An Intro to DifferentialEquations.jl

Chris Rackauckas

December 29, 2019

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`.

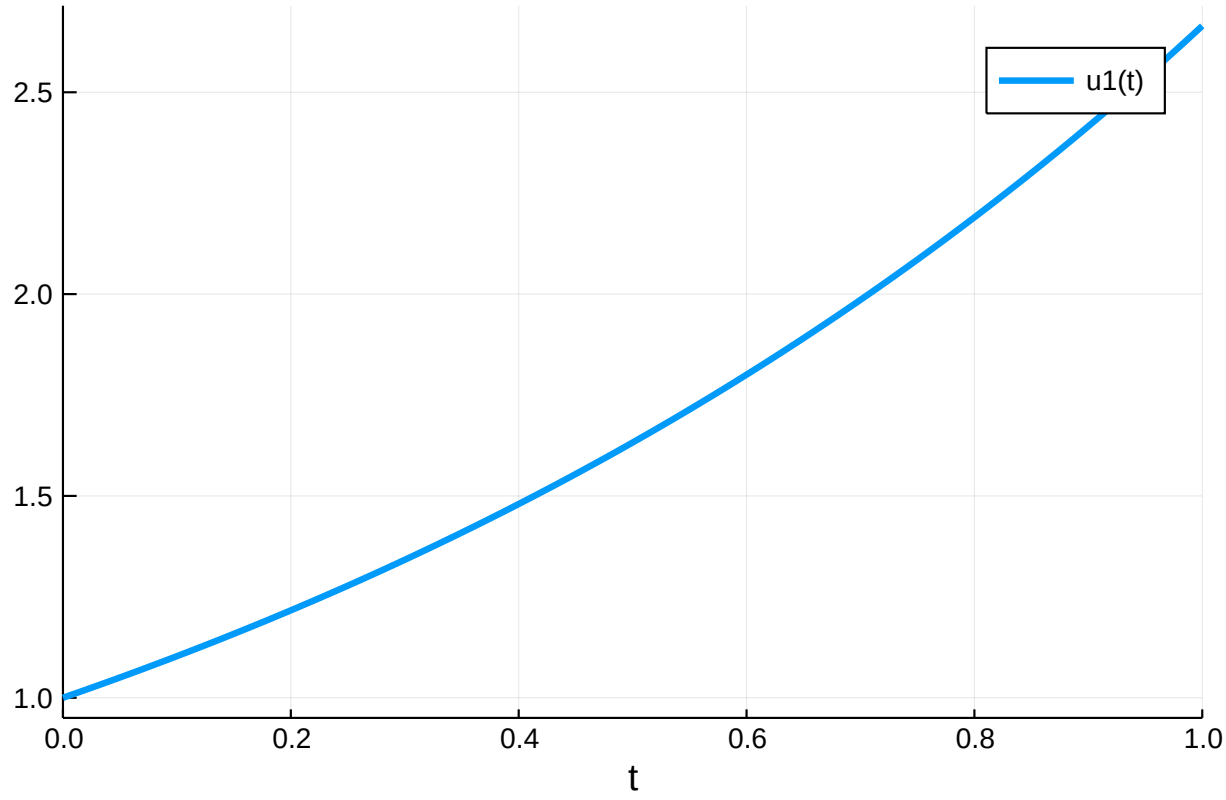
```
sol = solve(prob)
```

```
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
 1.9730384275623003
 2.664456142481452
```

and that's it: we have succesfully solved our first ODE!

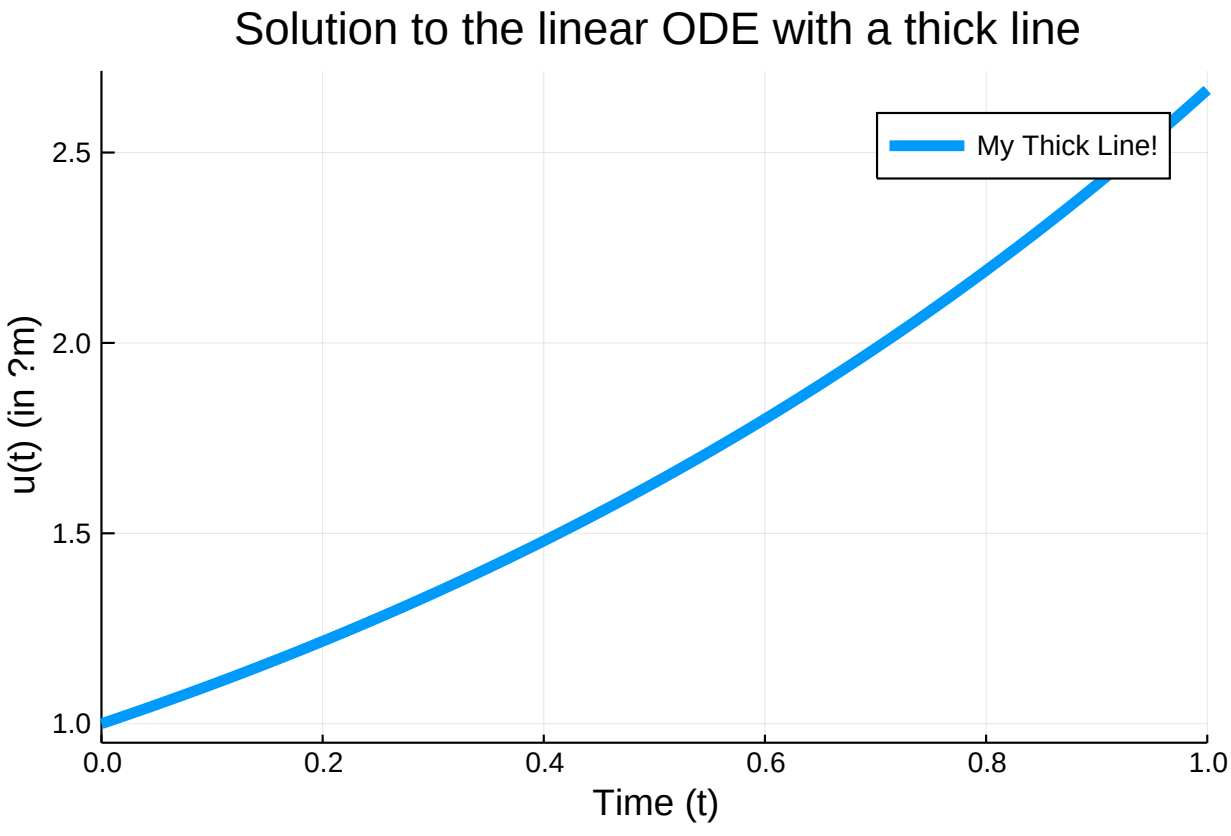
Analyzing the Solution Of course, the solution type is not interesting in and of itself. We want to understand the solution! The documentation page which explains in detail the functions for analyzing the solution is the Solution Handling page. Here we will describe some of the basics. You can plot the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

