An Intro to DifferentialEquations.jl

Chris Rackauckas

August 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`.

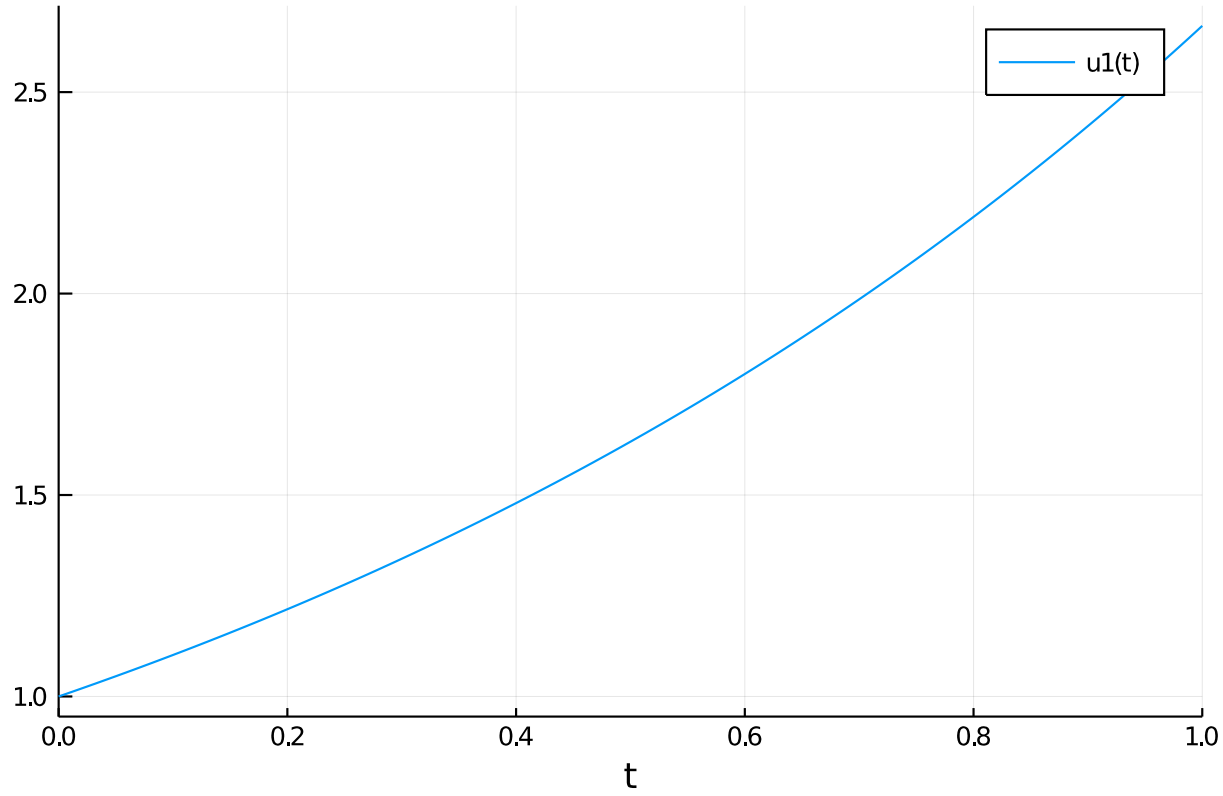
```
sol = solve(prob)
```

```
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!

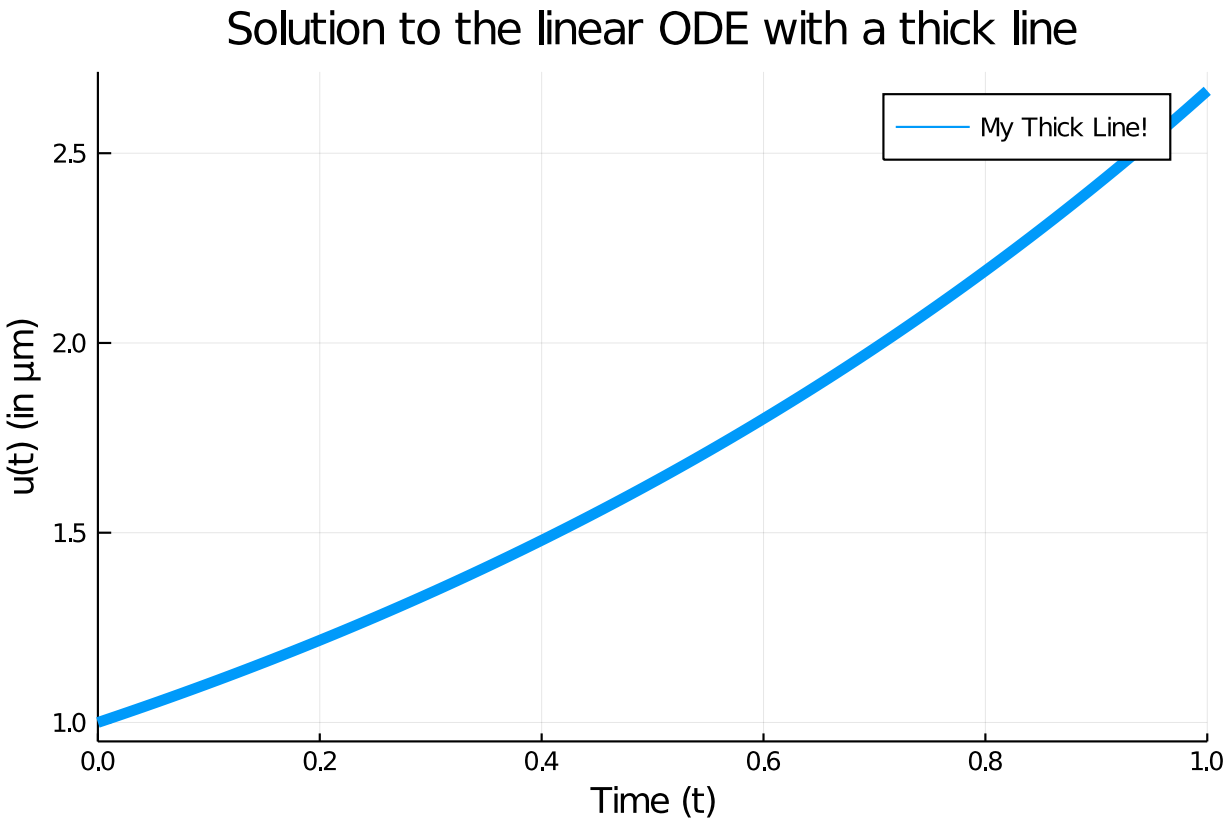
Analyzing the Solution Of course, the solution type is not interesting in and of itself. We want to understand the solution! The documentation page which explains in detail the