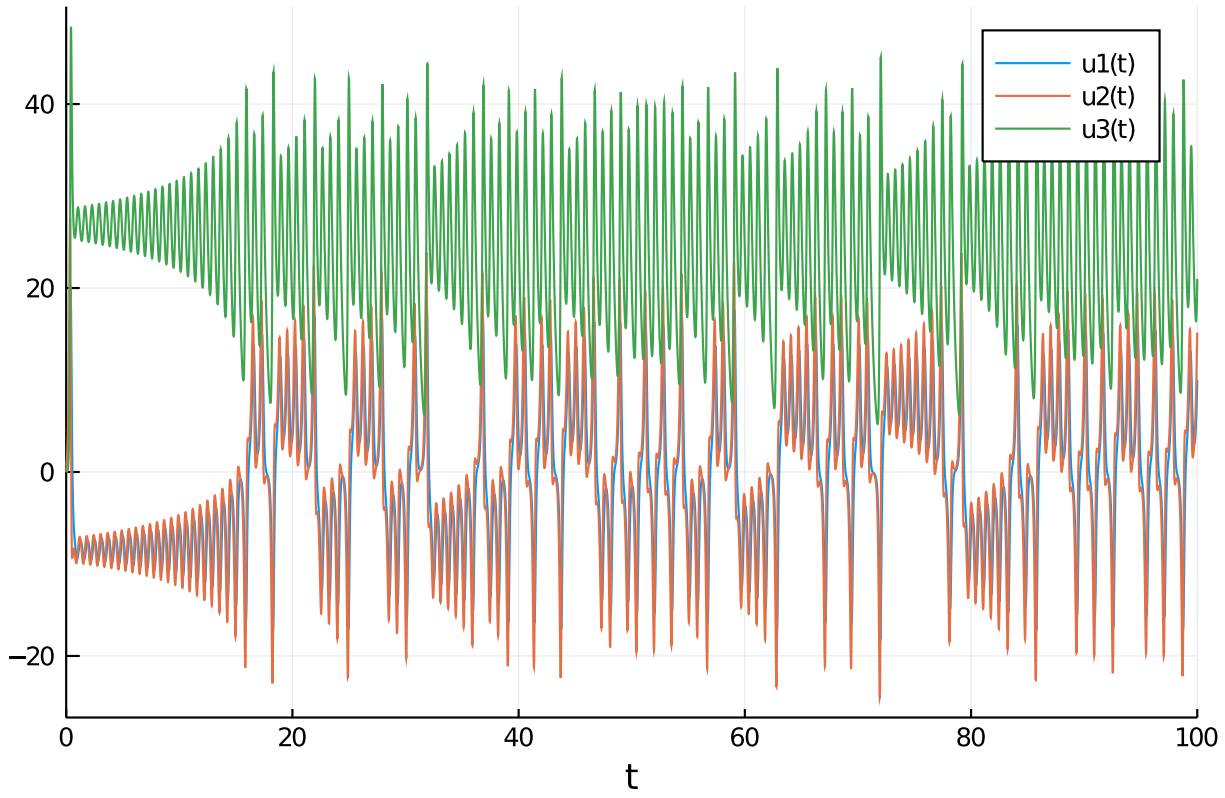
We can get a real matrix by performing a conversion:

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

```
3×@*(1294 Array(*@{Float64,2}:
 1.0  0.999643     0.996105    0.969359     ...@*( 4.52712 8.04367 9.975380.0 0.000998805
0.0109654 0.0897706 6.89588 12.7116 15.14390.0 1.78143e-8 2.14696e-6 0.000143802 16.5854
18.1254 21.0064
```