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



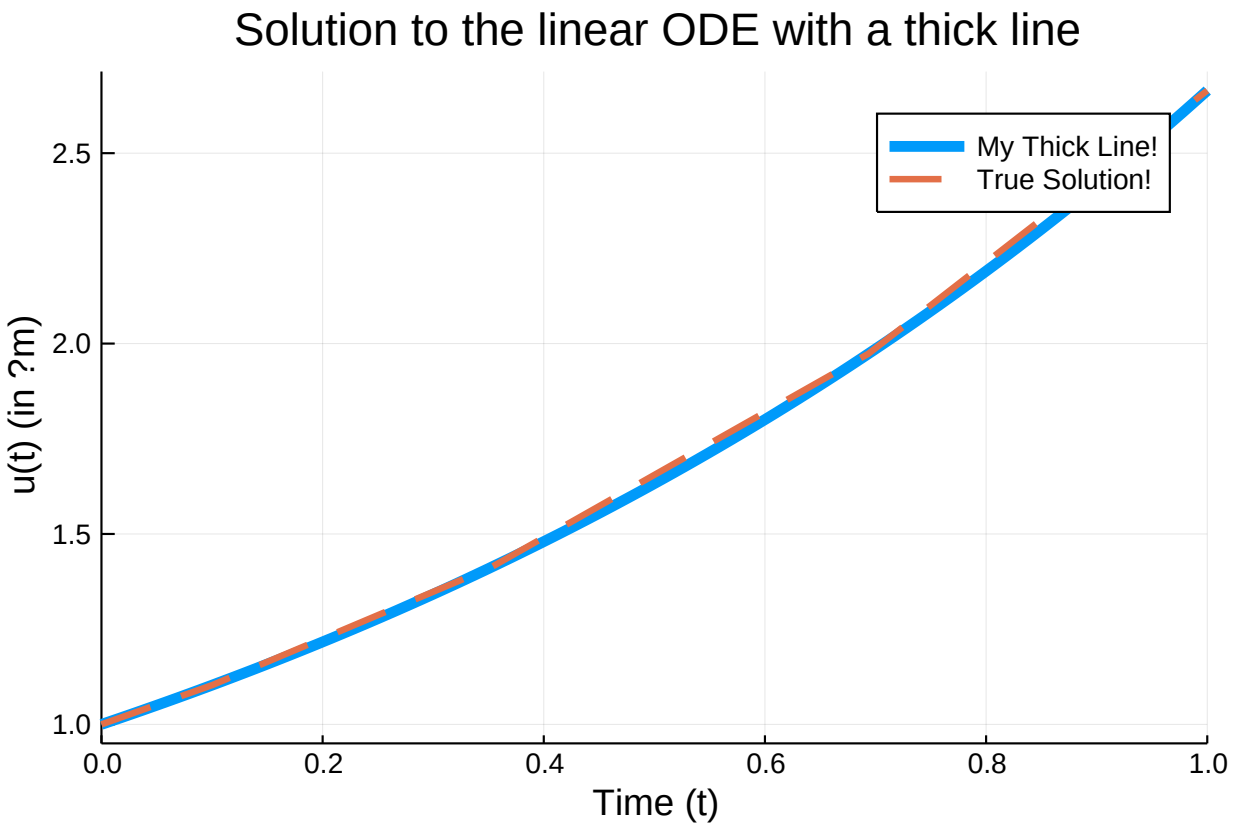
From the picture we see that the solution is an exponential curve, which matches our intuition. As a plot recipe, we can annotate the result using any of the Plots.jl attributes. For example:

```
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
 1.9730384275623003
 2.664456142481452
```

`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
 2.6664820303831105
 3.664456142481452
```

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
 1.9730384275623003
 2.664456142481452
```

