An Intro to DifferentialEquations.jl

Chris Rackauckas

September 25, 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`.

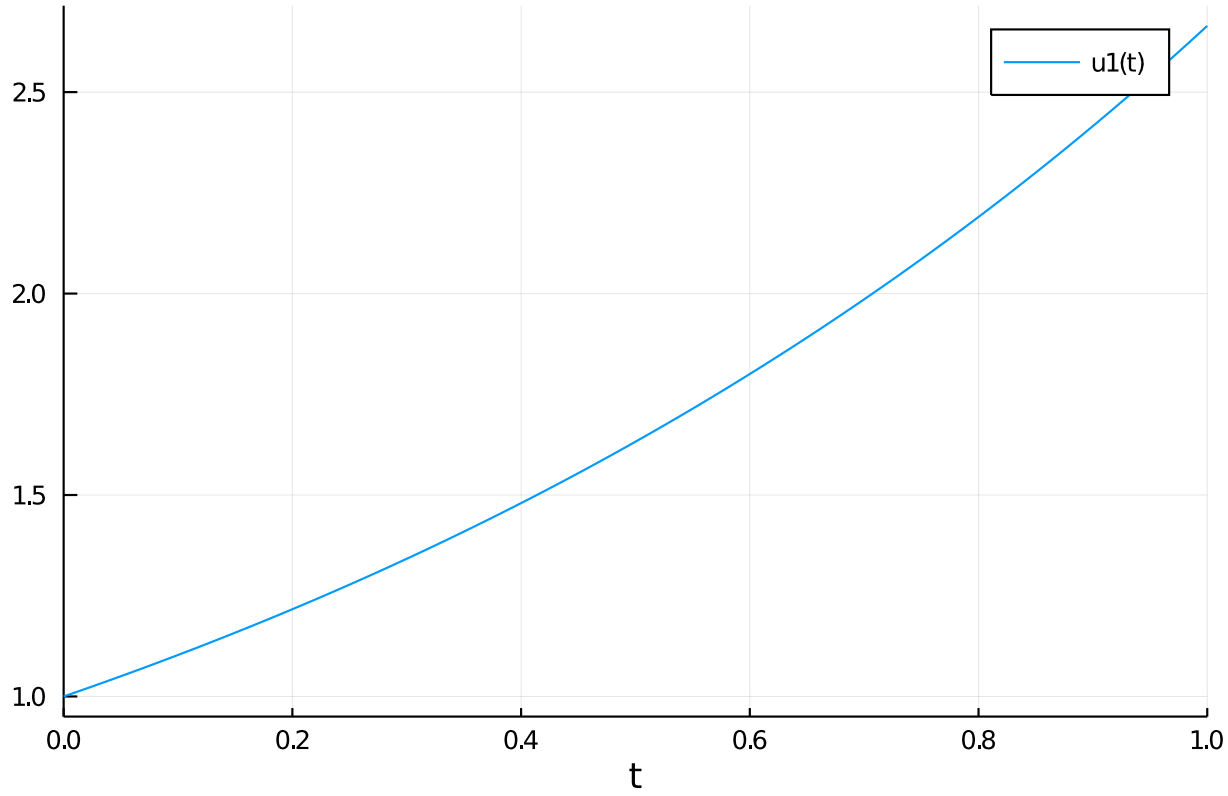
```
sol = solve(prob)
```

```
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!

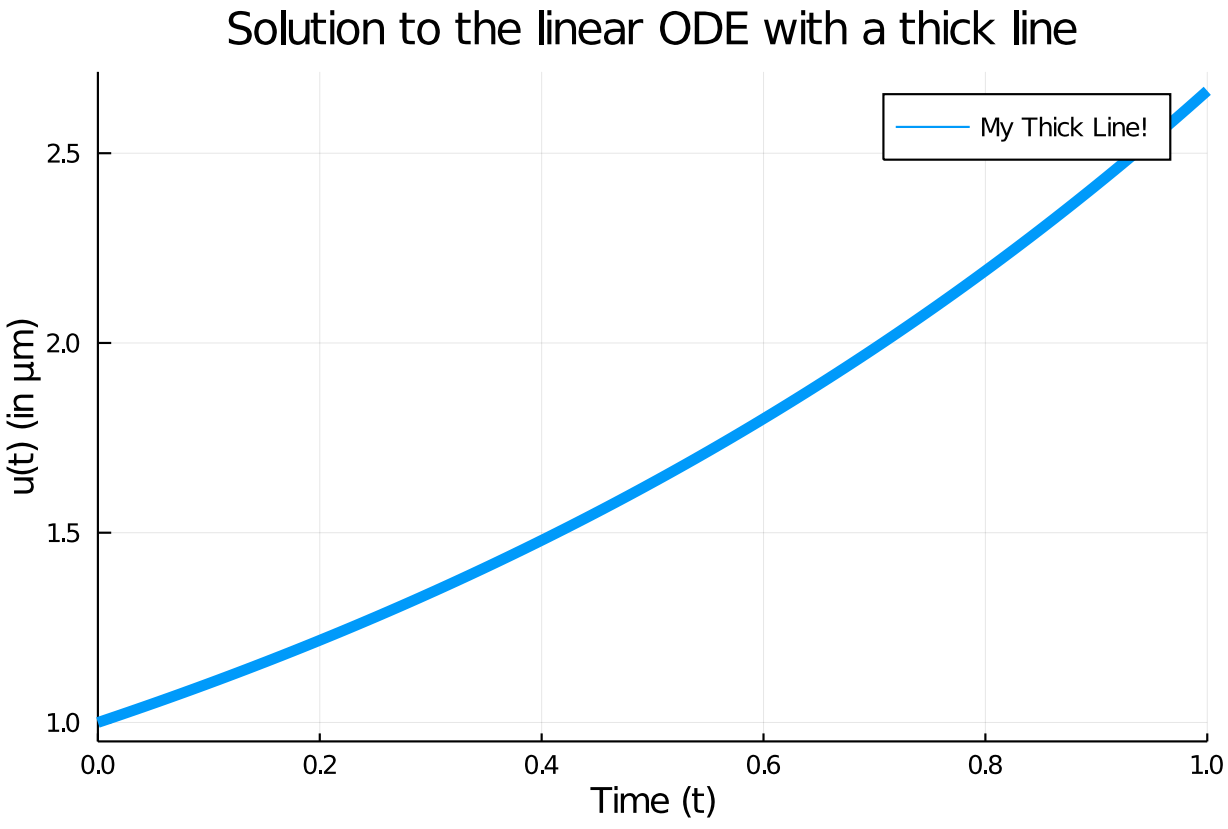
Analyzing the Solution Of course, the solution type is not interesting in and of itself. We want to understand the solution! The documentation page which explains in detail the functions for analyzing the solution is the Solution Handling page. Here we will describe some of the basics. You can plot the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

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



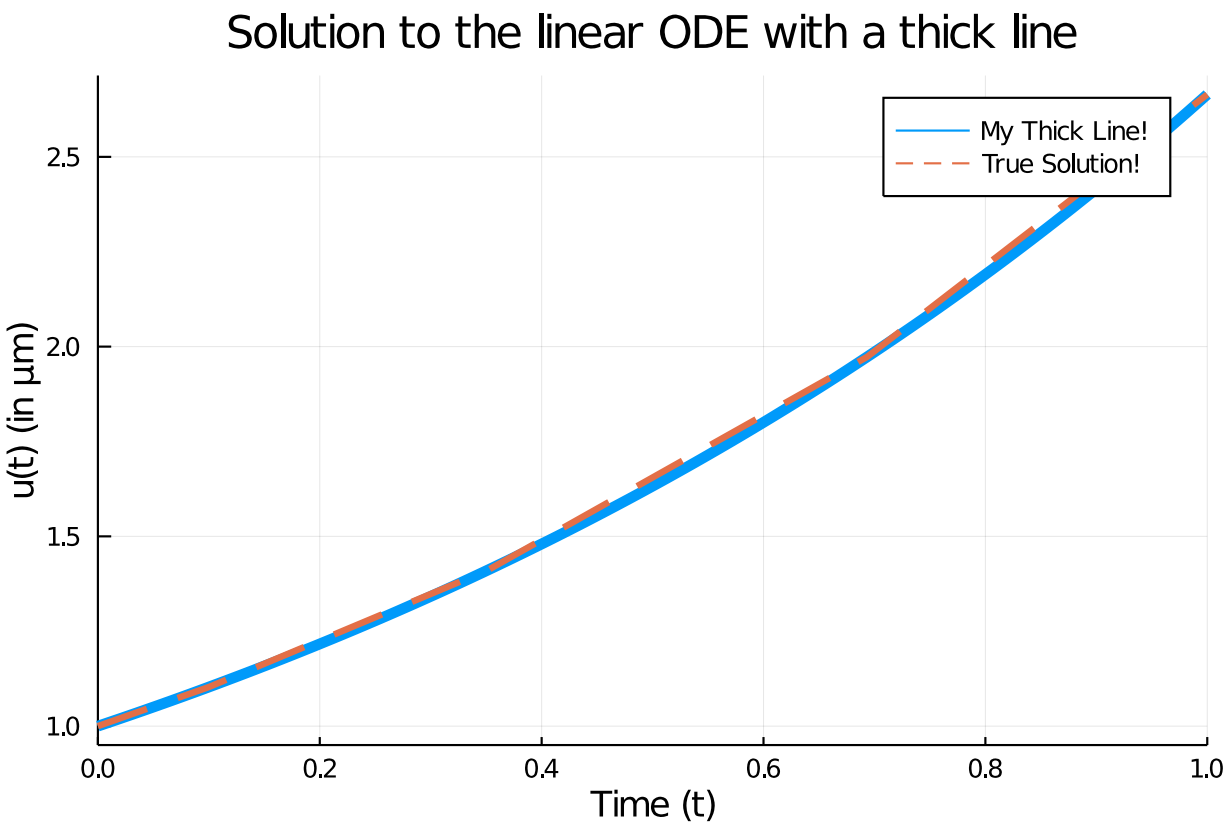
From the picture we see that the solution is an exponential curve, which matches our intuition. As a plot recipe, we can annotate the result using any of the Plots.jl attributes. For example:

```
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:

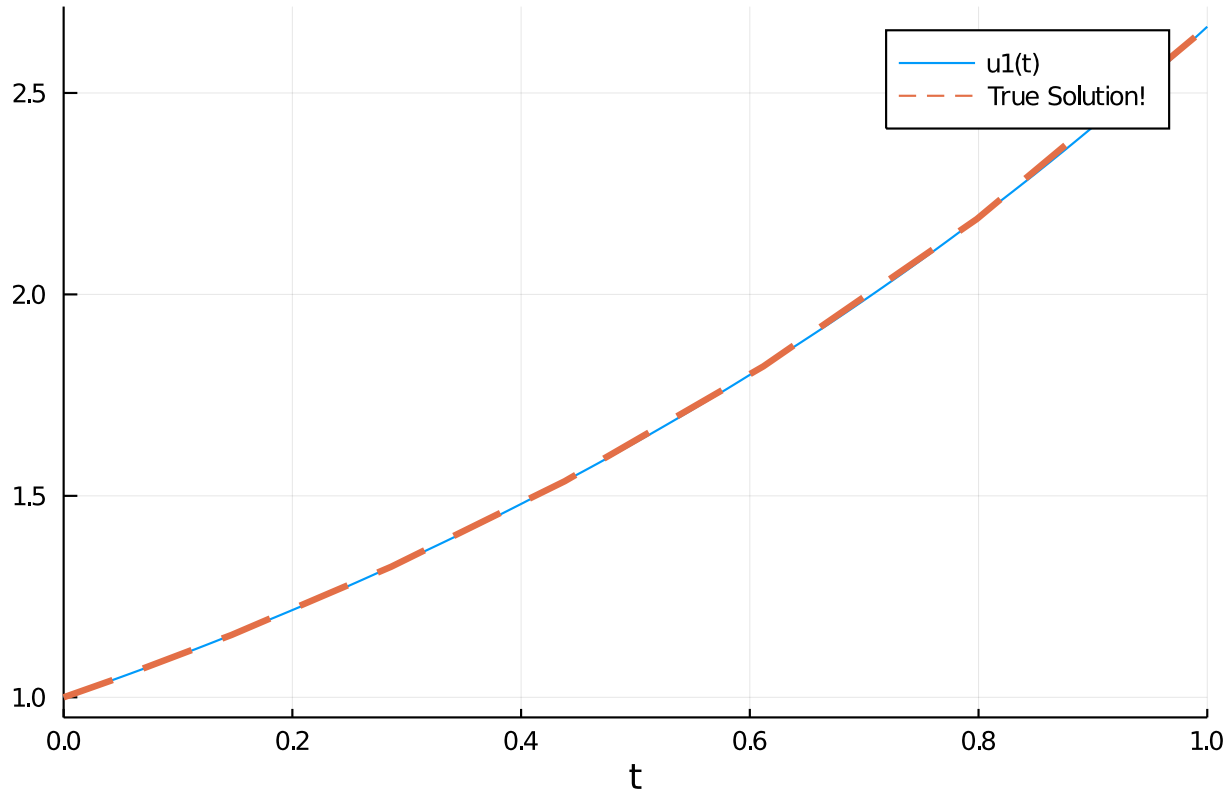
```
sol = solve(prob,abstol=1e-8,reltol=1e-8)
```

```
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

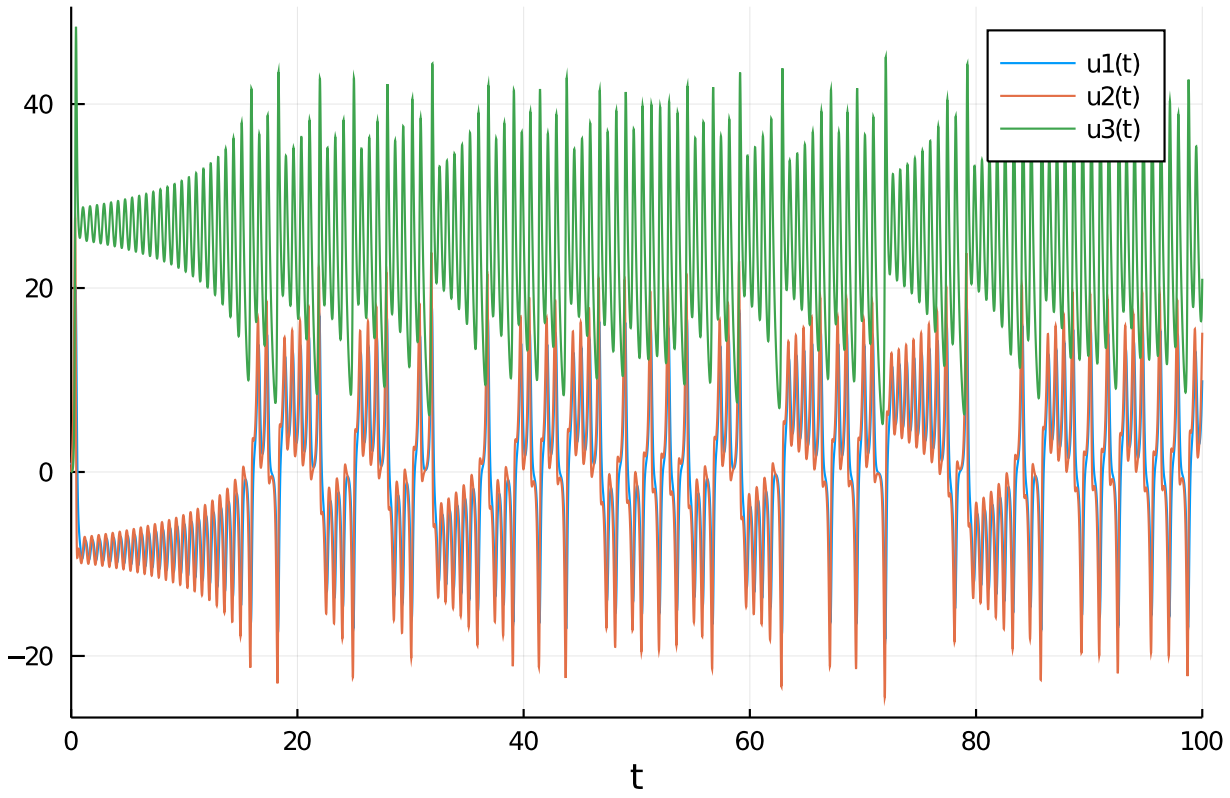
We can get a real matrix by performing a conversion:

```
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
```

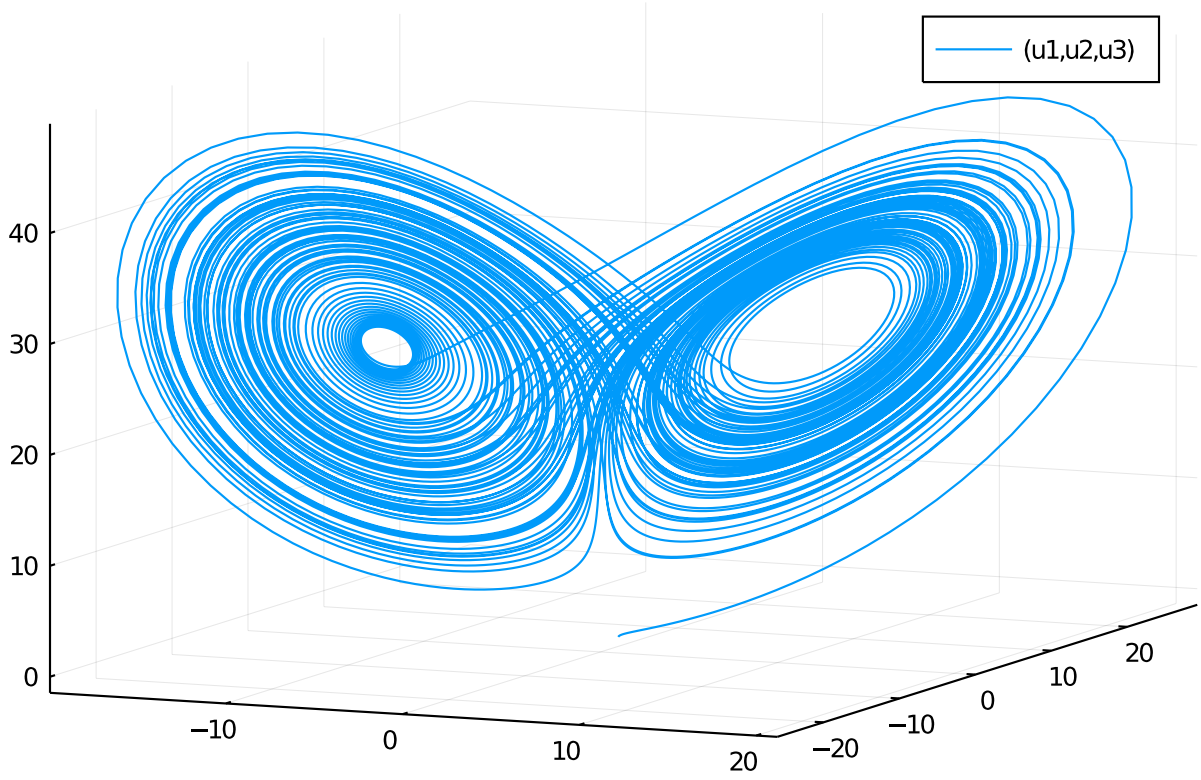
This is the same as sol, i.e. `sol[i,j] = A[i,j]`, but now it's a true matrix. Plotting will by default show the time series for each variable:

```
plot(sol)
```



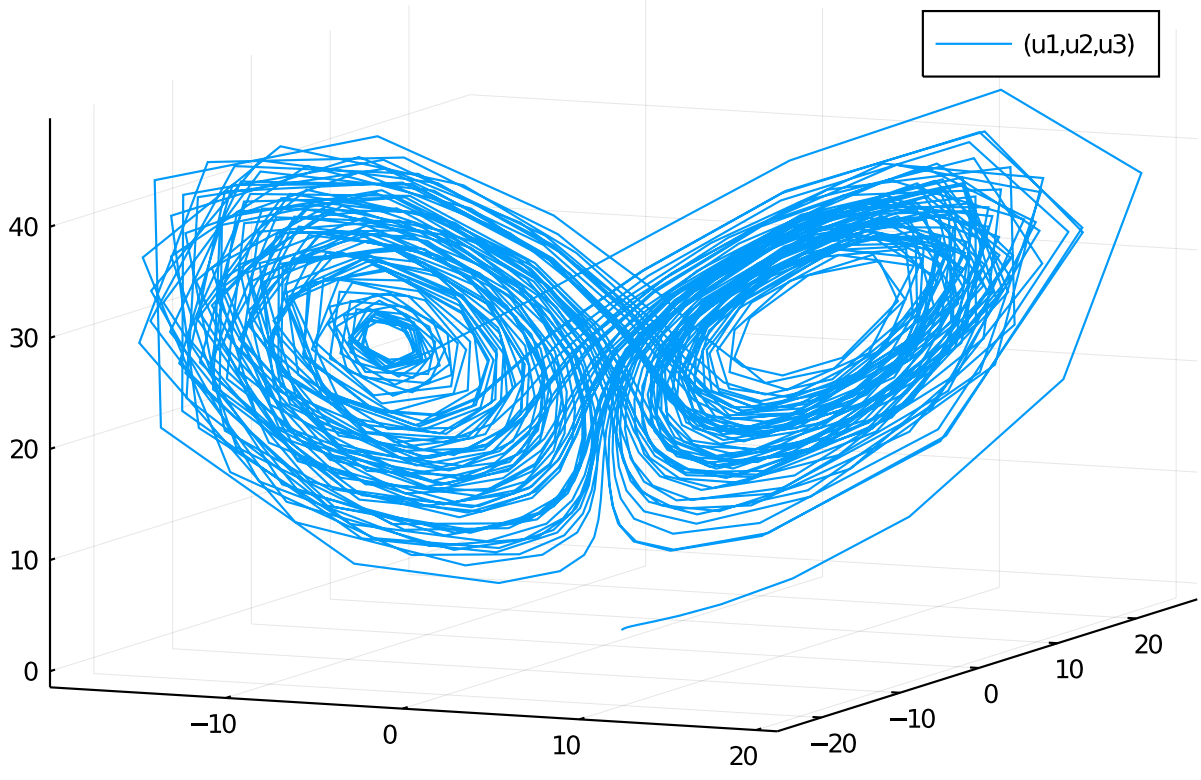
If we instead want to plot values against each other, we can use the `vars` command. Let's plot variable 1 against variable 2 against variable 3:

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



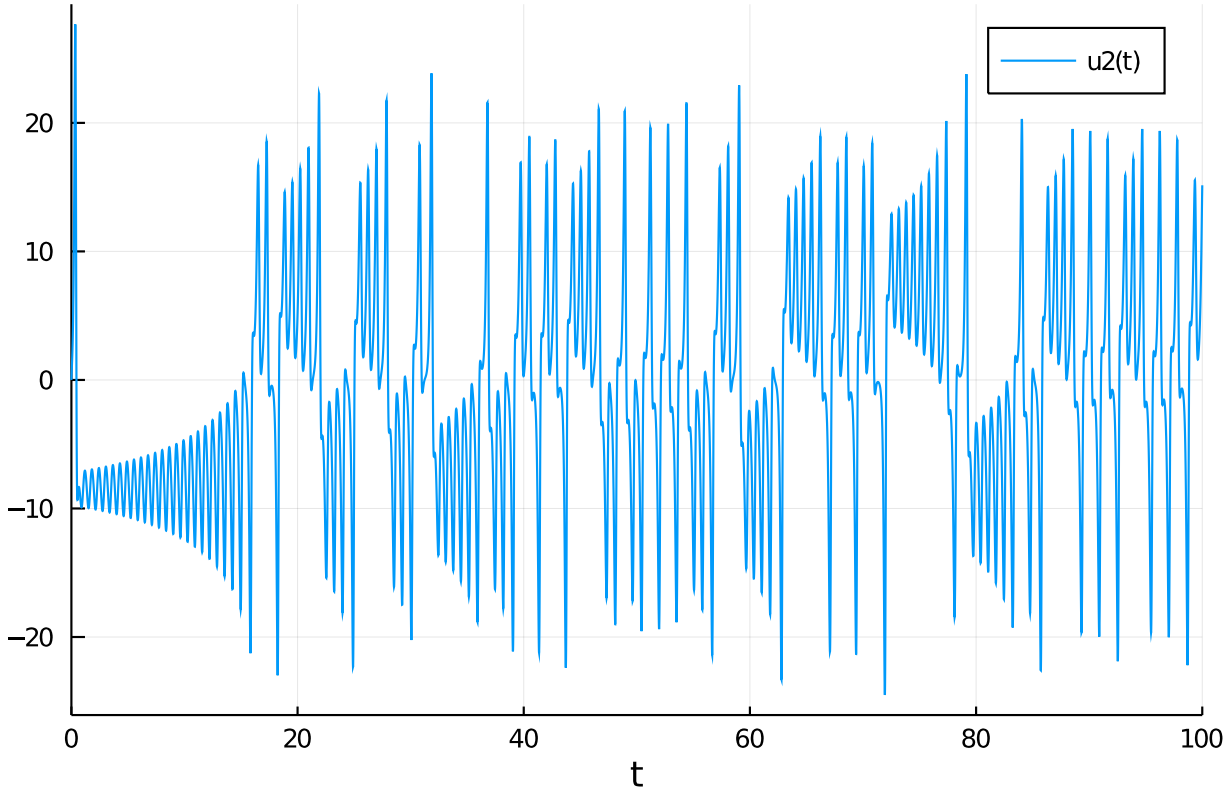