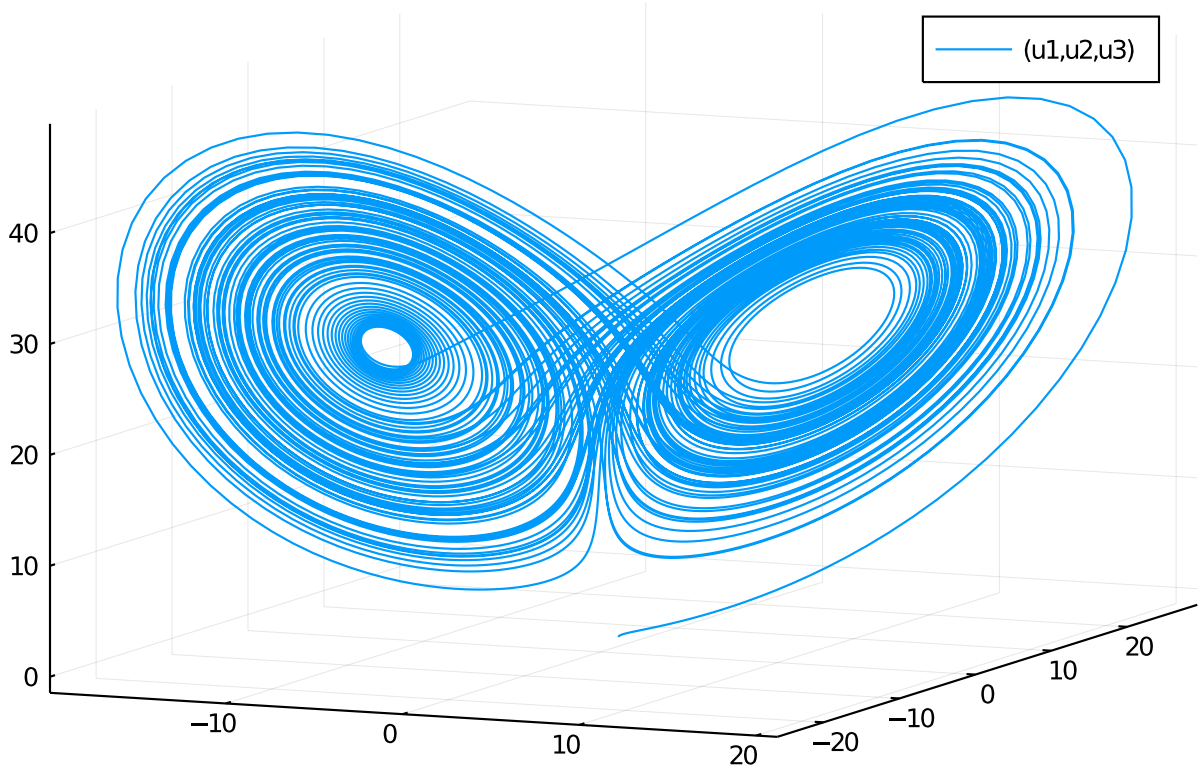
This is the same as sol, i.e. `sol[i,j] = A[i,j]`, but now it's a true matrix. Plotting will by default show the time series for each variable:

```
plot(sol)
```