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)
```

```
1.5542610480553116
```

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:

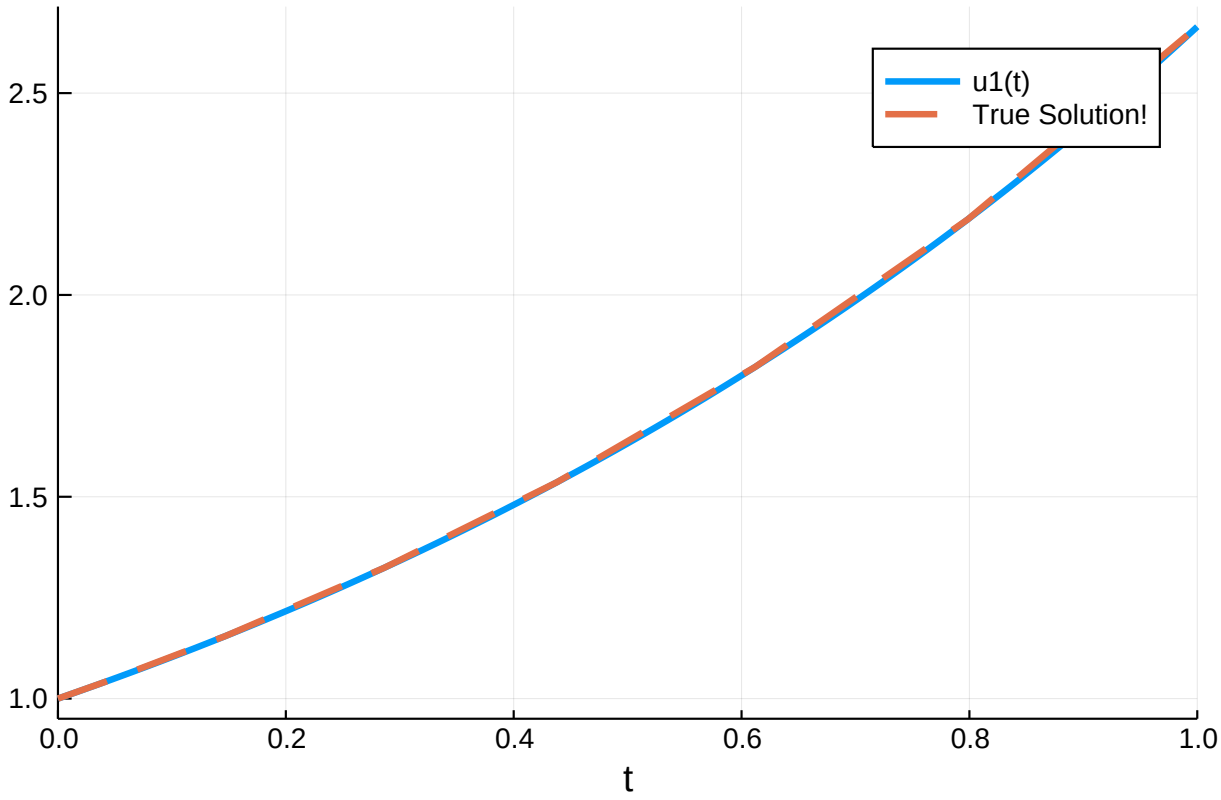
```
sol = solve(prob,abstol=1e-8,reltol=1e-8)
```

```
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
 1.1029627851292922
 1.2165269512238264
 1.341783821227542
 1.4799379510586077
 1.6323162070541606
 1.8003833264983586
 1.9857565541588764
 2.1902158127997704
 2.4157257420844966
 2.664456142481452
```

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
 1.9857565541588764
 2.4157257420844966
```

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
```

```
 1.9730384275623003
 2.664456142481452
```

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
 2.664456142481452
```

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}:
 2.664456142481452
```

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
 2.043449143475479
 2.6418256160577602
 2.6644526430553808
```

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
 2.0905717422346277
 2.4861021714470244
 2.6644562434913373
```

0.1.3 Systems of ODEs: The Lorenz Equation

Now let's move to a system of ODEs. The Lorenz equation is the famous "butterfly attractor" that spawned chaos theory. It is defined by the system of ODEs:

$$\frac{dx}{dt} = \sigma(y - x) \tag{1}$$
$$\frac{dy}{dt} = x(\rho - z) - y \tag{2}$$
$$\frac{dz}{dt} = xy - \beta z \tag{3}$$

To define a system of differential equations in DifferentialEquations.jl, we define our `f` as a vector function with a vector initial condition. Thus, for the vector `u = [x,y,z]'`, we have the derivative function:

```
function lorenz!(du,u,p,t)
    σ,ρ,β = p
    du[1] = σ*(u[2]-u[1])
    du[2] = u[1]*(ρ-u[3]) - u[2]
    du[3] = u[1]*u[2] - β*u[3]
end
```

```
lorenz! (generic function with 1 method)
```

Notice here we used the in-place format which writes the output to the preallocated vector `du`. For systems of equations the in-place format is faster. We use the initial condition $u_0 = [1.0, 0.0, 0.0]$ as follows:

```
u0 = [1.0,0.0,0.0]
```

```
3-element Array{Float64,1}:
 1.0
 0.0
 0.0
```

Lastly, for this model we made use of the parameters `p`. We need to set this value in the `ODEProblem` as well. For our model we want to solve using the parameters $\sigma = 10$, $\rho = 28$, and $\beta = 8/3$, and thus we build the parameter collection:

```
p = (10,28,8/3) # we could also make this an array, or any other type!
```

```
(10, 28, 2.6666666666666665)
```

Now we generate the `ODEProblem` type. In this case, since we have parameters, we add the parameter values to the end of the constructor call. Let's solve this on a time span of `t=0` to `t=100`:

```
tspan = (0.0,100.0)
prob = ODEProblem(lorenz!,u0,tspan,p)
```

```
ODEProblem with uType Array{Float64,1} and tType Float64. In-place: true
timespan: (0.0, 100.0)
u0: [1.0, 0.0, 0.0]
```

Now, just as before, we solve the problem:

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

```
retcode: Success
Interpolation: Automatic order switching interpolation
t: 1294-element Array{Float64,1}:
```