This is the classic Lorenz attractor plot, where the `x` axis is `u[1]`, the `y` axis is `u[2]`, and the `z` axis is `u[3]`. Note that the plot recipe by default uses the interpolation, but we can turn this off:

```
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
 0.04244445270868048
 0.10264882166596963
 0.16050154945121875
 0.23489487313208723
 0.33447641266750416
 0.4563000693825595
 0.6037053473032081
 0.7674242341001349
 0.9371184382013034
 1.0
u: 11-element Array{Array{Float64,2},1}:
 [0.8560802386936441 0.9041991530320039; 0.5172765489558993 0.5337473709429
879; 0.11977614846773643 0.3282858403427533; 0.8253585894813911 0.582782007
4680976]
 [0.6873604362006522 0.790514116484995; 0.4965866147435862 0.58602742587218
75; 0.0282454662359211 0.2135883535498354; 1.073654349495993 0.828262138699
8659]
 [0.3454311739890223 0.5310409175340428; 0.34559169288714264 0.536158877558
8468; -0.023794389946542434 0.11211103915396779; 1.403452398502685 1.159774
5839738023]
 [-0.09442415801356341 0.17371576179894965; 0.0813643033129478 0.3627856353
1801444; 0.03462832209568199 0.10419344871562027; 1.6846702186987552 1.4495
95296577732]
 [-0.8105893427441719 -0.4347811658386703; -0.39349422183436067 -0.01097855
962385158; 0.3019769027914189 0.258982240513327; 1.9752244803643655 1.76238
75912375868]
 [-1.9959667074055458 -1.4815717341582153; -1.164233162295246 -0.6885386976
550212; 1.0628565897972229 0.8233356175871878; 2.1959838096507447 2.0350542
958699407]
 [-3.6757234833809562 -3.020507163239456; -2.0587634945035607 -1.5632255367
554073; 2.705477952922223 2.1653278794826045; 2.1180682303676823 2.05957618
66650295]
 [-5.702136216614708 -4.964074842220198; -2.4984274352003046 -2.16560328043
82965; 5.749078330092321 4.796299563435988; 1.3631889238557533 1.4913911701
354103]
 [-7.209705068452976 -6.572419950200866; -1.2194868824181124 -1.36080072617
97286; 10.132798777104771 8.759362371227702; -0.49333953424606247 -0.079242