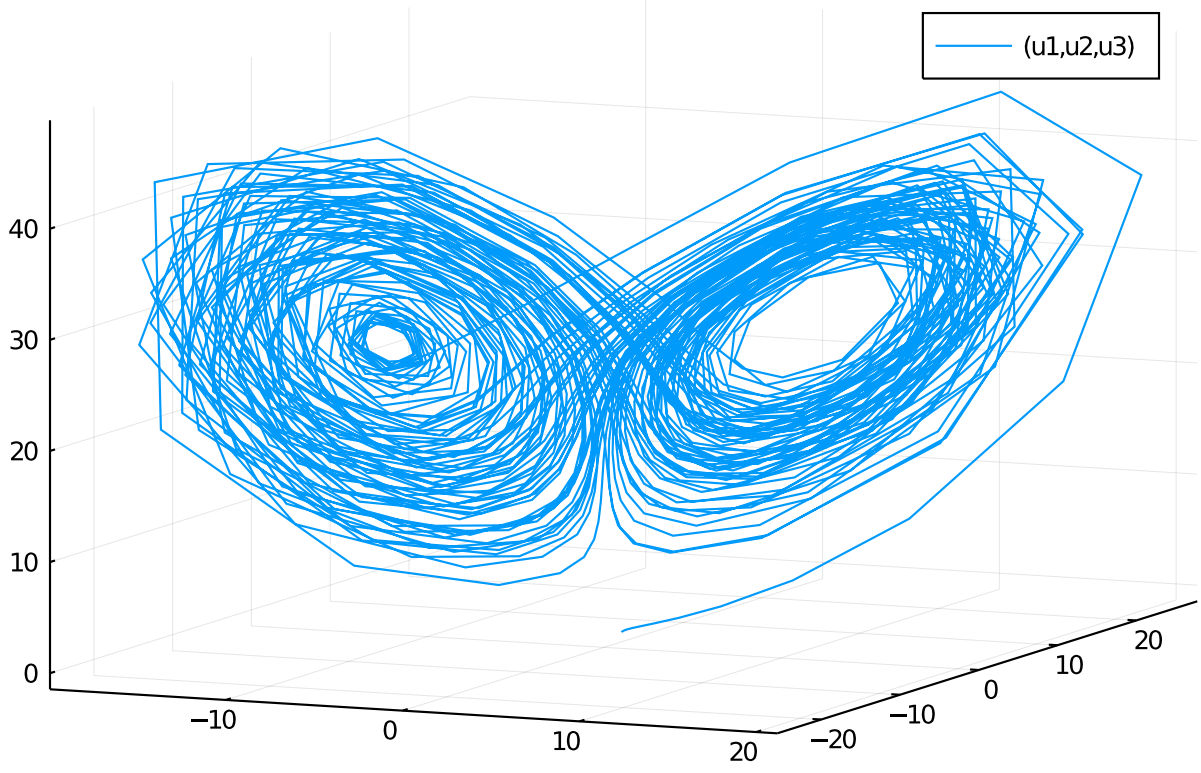
If we instead want to plot values against each other, we can use the `vars` command. Let's plot variable 1 against variable 2 against variable 3:

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



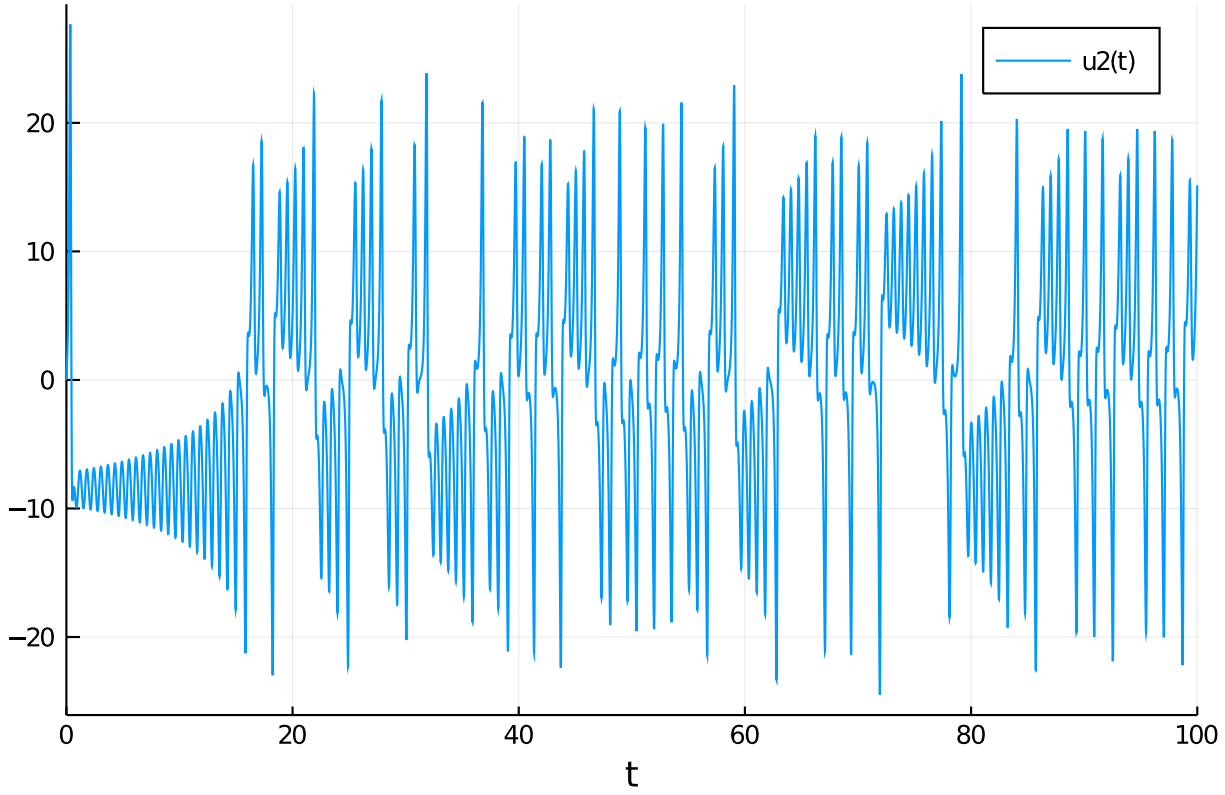
This is the classic Lorenz attractor plot, where the `x` axis is `u[1]`, the `y` axis is `u[2]`, and the `z` axis is `u[3]`. Note that the plot recipe by default uses the interpolation, but we can turn this off:

```
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: 11-element Array{Float64,1}:
 0.0
 0.0378804853136824
 0.09822979107923621
 0.17164916169410643
 0.26696228936116195
 0.3756626403200669
 0.48679521575542184
 0.6235274643467611
 0.7784864259417906
 0.939666577559981
 1.0
u: 11-element Array{Array{Float64,2},1}:
 [0.6107426767545865 0.20450179835791737; 0.9719930975314253 0.319576889858
14614; 0.4864414730703177 0.07577706275657703; 0.02899231113673495 0.603935
0807702066]
 [0.6208818206442575 0.08752175289132093; 1.047676831635972 0.2577817209687
7955; 0.39558935269469003 0.077908489214442; 0.11053170656595851 0.68913765
61025813]
 [0.6056010176824679 -0.1401512734970753; 1.1125271414428957 0.117186890798
1449; 0.25726939056803333 0.12853506181806693; 0.23628219333285186 0.806968
3107562626]
 [0.5345817241725646 -0.47958716768074816; 1.1029365655916277 -0.1062917915
149994; 0.11108883944082443 0.28138882838434215; 0.37810707051410214 0.9128
409817855674]
 [0.35893028706650737 -1.0025869403977175; 0.9610136034112924 -0.4353033718
5806; -0.01835272155050849 0.6520442580731975; 0.5347074395748432 0.9716898
274832753]
 [0.0546588423246282 -1.6637222649438865; 0.6602200461256789 -0.76926162668
97222; -0.04641085646533329 1.3337566336822007; 0.6602028273008376 0.902196
8373556304]
 [-0.34602173129195496 -2.3260767004378144; 0.2666705000678275 -0.928815558
6079767; 0.09266590615386605 2.3110393163001586; 0.7116170861914148 0.64825
93647391249]
 [-0.9027100174961151 -2.9458359577621116; -0.2066110587128041 -0.644258257
2639444; 0.5271748560738192 3.818791600153346; 0.6412531012408232 0.0431329
169338428]
 [-1.4890750914796622 -3.057857313386042; -0.5071649623352912 0.64084188157
46995; 1.3552354348497233 5.649646193347419; 0.34940867159267774 -1.0540331
651213428]
 [-1.8336820932915607 -2.038430639883; -0.2746778555074971 3.34065236004437
5; 2.4621650821609875 7.078446010371258; -0.2172199667903263 -2.59849798840
232]
 [-1.837992657909004 -1.2582683653031514; 0.007609609620422231 4.7221368083
04371; 2.885532656070101 7.304951139776333; -0.4971719330606758 -3.24476164
52632357]
```