```
   0.0
   3.5678604836301404e-5
   0.0003924646531993154
   0.0032624077544510573
   0.009058075635317072
   0.01695646895607931
   0.0276899566248403
   0.041856345938267966
   0.06024040228733675
   0.08368539694547242
   ⋮
  99.39403070915297
  99.47001147494375
  99.54379656909015
  99.614651558349
  99.69093823148101
  99.78733023233721
  99.86114450046736
  99.96115759510786
 100.0
u: 1294-element Array{Array{Float64,1},1}:
 [1.0, 0.0, 0.0]
 [0.9996434557625105, 0.0009988049817849058, 1.781434788799208e-8]
 [0.9961045497425811, 0.010965399721242457, 2.146955365838907e-6]
 [0.9693591634199452, 0.08977060667778931, 0.0001438018342266937]
 [0.9242043615038835, 0.24228912482984957, 0.0010461623302512404]
 [0.8800455868998046, 0.43873645009348244, 0.0034242593451028745]
 [0.8483309877783048, 0.69156288756671, 0.008487623500490047]
 [0.8495036595681027, 1.0145425335433382, 0.01821208597613427]
 [0.9139069079152129, 1.4425597546855036, 0.03669381053327124]
 [1.0888636764765296, 2.052326153029042, 0.07402570506414284]
 ⋮
 [12.999157033749652, 14.10699925404482, 31.74244844521858]
 [11.646131422021162, 7.2855792145502845, 35.365000488215486]
 [7.777555445486692, 2.5166095828739574, 32.030953593541675]
 [4.739741627223412, 1.5919220588229062, 27.249779003951755]
 [3.2351668945618774, 2.3121727966182695, 22.724936101772805]
 [3.310411964698304, 4.28106626744641, 18.435441144016366]
 [4.527117863517627, 6.895878639772805, 16.58544600757436]
 [8.043672261487556, 12.711555298531689, 18.12537420595938]
 [9.97537965430362, 15.143884806010783, 21.00643286956427]
```

The same solution handling features apply to this case. Thus `sol.t` stores the time points and `sol.u` is an array storing the solution at the corresponding time points.

However, there are a few extra features which are good to know when dealing with systems of equations. First of all, `sol` also acts like an array. `sol[i]` returns the solution at the `i`th time point.

```
sol.t[10],sol[10]
```

```
(0.08368539694547242, [1.0888636764765296, 2.052326153029042, 0.07402570506
414284])
```

Additionally, the solution acts like a matrix where `sol[j,i]` is the value of the `j`th variable at time `i`:

```
sol[2,10]
```

```
2.052326153029042
```

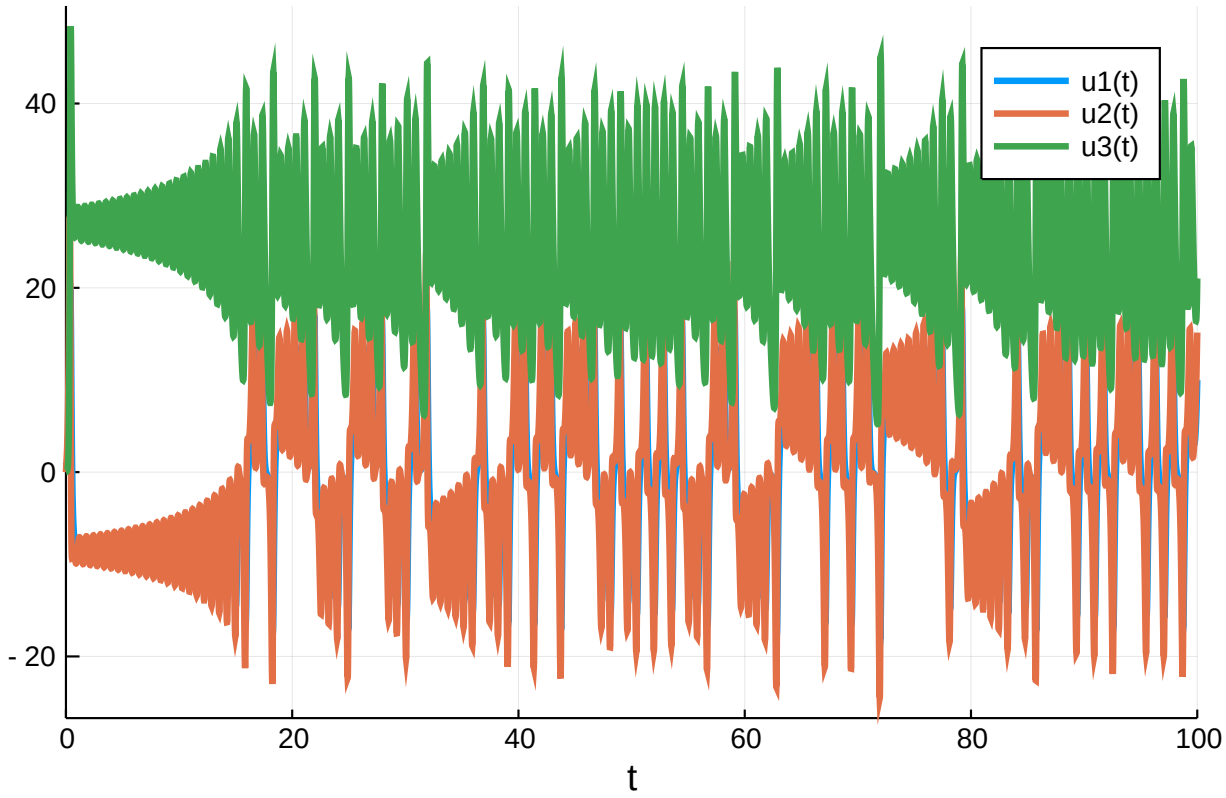
We can get a real matrix by performing a conversion:

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