functions for analyzing the solution is the Solution Handling page. Here we will describe some of the basics. You can plot the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

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



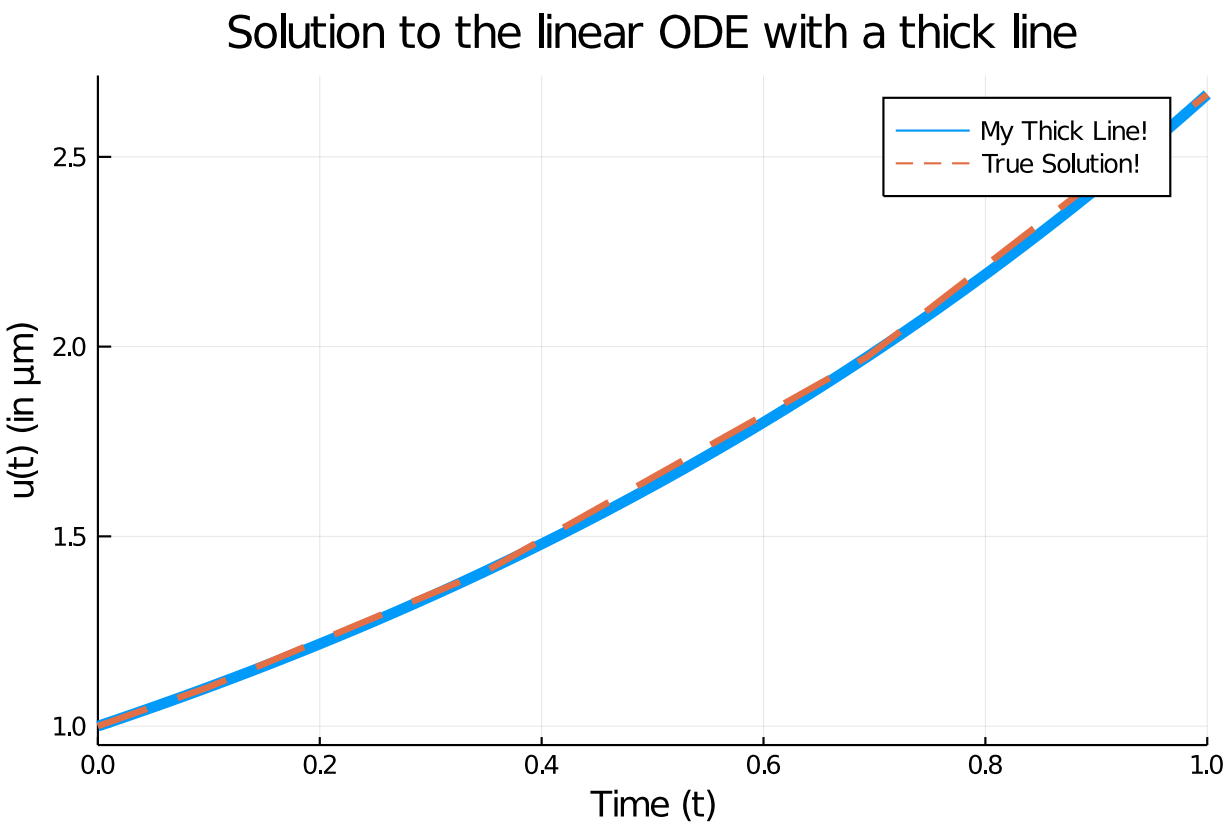
From the picture we see that the solution is an exponential curve, which matches our intuition. As a plot recipe, we can annotate the result using any of the Plots.jl attributes. For example:

```
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:

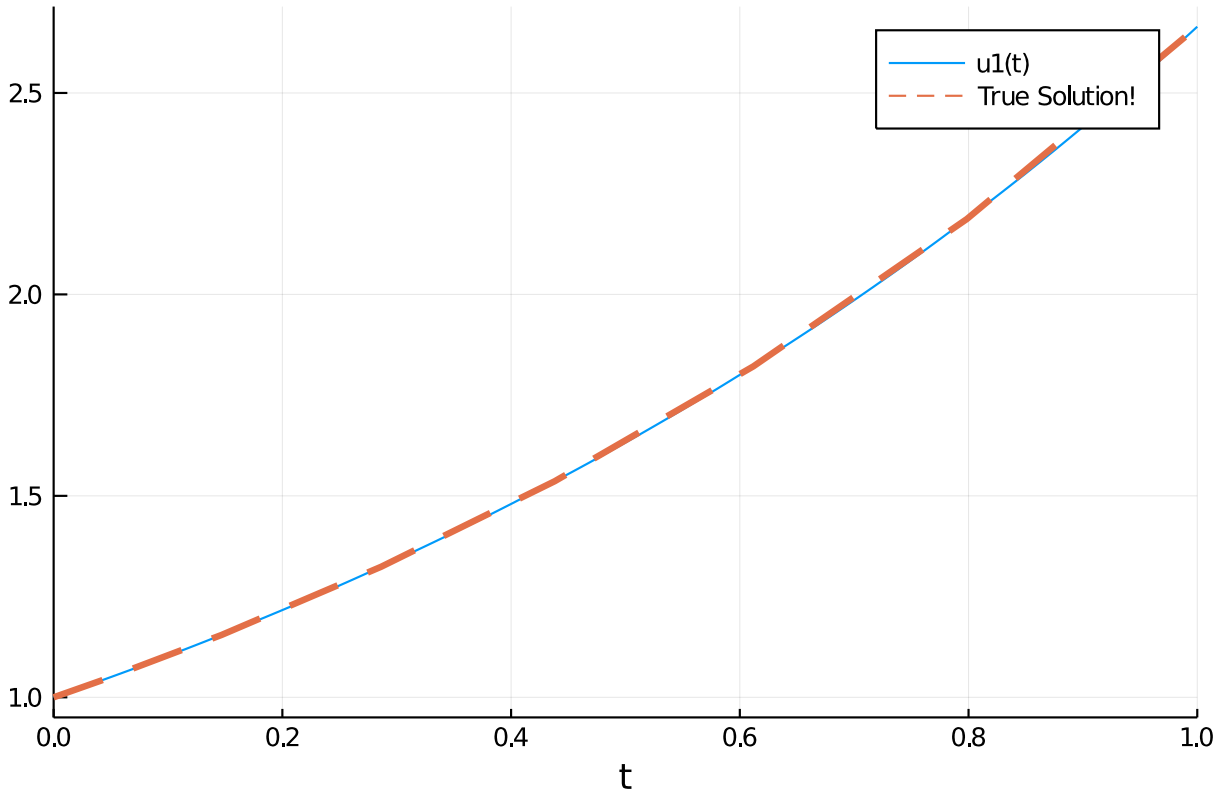
```
sol = solve(prob,abstol=1e-8,reltol=1e-8)
```

```
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
   ⋮
  99.39403070915297
  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 . 0 5 2 3 2 6 1 5 3 0 2 9 0 4 2

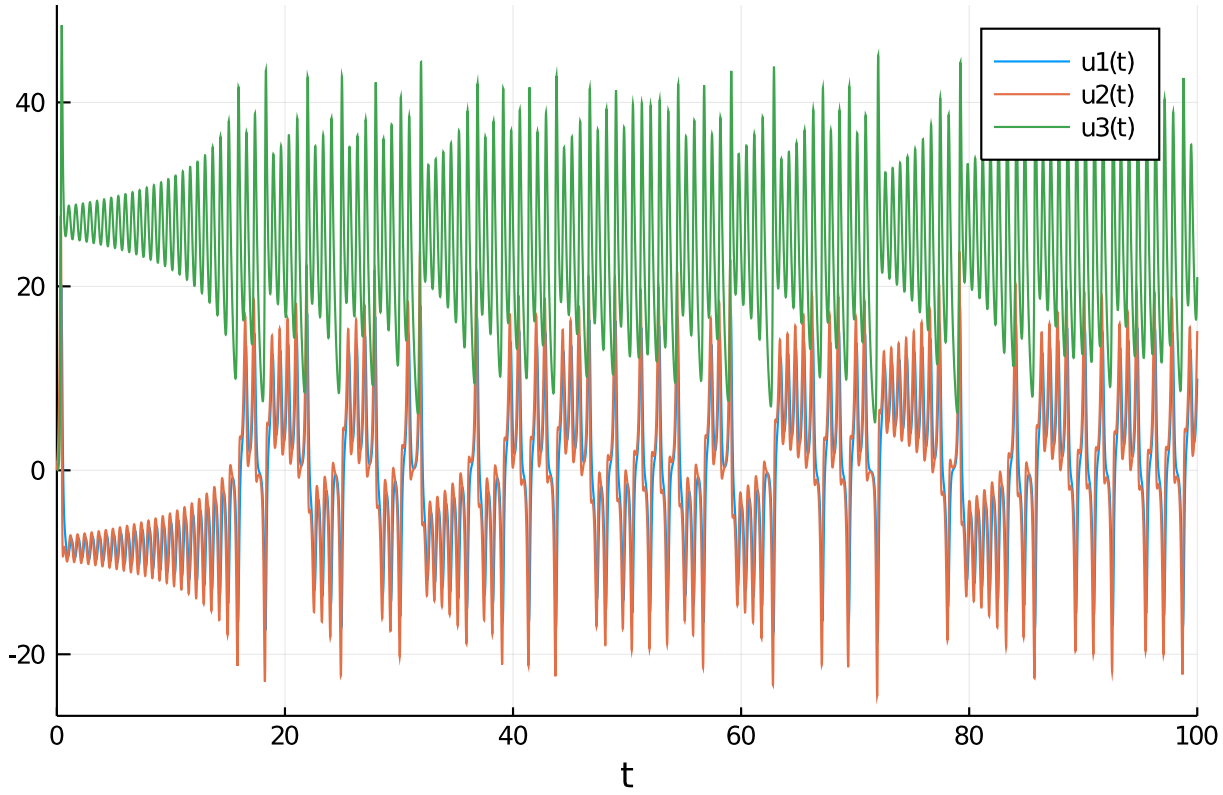
We can get a real matrix by performing a conversion:

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