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}:
 0.605601  -0.140151
 1.11253    0.117187
 0.257269   0.128535
 0.236282   0.806968
```

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}:
 0.610743   0.204502
 0.971993   0.319577
 0.486441   0.0757771
 0.0289923  0.603935
```

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
 0.03788048531368239
 0.20467587727168945
 0.4744527298129571
 0.7704521341216637
 1.0
u: 6-element Array{Array{BigFloat,2},1}:
 [0.610742676754586488740983440948184579610824584960937 5 0.2045017983579173
74815428956935647875070571899414062 5; 0.971993097531425265245275113556999 71
31824493408203125 0.319576889858146140355188435933087021112442016601562 5; 0
.486441473070317709215260038035921752452850341796875 0.07577706275657702938
98594900383614003658294677734375; 0.028992311136734949528204197122249752283
0963134765625 0.603935080770206633715702082554344087839126586914062 5]
 [0.620881820658283203504461273828570975101621889052719067321251485272263515
770513 0.087521752862484193681398833254176468990182839606411245650482591166
81480416185191; 1.047676831764527083906845103250306581652823062395592021699
260673556972690421968 0.257781721113545802269263534677019398431119020806019
457116005667092506576016800 3; 0.395589352722751562652165617891223614264428
811921650350524811484696495829322708 8 0.077908489332052684410618859955897866
555585833165028508438664432353445126043575794; 0.110531706598935047495257690
525299793647080761613561975122279559484233131018708 8 0.689137656101209021 8
937177211709570362794258135609415254587120977856919954588972]
 [0.484127559069267658309337641994305696135474236695955626273624197340294840
9768495 -0.651728525209022002911439699661894044379953352330375323043627829
410055871702399; 1.069150580315447969267933101814828009403659375158995250721
271580547821279994499 -0.218426231286807451488228387345310248517230849487156
746219831301248541105973287 8; 0.057136713908489553567982704550008761709879
872142106500817317038031596226806395 21 0.386694187641785030261076437253624
362696464242767371662087057006373209367961467 1; 0.436434941683587414882409 6
665527338843175754981171597827263334193629939596635352 0.944308389516635576
593837714693897129215727491771957844800167527959300207479744 4]
 [-0.298165260808952326664326624459930988181370486188485832645097647425605 2
628534215 -2.256595002995211629494109271898128779089816567013072274044672 9
229761126220399; 0.312086087541834463710205017626299049613402443664921946 0
566245998421553396456 21 -0.924942309409075863496899536862360074849461888175
955032826987395394250605426920 7; 0.067990689321603110074401548608495666931 0
526198634807027945648638186451183298671 5 2.189670687079368340167445659881 2
812435989428115615494398249641170495398688074 2; 0.710327181840630340457285 8
936830896505370058059045079889688721207225539274997665 0.686518849837398518
397504562426057987271259585882040536328967679606010581896878 8]
 [-1.462755329540620902414815629000550622873140623915885943143526065700678 3
```

```
67911558 -3.074210909026630096493735459469761388450863179852393457045206000
838088236535704; -0.501905385640268304370907822269544405309007416209988272 6
75384284395665757768660 7 0.5441964988775841358354898222758081331668570909 39
61248256096815993946658670214 76; 1.304911856257153035529909064434493633408 5
13710165526822336385476139406638459792 5.559308849940339074461446482012078 6
29509978828332306368018275053538411113091068; 0.370541746155783269884749280
84607418482420229397805044671033994943646318740699 13 -0.986807989439311850 6
40671489131169056177412791119560751145853153767877019914171 1]
 [-1.837994144793673947361253723483277109662532826680885885283052459216110 9
61807102 -1.258261205079914864606808516015105125197862336475965917047756447
103748396369034; 0.007613190524543504221409266784841169599655556939469447 64
92272358058280159464808 03 4.722155721401007674899718381631005032487647067 62
126969997990187124372765950469 5; 2.885538905535507534856006097880038339693 7
58466736340087304030903039705823837946 7.304959178441350922207708221884888 1
25352265928420269505515636056252473239718279; -0.497174943813181458747346 41
51805657631380839232493789670712244186043812457472982 -3.244771031621865588
68335010744092996019423127838814195826310699605798547139517 9]