82360305845]
 [-6.77979433637902 -6.564953937575206; 3.123241849354386 2.149009759604728
; 14.70277867524787 13.118952350332723; -3.599222114712509 -2.8287661608196
584]
 [-5.827606210385362 -5.870600780155404; 5.662596913011308 4.30329971685852
3; 16.028675212109267 14.474716194196905; -5.016253403287097 -4.11082020864
6803]
```

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}:
  0.345431   0.531041
  0.345592   0.536159
 -0.0237944  0.112111
  1.40345    1.15977
```

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}:
 0.85608   0.904199
 0.517277  0.533747
 0.119776  0.328286
 0.825359  0.582782
```

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
 0.10396426099019132
 0.3189803551515885
 0.6045115879068522
 0.926593118577873
 1.0
u: 6-element Array{Array{BigFloat,2},1}:
 [0.85608023869364413549476466869236901402473449707031250.9041991530320039
3275062651810003444552421569824218750.5172765489558992513252633216325193643569946289062500.5337473709429878532262137014186009764671325683593750.11
977614846773643186850222264183685183525085449218750.3282858403427533211527
133971685543656349182128906250.8253585894813910694978176252334378659725189208984375 0.5827820074680976070169435843126848340034484863281250]
 [0.3366265289951892430182778348126971593968236381874808920770012449047819690582486 0.5240839801451442919655690779839234159403321388601296584840621174427093175855904 0.3408061774210886351156020561021945091230965687489100300460607189798417586304823 0.5335237479824498628417759459587785143618427008001843021177374669405617057343656 -0.023738170447708130997359787231069818604877559998889623111735412588030262829105870.1108652976229023682582725076626953912499740668143999669603514254684611192564207 