```
3×1294 Array{Float64,2}:
 1.0  0.999643     0.996105    0.969359     …   4.52712   8.04367   9.97538
 0.0  0.000998805  0.0109654   0.0897706        6.89588  12.7116   15.1439
 0.0  1.78143e-8   2.14696e-6  0.000143802     16.5854   18.1254   21.0064
```

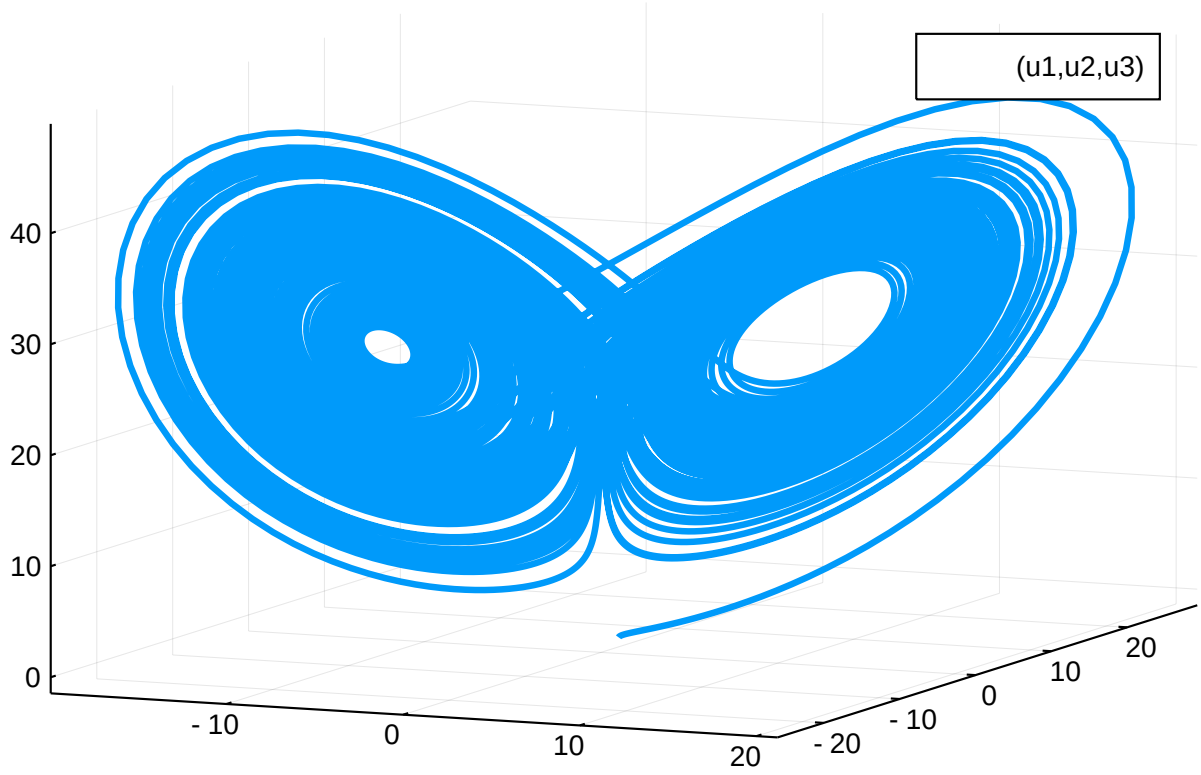
This is the same as sol, i.e. `sol[i,j] = A[i,j]`, but now it's a true matrix. Plotting will by default show the time series for each variable:

```
plot(sol)
```



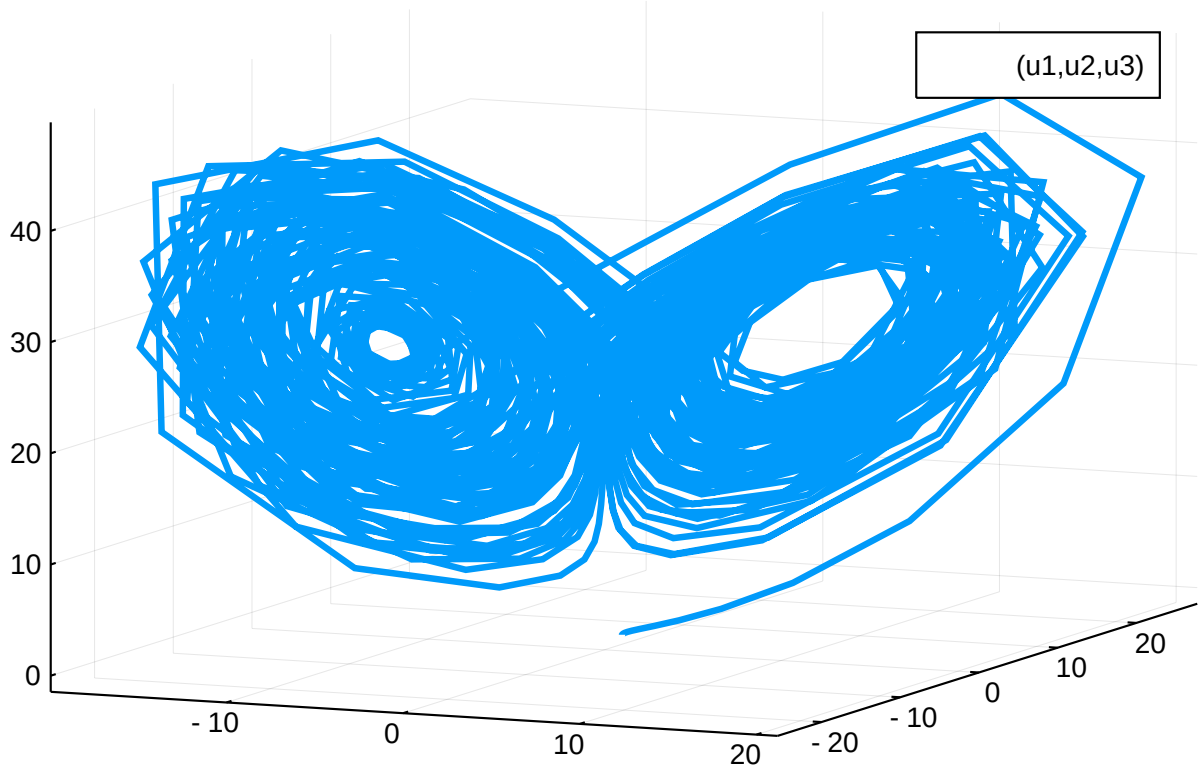
If we instead want to plot values against each other, we can use the `vars` command. Let's plot variable 1 against variable 2 against variable 3:

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



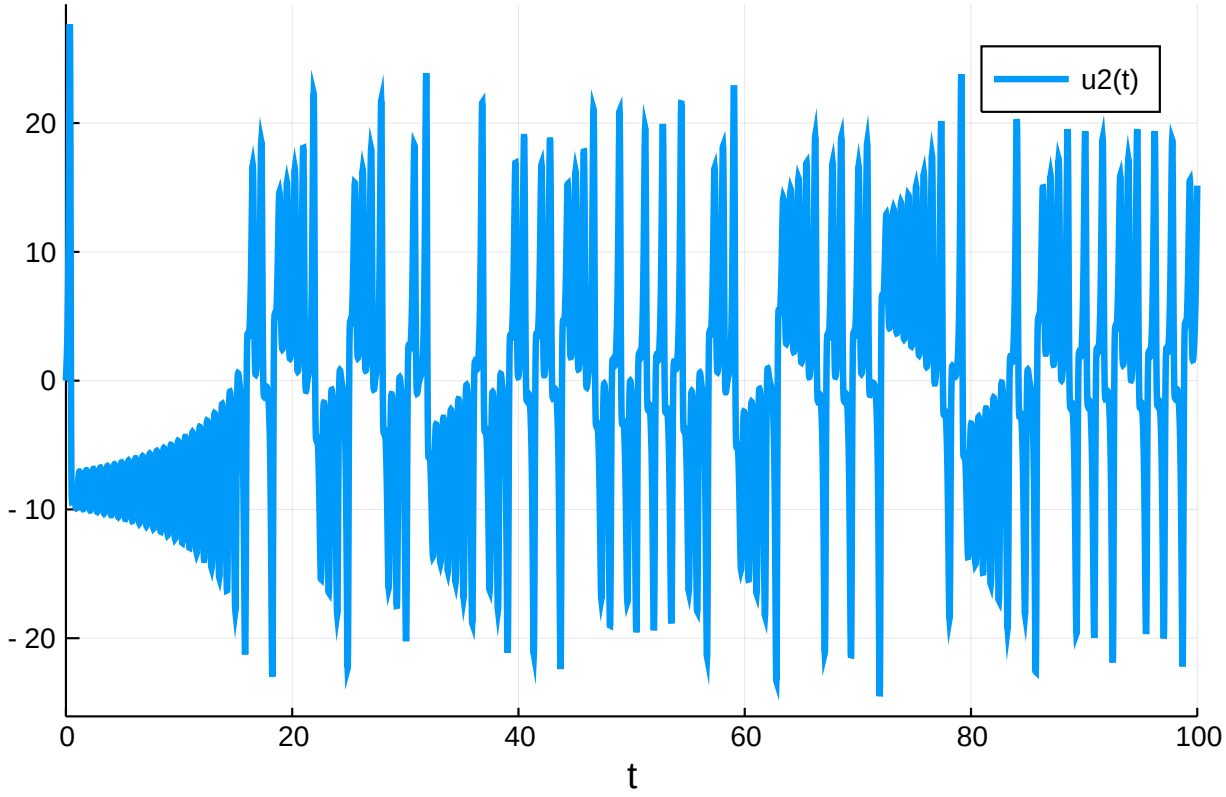
This is the classic Lorenz attractor plot, where the `x` axis is `u[1]`, the `y` axis is `u[2]`, and the `z` axis is `u[3]`. Note that the plot recipe by default uses the interpolation, but we can turn this off:

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



Yikes! This shows how calculating the continuous solution has saved a lot of computational effort by computing only a sparse solution and filling in the values! Note that in vars, `0=time`, and thus we can plot the time series of a single component like:

```
plot(sol,vars=(0,2))
```



0.2 Internal Types

The last basic user-interface feature to explore is the choice of types. DifferentialEquations.jl respects your input types to determine the internal types that are used. Thus since in the previous cases, when we used `Float64` values for the initial condition, this meant that the internal values would be solved using `Float64`. We made sure that time was specified via `Float64` values, meaning that time steps would utilize 64-bit floats as well. But, by simply changing these types we can change what is used internally.

As a quick example, let's say we want to solve an ODE defined by a matrix. To do this, we can simply use a matrix as input.

```
A  = [1. 0  0 -5
      4 -2  4 -3
     -4  0  0  1
      5 -2  2  3]
u0 = rand(4,2)
tspan = (0.0,1.0)
f(u,p,t) = A*u
prob = ODEProblem(f,u0,tspan)
sol = solve(prob)
```