sol[1,3]
```

0 . 4 8 4 1 2 7 5 5 9 0 6 9 2 6 7 6 5 8 3 0 9 3 3 7 6 4 1 9 9 4 3 0 5 6 9 6 1 3 5 4 7 4 2 3 6 6 9 5 9 5 5 6 2 6 2 7 3 6 2 4 1 9 7 3 4 0 2 9 4 8 4 0 9 7 6 8 4 9 5

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
 0.037880485313682388398084516187830238264143717656443217945847242874606236
07108105
 0.204675877271689452417522856815077613517093133111538903522066929440925188
0275945
 0.474452729812957085027989974099192943137032342325685064163471482391289143
9940861
 0.770452134121663686554807451511767273978335027743544281845128143757924344
2405248
 1.0
u: 6-element Array{Array{BigFloat,2},1}:
 [0.610742676754586488740983440948184579610824584960937 5 0.2045017983579173
748154289569356478750705718994140625; 0.971993097531425265245275113556999 71
31824493408203125 0.319576889858146140355188435933087021112442016601562 5 0
.486441473070317709215260038035921752452850341796875 0.07577706275657702938
985949003836140036582946777343 75; 0.028992311136734949528204197122249752283
0963134765625 0.603935080770206633715702082554344087839126586914062 5]
 [0.620881820658283203325296517057406072350582641201578181713028698618305 93
97874694 0.087521752862484202500456221531692945112574889888319363277413078 9
2259811848597719; 1.047676831764527079602749481795278680223216751586154099 8
751099762986629785343 04 0.257781721113545807314773296888193076871535703460 0
330613925289708786167832336579; 0.395589352722751568884013858561773193788 58
780584160981541064882750932652367928 79 0.077908489332052683520220165281804 1
127885767306264940384327995657555167288700942; 0.110531706598935041896748 95
805984602485868209848947722551585770984065860527610 87 0.6891376561012090162
```

```
5989773169378367975173071242752546891362076066134721001231851
 [0.484127559069267685873746517326989166897698124938410058777811893228293218
8191398 -0.651728525209021915687110054830072292866588002329803625140349494
5420398036534489; 1.069150580315447990087556461511951098934469531931924972954
54877843339619302088322 -0.218426231286807395383551222455320424010603924166
75091701836377703682465630013 8; 0.057136713908489577918713353556716911525204
82633486200135327755241502295513109082 0.386694187641784972614061877971458
84993142429277423984535633152355123288535727 22; 0.436434941683587387190283754
50836636274418032300082837042129992528683873297452192 7 0.944308389516635563
85866529560688341779188343550686804069613170509656419347184641 64]
 [-0.298165260808952184749139156800053660679127305896348124374917274668712213
4935859 -2.256595002995211419773576494416404703411487101871903853153271322065
5263332742; 0.312086087541834599213104829033597991572589950945003104633818182
314590438578674 -0.924942309409075845899646708259067475051975733717803020526
960921239757169867606 5; 0.067990689321603039924562436662584177426952318311
945301535155318681397458686753 2.189670687079367982120937995832383222778249
497840809479348734347083860792334767; 0.710327181840603033485542222378564
355796658052389543530527728578477364590964310 7 0.6865188498373986287723974
584473887120545074529247177035555070825367104900262 14]
 [-1.462755329540620661515243162323026216970881786673271809620253530620289510
747762 -3.074210909026630231627796525666063646569821738923088045715331302359
026109275258; -0.501905385640268250661946387641867288779664996516776143287
724089961586981996674 7 0.544196498877583277552096215333386864875118321127435
269861388157097235602630014 8; 1.304911856257152583474965922069895669640460
551162890070108725423520729391309072 5.559308849940338252681236827759447827
161520358773036570478164837703601931914988; 0.370541746155783457964801396156
032664597687824954098079654869655813606209404281 5 -0.9868079894393112474766
695000241290129115073494498562235523266979337060978720073]
 [-1.837994144793673992317012881232281522391477729088784372958954469391108781
868799 -1.258261205079915903053513388654487339735294153543247920343471905696
749398837918; 0.007613190524543111030543560117301514806772187006746024233790
435634479505521652825 4.722155721401005976481002157731312985681096586177140
838608748871182612919061927; 2.885538905535507059208614535549761199125506437
739215934420354375536104918377 6 7.304959178441350798121343395106225935638
367179305808804950156319776818775972458; -0.497174943813181116962128707414
232264024973653097321444769917041430046943676843 4 -3.24477103162186483511864
224867821205026233617998844858004123516118539829891305 1]
```