1.4102819292562876595484936325658388406779671963134046302778090299775104999354381.1667212334633123034735804194973924084058528331414441318469219679682024874943430.7971536478788545944663846514425077063712514849170008379289436739280673074838670-1.3034256544526950264560153608269357825002188141921943576566097376866383839982490-1.0403263478894363277678730592700586604331195497487962437792377589765334644400050.5752157771316485099271199492836680833669227355176355998172791082357933499393756 0.9114376972567753054157788622235681618320779424507074166859520565924513198077843 0.7057339458626108360973124576344472736878894958792950896543981157135378503667166 2.1765191334171024624737762048195050042921634471959617324052242768102033285165192.0057244272617974790184965090066689690529342276238334013380308560759874216759130.7122234787766846633058033857291157424391204073496066384745085208160307052221 -4.9740860290867070298084364176513644667739584418343315509098975851183749508711675 -2.4975212219878032316980284506947167361507423172883000892989727022715170525531650.1662402934437697341804006644129937874266754533667221420362485564521517009149335 5.7685865528968828622504454340569539833474705016889662591578860875486932430037 4.8135305270551064821755493428733766707279553432768351380631037382461479818740621.3567845732320339761319340047976740589488804011663809846306030099246274393766291.4862005253260829801215056511099233004710272253128942138120953855083833256184360.8913235332480013203924441014887759253199070595529152090574379961712500
```

```
95008493 -6.6389987869320697907677537533727063602852304399844369068505072546
671483466693659; 2.7490544983072727205202739395612704809348995615183640295568
6809162904097280094994 1.8357282299700090996758930324148019493809449584302225
2537801112745055306921743956; 14.45168576939329434065638245439584036071814240
4029082052041266488987359572069255 6 12.869707035719918370207129723741821221841
8048544321832729596965087195334525039 2; -3.37445403385254569763056254820030