```
3×1294 Array{Float64,2}:
 1.0  0.999643      0.996105     0.969359      …   4.52712   8.04367   9.97538
 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
```

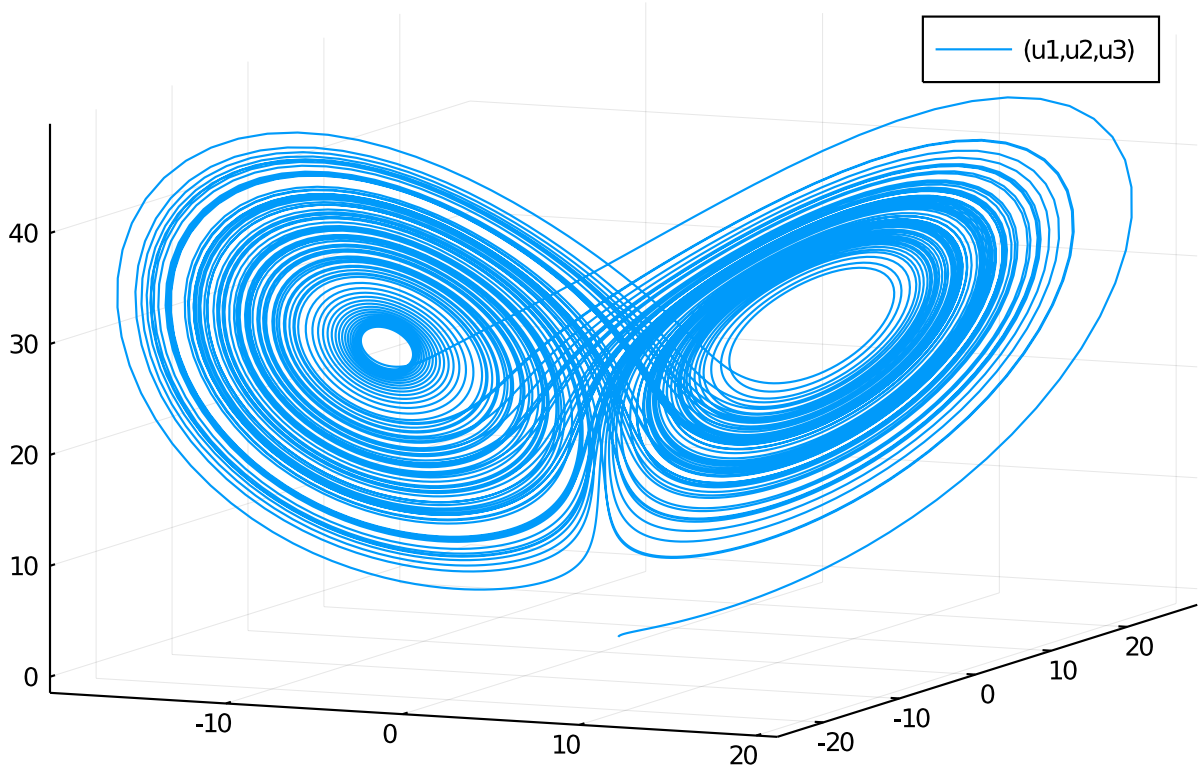
This is the same as sol, i.e. `sol[i,j] = A[i,j]`, but now it's a true matrix. Plotting will by default show the time series for each variable:

```
plot(sol)
```



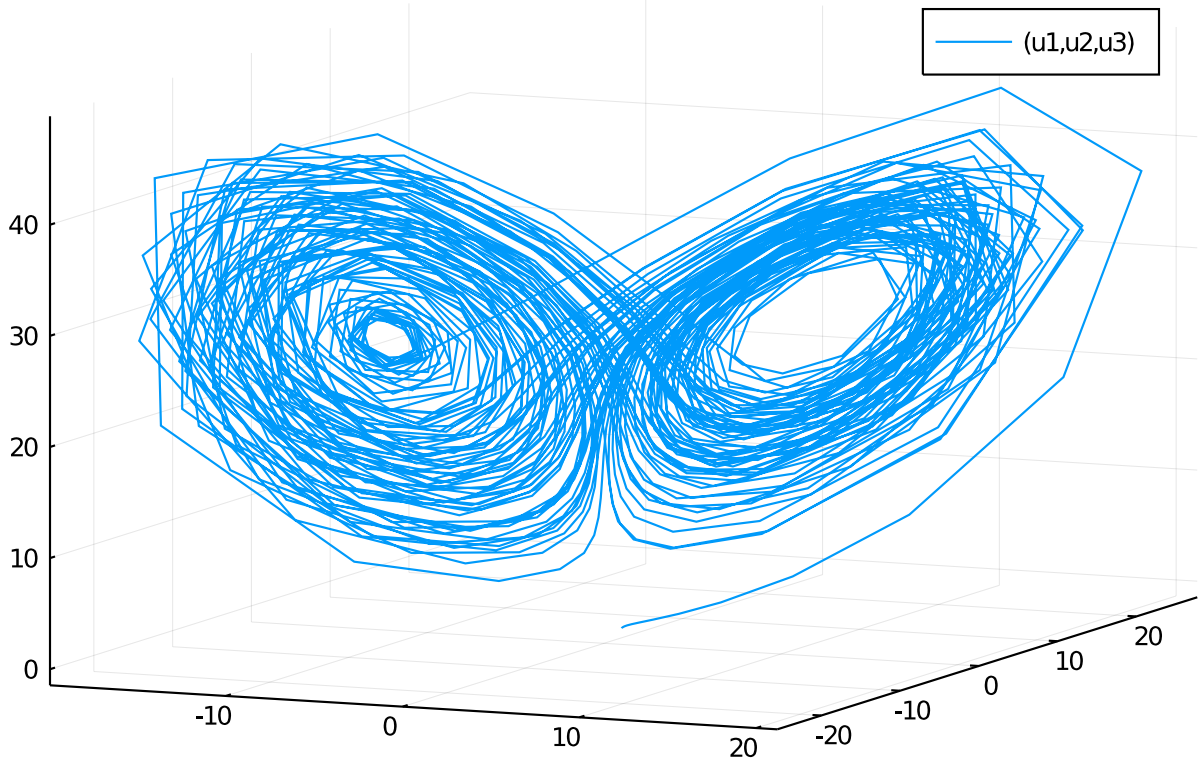
If we instead want to plot values against each other, we can use the `vars` command. Let's plot variable 1 against variable 2 against variable 3:

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



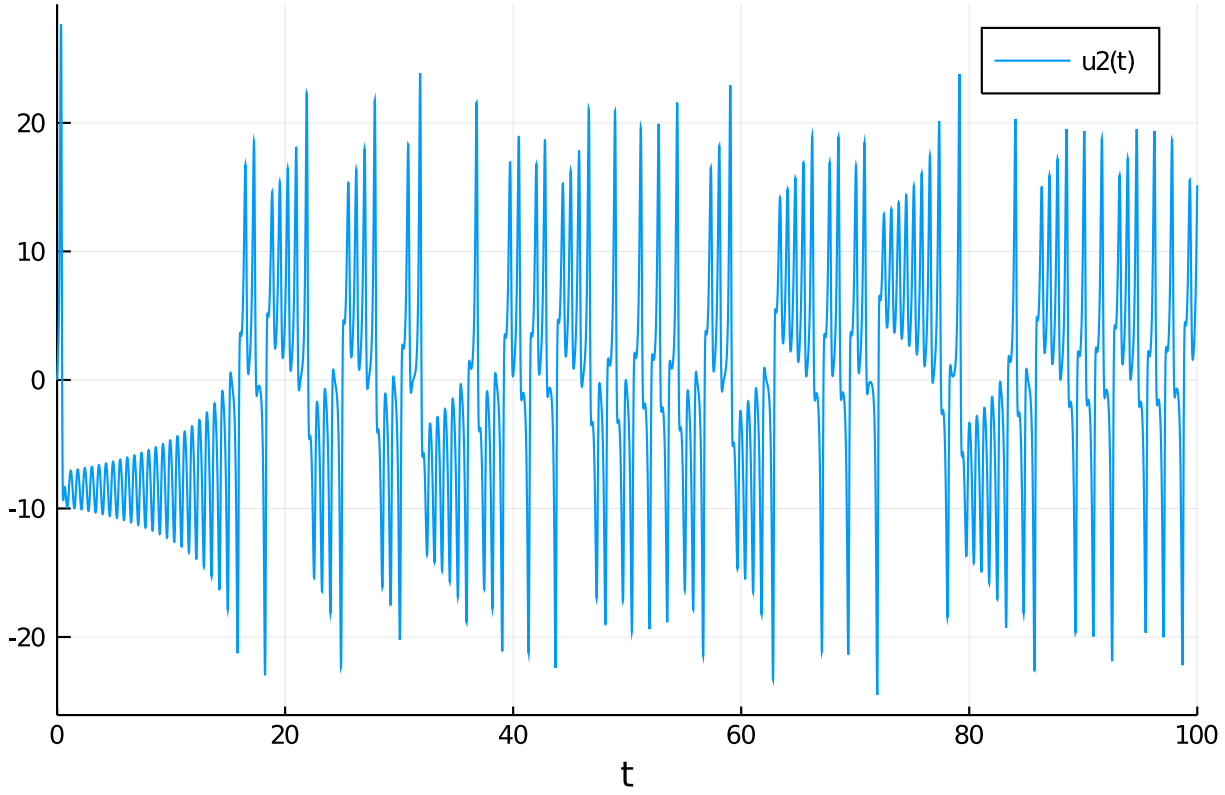
This is the classic Lorenz attractor plot, where the `x` axis is `u[1]`, the `y` axis is `u[2]`, and the `z` axis is `u[3]`. Note that the plot recipe by default uses the interpolation, but we can turn this off:

```
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
 0.039046412040314865
 0.1069960433294535
 0.1859396414186904
 0.2852928762033361
 0.4051310770683696
 0.5450012184737547
 0.7013340641388637
 0.8630233010456608
 1.0
u: 10-element Array{Array{Float64,2},1}:
 [0.8981761394802541 0.566529483626337; 0.959908781245608 0.147285084606845
42; 0.6923834756571927 0.5182233752903855; 0.8255563241254409 0.94388543425
92153]
 [0.7450844320819321 0.3767925204802954; 1.001218916555126 0.16204534816577
46; 0.6004240596467894 0.48550562792573587; 1.071281145645676 1.18830441461
9529]
 [0.3510059344303652 -0.08366823536565304; 0.9133620772093398 0.04233257157
929836; 0.5342751640195939 0.5362336843145674; 1.4646593007295692 1.5773827
811454133]
 [-0.3018989393983209 -0.816717818067209; 0.5922665240857082 -0.29041806837
86602; 0.6524560807508272 0.8128044329988469; 1.8436522126537627 1.94598413
47354732]
 [-1.3851840127689243 -2.0012562906680116; -0.0441497265839329 -0.896459808
4795123; 1.1793398192886073 1.5735096578921979; 2.1532270418012547 2.229322
4706780634]
 [-2.9664137190487914 -3.6924273954244082; -0.910494996246888 -1.6450006313
624517; 2.478272571132553 3.2008852057293606; 2.19238452473399 2.2078718748
333204]
 [-4.900208648074159 -5.704927032073844; -1.550898490138866 -2.006989034947
9547; 4.960541000170109 6.109084179537367; 1.6424255643333927 1.53220665032
76472]
 [-6.561827721108058 -7.323104485305611; -0.952902931214553 -0.874496499614
9791; 8.74351387475751 10.35757349541721; 0.11534469720318374 -0.2205309256
97497]
 [-6.757720201684531 -7.2246726751028305; 2.105419143041153 3.0208892591226
59; 12.982958998330865 14.931431014677134; -2.5437559361889104 -3.205985716
2947316]
 [-4.841792768101085 -4.776613686656013; 7.176756773486831 9.01874707240543
2; 15.719148275221048 17.678533002750665; -5.542211546252572 -6.52735433520
3524]
```

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}:
 0.351006  -0.0836682
 0.913362   0.0423326
 0.534275   0.536234
 1.46466    1.57738
```

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}:
 0.898176  0.566529
 0.959909  0.147285
 0.692383  0.518223
 0.825556  0.943885
```

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

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