```
retcode: Success
Interpolation: Automatic order switching interpolation
```

```
t: 11-element Array{Float64,1}:
 0.0
 0.01954429871008149
 0.0669099599652496
 0.1303998997112701
 0.2129545597458341
 0.3152505169763467
 0.4320759467693855
 0.5738398466397958
 0.7300593038072928
 0.8945315166323424
 1.0
u: 11-element Array{Array{Float64,2},1}:
 [0.22002942928397506 0.6130752662428487; 0.024190230392423517 0.3788458930
758998; 0.06984792307276888 0.6072202209982369; 0.6177319106714136 0.024958
244342418556]
 [0.16051367561083055 0.619254030688266; 0.005884207558136113 0.45287271320
863726; 0.06757724804589688 0.5601811631746226; 0.6763837030366282 0.095275
11147524088]
 [-0.012103825416873243 0.6057493581334595; -0.06842530160545174 0.58981492
12917788; 0.08814633259362886 0.4520264589162547; 0.8102263944755734 0.2647
4217534905553]
 [-0.30452009072714936 0.522664563572651; -0.22605479261020126 0.6800588932
264329; 0.18345677378820374 0.3310090002892933; 0.9658220556117337 0.485027
4187257579]
 [-0.7801373197149025 0.30205171942775116; -0.5027020506966361 0.6467210127
538839; 0.4460905829296234 0.24246303903892052; 1.1132941942805359 0.747191
0727885724]
 [-1.4881638312857906 -0.14073191159281623; -0.885885799612882 0.4059757084
5933606; 1.0250841951968275 0.2938655055870082; 1.18108837380321 1.00798018
52355856]
 [-2.378861988040944 -0.8420433738859363; -1.230980446703252 -0.04279209365
3845675; 2.0604060878129657 0.6455993809247602; 1.0559470405918345 1.175377
1462926803]
 [-3.3815724962477383 -1.8646816263502894; -1.2222951848919106 -0.606941794
3961943; 3.8224375216235313 1.5744376817963013; 0.5416819672563736 1.117896
8426643452]
 [-3.9996091092373875 -2.9578486096105205; -0.21674884580944997 -0.83908386
12776872; 6.174507088015794 3.2348746330975207; -0.5572636000370653 0.62742
94198593589]
 [-3.4939973342545096 -3.622248411981576; 2.460859310508148 -0.079587217369
98964; 8.49485915337516 5.464515082313784; -2.3178429101412545 -0.453216343
99322264]
 [-2.222033450382404 -3.510393230763207; 5.1828242940264335 1.1980674478390
618; 9.414797551757655 6.889462532282994; -3.7029970797479663 -1.4533269651
207363]
```

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}:
 -0.0121038  0.605749
 -0.0684253  0.589815
  0.0881463  0.452026
  0.810226   0.264742
```

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}:
 0.220029   0.613075
 0.0241902  0.378846
 0.0698479  0.60722
 0.617732   0.0249582
```

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
 0.019544298710081485
 0.15320973766025
 0.40414263840398756
 0.7130331147065483
 1.0
u: 6-element Array{Array{BigFloat,2},1}:
 [0.2200294292839750642798435364966280758380889892578125 0.6130752662428486
9321002133801812306046485900878906 25; 0.0241902303924235173582246716250665 4
8585891723632812 5 0.3788458930758997755816608332679606974124908447265625; 0
.0698479230727688804591934967902489006519317626953125 0.6072202209982369147
66721383784897625446319580078125; 0.6177319106714136420777094826917164027 69
0887451171875 0.024958244342418556271923080203123390674591064453125]
```

```
 [0.160513675610498737412894576884650983494522030465195084519339212452721 80
71454121 0.6192540306882626946889412960426704456502052257544473735472165533
300799712647532; 0.0058842075619023098010493817115503276669438888086297038 0
994215310162034544838827 5 0.4528727132125269606473394202908179725166047731 9
888996984205266864530272181261 08; 0.0675772480488214784220555788403821917 30
6124975393256596008091493222230359407874 9 0.5601811631768398030200279529634
0970441025018383497695052265486515346323526846 08; 0.676383703036180701704 87
7304744887180375255171328249036532959359578804577786991 4 0.095275111475604 7
4291681091522069998435157836581697062188316213595696878310597405]
 [-0.4257483307170469832423271609954027265239306488789935627424142108361402
614726985 0.4743931802726334599597455692404697629059713081030764289533840 10
303811398505307; -0.2959783140315804841937719947173562418224448311826911078
36773611445240521446064 0.6872055917668520205136685836700836390158387269050
149527629008805237135323955434; 0.2392920584680226382424102979102241909363 4
8595460741928688101427956821425634293 1 0.2973816832319545466561281658010690
006846136000260350963907168286363560405254579; 1.013487034522756800898572 73
1786345029027269712737157409339444994292689063241801 0.5608473687135851816 0
4808939743518804396584952868795630585578427415680528571708 5]
 [-2.1640996578653220451762791059557629165806385913405399357022497284225837
91763218 -0.6585453825805009849014932419666637298618895653410294844782585 53
9346419411573922; -1.1675911557466083027901586443687528061894210208106229 50
9970660732570054535227 21 0.0739390648326732487749199965962580567258633791 45
892300252513281675886856534132 89; 1.776332711680226554083021488264955015754
778605058139199852661671349375055291 76 0.5293347574008875977408015742572534
74310316910049976854160534373235800673582041 5; 1.108285410677809916925353 67
```

```
9344131743579781762248275704294072848324745802700105 1.15069562794831493591
06973740801132181909222159124403567701732717976851409616231
 [-3.97282072904148923181880857154449850757312362536631027414911575021282090
3605795 -2.85161476756805320594439935745851831169665264989685896875481569 6
658183157873409; -0.39181523564450948924137724841413544894935378281566920576
61219467890597265015191 9 -0.84909106540731083153105871058029561139651769203
29588573522057836459854914437345; 5.91118612071969666999354370422047109220740
07602872376849853472200601511615235543.025672530259098439319455700859720707
74075524511563553326996531763372275504337215; -0.40935963552194140424146399 9
98329450452571239074550759742609983114770127577321430.705331087841190174656
06496962204727995026348626736908699247346735604351657081784]
 [-2.22202571573738401970510987747504655951663419048246814885097685785364196
6425964 -3.51039481949895814787040245577261741894065308313573712247101474623
3586110072788; 5.18284999462616436479136978561079380232001075568841609656266
74197315223990621924 1.19808007400140505295301050984709713305989291282476280
58102183008520364336070820; 9.41481173391616120680115462933071151994293938 6
17551966060845966306184991146559 5.889478135630163653955067623022614451335
44418201256828768856272690426057448850 3; -3.703010142561749993460600827427 7
73457638097103561649711941248634674747914394809 -1.45333492780085158632137 6
38383171693575658382366330426992737567896232516768116 9]

sol[1,3]

-0.42574833071704698324232716099540272652393064887899356274241421083614026 1
4726985
```