4540320037390079323208400284946375854011274433 -2.626803512134197873968276
0829387395421390655020526589442009094276483039272624995]
 [-5.827601509909171422103188764334165414374128592358185258123816124880127986
560958 -5.870599492760751044235988364478636350411315078338826426479485991659
736402092067; 5.662641699512320626260291392311738559579440644916082719675725
65134189153985532 5 4.303337630012098389004417366514067799163686082478408158
162651101325158517830261; 16.028715214588039604327399636299967481999263020
3868088085409629336965765471001 6 14.474754325369579176486991189303984328581
2671965660601389999638671289684989028 9; -5.016279107487723840606860997615
8562604090245930897842599777579255781660190095 24 -4.1108426719901786961413
9981740152849051101712356819858731035928814574159777899 4]

sol[1,3]
```

```
-1.797153647878854594466384651442507706371251484917000837928943673928067307483867
```

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
 0.10396426099019132721396362354473041608918024099227779776668430983406640390777787
 0.31898035515158852771716444584467967802773247807772004903106968816934364701373345
 0.60451158790685223193907525793204774135696039257924021849495717070555466442422 37
 0.92659311857787297165340763778621505139411958556923345024293440485713578911885 78
 1.0
u: 6-element Array{Array{BigFloat,2},1}:
 [0.85608023869364413549476466869236901402473449707031 25 0.9041991530320039327506265181000344455242156982421875; 0.5172765489558992513252633216325193643569946289 0625 0.5337473709429878532262137014186009764671325683593 75; 0.1197761484677364318685022226418368518352508544921875 0.3282858403427533211527133971685543656349182128906 25; 0.8253585894813910694978176252334378659725189208984375 0.5827820074680976070169435843126848340034484863281 25]