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
 0.04439183904555069
```

```
 0.11771676651298536
 0.20605533246155705
 0.31468451138166686
 0.43028172071784404
 0.5494312304478333
 0.672171629988903
 0.8246118445270447
 1.0
u: 10-element Array{StaticArrays.SArray{Tuple{4,2},Float64,2,8},1}:
 [0.22932371025126863 0.1725729837929979; 0.5988783719673871 0.558270700518
6731; 0.7950484972703886 0.7067286061753983; 0.9088468667880318 0.553382553
293603]
 [0.012849075067361881 0.04120030776233033; 0.5776993047470756 0.5713125721
478617; 0.8172217810810329 0.7145206182431396; 1.0884374943986317 0.6718917
45070644]
 [-0.4485344285794179 -0.24299277221050583; 0.4315666049040173 0.5159508961
853313; 0.967321781916795 0.7976178546428485; 1.3335115454246012 0.83264558
26307495]
 [-1.152993998379832 -0.6805927715048965; 0.12683058328874391 0.35805731722
90347; 1.3729898036383266 1.0375943437650759; 1.5180056218535884 0.95192274
93592948]
 [-2.1723218101647292 -1.3143736547261546; -0.3025827139347661 0.1173815341
6315672; 2.2591195942987046 1.5736374802711457; 1.5302998828056433 0.956179
4039638346]
 [-3.301824520091458 -2.0113103581626004; -0.5918001261556982 -0.0430202146
9409953; 3.6890562112843694 2.444858603592143; 1.2212279017272847 0.7496526
190190814]
 [-4.290514628030032 -2.6074964964195826; -0.399819587666589 0.094254962675
58744; 5.6173876382208645 3.6176975282854302; 0.496547190936951 0.274570300
4772448]
 [-4.813542437416433 -2.89270731213996; 0.6592220748244556 0.77406848603044
05; 7.8714897040049046 4.9747580993720195; -0.7145782579071296 -0.509711845
4606824]
 [-4.207323203709997 -2.4300106541360664; 3.628222244181996 2.6504360523439
96; 10.445825337370675 6.477356765714792; -2.8433862820153646 -1.8701671582
185273]
 [-0.9051234009059521 -0.23176126746763082; 9.605171013153203 6.38692509666
0997; 11.66960663241951 7.03927248529625; -5.881756735513337 -3.77678947507
48603]

sol[3]

4×@*(2 StaticArrays.SArray(*@{Tuple{4,2},Float64,2,8} with indices SOneTo(4)
×@*(SOneTo(2):-0.448534 -0.2429930.431567 0.5159510.967322 0.7976181.33351
0.832646
```

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/SciML/SciMLTutor
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 5.6.0
[91a5bcdd-55d7-5caf-9e0b-520d859cae80] Plots 1.6.3
[90137ffa-7385-5640-81b9-e52037218182] StaticArrays 0.12.4
[c3572dad-4567-51f8-b174-8c6c989267f4] Sundials 4.3.0
[37e2e46d-f89d-539d-b4ee-838fcccc9c8e] LinearAlgebra
```