```
retcode: Success
Interpolation: automatic order switching interpolation
t: 5-element Array{Float64,1}:
 0.0
 0.16502322639834272
 0.41957005271629544
 0.7230089870606555
 1.0
u: 5-element Array{Array{BigFloat,2},1}:
 [0.898176139480254098756972780392970889806747436523437 5 0.5665294836263370
38160158954269718378782272338867187 5; 0.9599087812456079671363795569050125 7
77721405029296875 0.14728508460684541780949530220823362469673156738281 25; 0
.69238347565719271514694810321088880300521850585937 5 0.51822337529038553505
24509049137122929096221923828125; 0.82555632412544088793993068975396454334 2
59033203125 0.94388543425921533902567261975491419434547424316406 25]
 [-0.10947537943961168961502994866302683756571140599781534674530867772460156
85945686 -0.60286095943306818210486811822112736156167626750043555004162734
3447645903048013; 0.69710951136444044464116614939624133797500820485636611715
98397227463647990581583 -0.18534198994535765169347422690480328943028948412
90574439110993627376614871660448; 0.59771230151651622832628865200205604514 8
40231854887945075143017433650995420499 96 0.71370731595482945382169964469393
06348333106797465662362518856632242873236061595; 1.75308635646768756787548 6
28206290349073043027875009538376490918144572675879237 2 1.858902022774533621
17836213764424750901969603526972391159841410827161483636507 4]
 [-3.16812798475842832826321485171809331274774461914252455596140912622056 14
10330754 -3.90546870601761544225763227905253359709193533525224041278922896 1
427252492192838; -1.00532031398972378139436073974762701569570614415467373985
514800573166185040045 57 -1.71867719424487543964951010678173799996137063029 1
990821034604600748021477236319; 2.68690650622930820637148592278505442223403
210311394514438047671783851540347312 2 3.45193303159020009856177719373912671
057903197649833027657973250113367953242441; 2.16763138844736872392592690985
022290695433937305404092411799364277554646014071 6 2.17313726431896459987610
579025983502498550854046626112376756880471323501231190 3]
 [-6.70242222244720369668589722159506999981537678548679898839680834638554 40
99082035 -7.44148857127922227761769690720134264579537912409941346610016720 8
914579295324264; -0.70710673133541696686633580834956387946653285169778842773
972279439792135751654 73 -0.53367830822736528621271112511700145968265106782
509503319415887848832987035985 52; 9.31809603078748595334419961830212617068 7
382516928637805494119610793879238335269 10.98953528986664311250833356541592
120418631824357271531156746548252113760099 54; -0.17799126639814724875925313
175139824902515293956907092117879688275845688570618 58 -0.552513276570957207
255238963425450130628750844118455556811232704638611911142262]
 [-4.84178297015785640181871917227238665265864616169688178310988340158843 91
31696164 -4.77660029887503542139969131482641448100830492019513361529542686 6
277154689378973; 7.17679826485600418876176595724475017109983797565323669047
964500358044215216783 3 9.01879615837993026117598992233275241487365704555277
240044761263701712356511894 5; 15.71917545380199183891778160509960577903354 3
265852195427526408570088048724722 11 17.67856196660179907685266364198765840 4
```

```
970577357164092365665658402210928854862 31; -5.54223386422344592328181835446
418126383955098057008697887973418670356095026391 6 -6.527379575572761901769 0
200607698060141019413285468653969361419686669014290441961]