 [0.33662652899518919867759023871698550026725361316503150278813367636635034841224 64 0.5240839801451442569044419390001402995251734522331366558566501187611431793456454; 0.3408061774210886109410264728505718930756295201603134147089638356980030964828397 0.5335237479824498494538625894115735017580141575261048409043366064590214394384 54; -0.0237381704477081305762198391891896656546499346490290858048237475678768729009 4892 0.1108652976229023621196149061011731409993454853650852278555817838567958719027 27; 1.4102819292562876937866174905084924605474256395627473667465181004185021935660 71 1.166721233463312338305084924605474256395627473667465181004185021935660 71
```

```
850340402101881010559135038187361635105113542565282984456 9]
 [-1.797153647878854784642210574484575751806475455151244056480054414270366 1
12312462 -1.303425654452695196418488162260366029169382372412620880399702442
946051498089127; -1.040326347889436447631534063395849825214592501902859859 8
98149537362247496060962 -0.575215777131648618778063916339950762522373654615
983540868200960522947662773961 3; 0.911437697256775445877707453187973486833 3
92569877514215522331841377421365235681 3 0.705733945862610944377095325627341
233082495087184039675436864669622194094098881; 2.176519133417102484181204791
134774273383389420819689820321449470483563654539494 2.005724427261797509944
343655918737837950051953437003241294298400378307318890353]
 [-5.712223478776684795875717497655181934361344843762878335662975191352573 4
9708111 -4.974086029086707161412331647257489048834420242490619153364807437 0
15894932962449; -2.497521221987803219496342237679748578558358740048143546 35
540524129418818375990 4 -2.166240293443769742331649490324242096893450474737 4
555688478999878775281226741 21; 5.768586552896883119045187546673165458255330
829000574276118622731700792287802841 4.813530527055106709020049316313366623
428663997007490707495037885202175278183183; 1.356784573232033891700783403 91
169945552091595827018419198096813382711791638594 2 1.486200525326082911670 7
455429329540418673159029370965640537258010135286951 6436]
 [-6.891323533248001254676587160555836737448374657168980689417140750054782 3
76493907 -6.638998786932069748003705987383612777597524188241769799101337096
697996974170088; 2.749054498307272950087248499789164146758011652779015916 62
283149345452448702022 03 1.835728229970009291483427590344067888853879173933 88
670976114333902193008629749 5; 14.45168576939329449993209983906592247778920 5
859165684361717090724355674281757 99 12.869707035719918527760115937571304554
151823299954380939328405920187109827886 89; -3.374454033852545837047705425 96
122542349828583717885200761319579424365383931825 5 -2.626803512134197999115 8
232628342307439458519553402107520475579720769405853324 31]
 [-5.827601509909171422103204774989560711070018486715608301545942047953586 2
60729707 -5.870599492760751044236003562454840273642378359938725741974453324
818158499841412; 5.662641699512320626260290981918698956239156707137837335 38
316402841548502135255 8 4.303337630012098389004415300564262711827095188842 78
085361742396026091178065824 6; 16.028715214588039604327429492339057209282919
317923239904061375578719401757309 47 14.474754325369579176487017766252647897
563038025331451931951646682610235664502 55; -5.016279107487723840606866344 36
132771574150746604896046030355954807230702142053 5 -4.110842671990178696141 4
040159820169729949006477929916796212241220598430759011 15]
```

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
 0.03466471968189963
```

```
 0.102924084886496
 0.185727005546646
 0.28613066860277514
 0.4078690959069953
 0.539774497821959
 0.6917346336224831
 0.8455349386291112
 1.0
u: 10-element Array{StaticArrays.SArray{Tuple{4,2},Float64,2,8},1}:
 [0.6678548507659938 0.7442807739163952; 0.6642605720645545 0.0944976477843
4133; 0.9497717547596194 0.8870977440325551; 0.6185692703188139 0.819094165