sol[1,3]
```

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

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

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

retcode: Success
Interpolation: automatic order switching interpolation
t: 5-element Array{BigFloat,1}:
 0.0
 0.165023226398342726815050682973054131641368509660574926187338908051198006
6216752
 0.419570052716295498332379067019548429047016914355522527114431215638799275
7434687
 0.723008987060655566094279389772539116679370111908820529776405510220688843
7768299
 1.0
u: 5-element Array{Array{BigFloat,2},1}:
 [0.898176139480254098756972780392970889806747436523437 5 0.5665294836263370
381601589542697183787822723388671875; 0.959908781245607967136379556905012577
77721405029296875 0.14728508460684541780949530220823362469673156738281 25; 0
.692383475657192715146948103210888803005218505859375 0.51822337529038553505
245090491371229290962219238281 25; 0.825556324125440887939930689753964543342
59033203125 0.943885434259215339025672619754914194345474243164062 5]
 [-0.109475379439611794432795196788157237262619711747455585643977087208422 7
897076001 -0.602860959433068298998521068328637255204741412487675879377324455
09973240806545034; 0.697109511364440389125194749645107203610905035313040096
563384383621358664785136 1 -0.185341989945357707942956308438773929324163499 0
383780896541251293470030714302575; 0.597712301516516254203117397235329095 97
610308182614781970120750464411790270602 4 0.713707315954829504256943856276 25
252460110755728834206755136109050275385747120 95; 1.753086356467687621177652
035248669731573301372224530519801352966171762145675398 1.858902022774533672
678377263979130347284006106982487468351691217238203476190328]
 [-3.168127984758429165986461412366076167158188260961459594885883510621151 1
56485822 -3.905468706017616325728160050797730388137349598467806814503508222
346954832963097; -1.005320313989724165206828506779090844872440842840266012 3
375535393708522711842 86 -1.718677194244875732493795975731538768836404578637
278280553917785027803868728047; 2.686906506229309093968246138682707400962 75
402239021008913765272323192832053862 5 3.451933031590201162890788522764059 33
581302042405747036839215494456605270285168 8; 2.167631388447368607098497 49 25
765533451080337231849993691425861668627666880446 96 2.173137264318964440378 7
011481425282302143825852376529872435313504059002552788 79]
 [-6.702422222447204240694915214855833990162912588848772774434514903498913 8
977568 -7.441488571279222715534373863363943122042437296621051128848209556 54
1196756870859; -0.707106731335415805288237946435033876426265605593005018401
517585462825986603523 3 -0.533678308227363702883460276843384183942483640165 4
888781669656782861184833598422; 9.318096030787488445898174951885153916109 31
541434020415337827517972134930227687 8 10.989535289866645846696722516746907 2
```

```
0124393670178230457008868714032626042482995; -0.17799126639814855866400719506760070006969754010264043276004098250432897500420 3 -0.5525132765709586877693640557524507413885270133229792629825364227430012409608405]
 [-4.84178297015785513231271398235015016484640932533837591584683945677609401933388 -4.776600298875033874842454971190332270511412886835898719826019906938576148400592; 7.176798264856006730210750842091930843383783401471413874606179745081683883722825 9.018796158379933211726505335315667638012028494612812993033014909839654942577931; 15.719175453801992606353696085865263324788116397458273938988166555679937633874 4 17.678561966601799775128534042852348032322385234553138890473707817316648123235 16; -5.542233864223447241719493808823041558431495736033390497043188677421122683135989 -6.52737957557276335314182887874035976634763330130493776694743782285919848743370 9]
```

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
 0.04934038230948589
 0.12746824795907116
 0.21529410162344464
 0.32617126195125423
 0.4496926996092382
 0.5930602222914083
 0.7388617109991696
 0.8917753647948521
 1.0
u: 10-element Array{StaticArrays.SArray{Tuple{4,2},Float64,2,8},1}:
 [0.6734795581781894 0.6565989943848607; 0.7811030205598837 0.8677097775708533; 0.8115899804553721 0.7564773165944856; 0.9438534530763114 0.30881168055270962]
 [0.4310161978371491 0.58740313851584; 0.8039721428069945 0.9782609605207374; 0.7552046259837844 0.6529008229281703; 1.2416056440376548 0.5010264463015848]
 [-0.12388743107529171 0.37449474489549395; 0.6406226239814128 1.00916989047958; 0.8155358013419143 0.549593576583781; 1.6534240749596099 0.7790890872644718]
 [-0.9768592731757202 -0.009067695895584371; 0.22884066039762652 0.8609746463963469; 1.1635627114743854 0.5612769376456377; 1.9934966226594695 1.0335557977428156]
 [-2.3261640585865315 -0.680228690258079; -0.4654025196614129 0.4801420739602837; 2.1205733178942108 0.8342839176296077; 2.1679000653430807 1.2289496536820244]
 [-4.0073096108969395 -1.590763302559515; -1.1334528586286141 -0.0255371128
```

```
85180624; 3.9408652012441077 1.545024661996175; 1.9258547455585528 1.225245
5702255296]
 [-5.789322020625512 -2.667682147031413; -1.1611742546129453 -0.38940633643
327444; 6.986392182348398 2.925471129166359; 0.9342523672404413 0.851801979
4347593]
 [-6.767099848328803 -3.461463156474695; 0.3963430835347792 -0.077636575282
33808; 10.717296084337033 4.805379396063686; -0.9555383758310105 0.00082258
19842855309]
 [-5.9837640567295605 -3.4995250216208924; 4.495355361933915 1.454913347030
9117; 14.378858072345118 6.887519698831289; -3.867930184596118 -1.414781024
3156858]
 [-3.7669605925806224 -2.741217621044546; 9.150922634081248 3.4687901334870
6; 15.995251126066652 8.045665642467664; -6.374409117672755 -2.693716831862
229]

sol[3]

4×2 StaticArrays.SArray{Tuple{4,2},Float64,2,8} with indices SOneTo(4)×SOne
To(2):
 -0.123887  0.374495
  0.640623  1.00917
  0.815536  0.549594
  1.65342   0.779089
```

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 5.4.0
[91a5bcdd-55d7-5caf-9e0b-520d859cae80] Plots 1.5.8
[90137ffa-7385-5640-81b9-e52037218182] StaticArrays 0.12.4
[c3572dad-4567-51f8-b174-8c6c989267f4] Sundials 4.2.5
[37e2e46d-f89d-539d-b4ee-838fcccc9c8e] LinearAlgebra
```