3248551]
 [0.5650061212970365 0.6026983612782604; 0.7533652244898965 0.1973891192759
757; 0.8887291847182903 0.8261395335742894; 0.8147322009103141 1.0843475418
372746]
 [0.2534255626066652 0.1752111718163571; 0.7905278252218471 0.2231557857362
2424; 0.8418015567167034 0.8064995889158991; 1.172158854069822 1.5742338003
095222]
 [-0.3105384345849445 -0.6008125490858591; 0.6210718291275006 -0.0218867424
35127754; 0.9584509120630239 1.0214926367071773; 1.5311512895446837 2.07856
87880676074]
 [-1.233059985151846 -1.880502240847907; 0.192310959861965 -0.6083414525427
511; 1.430005152892541 1.7400718966224606; 1.810098607429582 2.491933462867
06]
 [-2.5937164002634336 -3.793827187463698; -0.4306426159030017 -1.4499236898
907477; 2.5802099124181073 3.424017625173468; 1.8369264074622103 2.58348249
8153002]
 [-4.1279910408650995 -5.9997192850577115; -0.8295592661177034 -1.980739229
7613169; 4.575878177916328 6.323038292396253; 1.3740128701237269 2.01513522
47952765]
 [-5.476712625862622 -8.037715381650798; -0.26429643874022435 -1.1774227540
103501; 7.655753601753048 10.83086674279012; 0.08811947196200043 0.30861136
18788058]
 [-5.64189953616111 -8.51189415924031; 2.176323318028735 2.2507392410318317
; 11.017722456737935 15.869629815221735; -2.077998582449022 -2.666942096674
1457]
 [-3.7021605136081677 -6.096456606288779; 7.070648130499037 9.1816742239409
29; 13.497291235530534 19.847963588894842; -5.000478936357684 -6.7890713178
21476]

sol[3]

4×@*(2 StaticArrays.SArray(*@{Tuple{4,2},Float64,2,8} with indices SOneTo(4)
×@*(SOneTo(2):0.253426 0.1752110.790528 0.2231560.841802 0.80651.17216
1.57423
```

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 1.6.6
[90137ffa-7385-5640-81b9-e52037218182] StaticArrays 0.12.4
[c3572dad-4567-51f8-b174-8c6c989267f4] Sundials 4.3.0
[37e2e46d-f89d-539d-b4ee-838fcccc9c8e] LinearAlgebra
```