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

 0.019544298710081482825839963845518945407573169091654137896495306215760772 3
5447418
 0.153209737660249974245069719091077301438903092402689325754989406084552058 3
3709907
 0.404142638403987425411009820687826744986540476827803729877758563482075013 3
3299068
 0.713033114706548029723455559233734459785140518809117995258579123904236026 4
4780281
 1.0

u: 6-element Array{Array{BigFloat,2},1}:
 [0.220029429283975064279843536496628075838088989257812 5 0.6130752662428486 9
32100213380181230604648590087890625; 0.024190230392423517358224671625065 6 5
8585891723632812 5 0.378845893075899775581660833267960697412490844726562 5; 0
.069847923072768880459193496790248900651931762695312 5 0.607220220998236914 7
667213837848976254463195800781 25; 0.617731910671413642077709482691716402769
0887451171875 0.024958244342418556271923080203123390674591064453125]
```

```
 [0.160513675610498743804811426547366502418172690558475186532513597130944758299181 6
.619254030688262694405441591259340149847185632756354161373654261840124425185895; 0
.005884207561902312040331739012463396903735020040302651918865064097328148967295954 0
.452872713212526953650707002566666743210785845254162646992868937278134137435789 8; 0
.067577248048821478353939918667934150172947991793536358823449036182923529245636 19 0
.560181163176839807745882435614285616829653373385217890854741375821106033071508 5; 0
.676383703036180695841351913129607791192866207610759469238310762649763454678732 6 0
.095275111475604735780208203090132225949315678414023522855679508440902362465714 1]
 [-0.425748330717046789397710441436805814796512132518040117374630677725142952855362
3 0.474393180272633542175759589596494476956193060139845350418606693889913364207134
5; -0.295978314031580371471597375636575219707337131323328643400355325421720485176610
3 0.687205591766852019449864379362868356317888685514512120580982618609982477624368
1; 0.239292058468022542382679160280570961938805535659297909476632482390962557502093 0
.297381683231954593826774008001702522575955222189815229199210490457671026134216 5; 1
.013487034522756730948157367293731769833782005244444590244179143499555556678 35 0
.560847368713585066040852621113232717896487913863506399470075089415004729755964 6]
 [-2.164099657865321163559037001073689152190208345963651050589298983111097687603384
 -0.658545382580500251278638063299262970617631104981030247003123122230198909266610
7; -1.167591155746608012094697401298024405253035663283277319050013782089561529421828
 0.073939064832673719793786011866933569081585528937355775690893864766886220956419
46; 1.776332711680225436870119581899413220419630863168837358700581849999165366119082
 0.529334757400887164699282473485278059747297494092148518938618186046938438340835
3; 1.108285410677810100878933467685791613761107131144709248131021003784260675985527
 1.1506956279483148134708025747018422704782710490452952701338013293682698925896366]
 [-3.972820729041488781644475928440292685823086222496198059518451194973398067763265
 -2.851614767568051715134120798516898892929936776837503650643116158101792963066355;
 -0.391815235644511771681336270637009115775131787326786427368470486194129253804168
2 -0.849091065407310896607012943500401296829935663506630402127355531989407526800344
6; 5.911186120719693051361824182885240347988465910284591575990021402134745900685395
 3.025672530259095608397997384947776211758441206040891403002271547528518358360786;
 -0.409359635521939420743656184344312779747507554997221611420691670016458937256845
 0.705331087841191201334071024933590073783808333236281924344415493722026141182077]
 [-2.222025715737383624010622486252890926948304319669290351027587144314152275634194
 -3.510394819498958056644914730969435406825133855087676356526831663115900925789333;
 5.182849994626165081583678322998608962691808826924331014601933637930111358119296
 1.198080074001405428906589792658079750845240337524326850848185639953761938834204;
 9.414811733916161332726990808110157265091010048498861455789831735715460117859638
 6.889478135630163959674789197948907234471411590375553154121194055514859796013183;
 -3.703010142561750327522015598021901029367109907272571568574026405450044228689412
 -1.4533349278008518420344588293756622952593079291987521446673794468481080546417 24]
```

Let's end by showing a more complicated use of types. For small arrays, it's usually faster to do operations on static arrays via the package StaticArrays.jl. The syntax is similar to that of normal arrays, but for these special arrays we utilize the `@SMatrix` macro to indicate we want to create a static array.

```
using StaticArrays
A  = @SMatrix [ 1.0  0.0 0.0 -5.0
                4.0 -2.0 4.0 -3.0
               -4.0  0.0 0.0  1.0
                5.0 -2.0 2.0  3.0]
u0 = @SMatrix rand(4,2)
tspan = (0.0,1.0)
f(u,p,t) = A*u
```

```
prob = ODEProblem(f,u0,tspan)
sol = solve(prob)
```

```
retcode: Success
Interpolation: Automatic order switching interpolation
t: 11-element Array{Float64,1}:
 0.0
 0.013622831418861838
 0.06805563950576322
 0.15173271657248188
 0.2452025015020835
 0.3564241015740898
 0.4848342119655686
 0.6275722341140875
 0.7796619238189032
 0.9544102307445974
 1.0
u: 11-element Array{StaticArrays.SArray{Tuple{4,2},Float64,2,8},1}:
 [0.7899273534028666 0.39306208050778024; 0.311513537941426 0.8264716352474
373; 0.7641032065329312 0.9870488250797518; 0.022998577064575088 0.40573103
192668025]
 [0.7968885107389193 0.3690067456091299; 0.3834509006626953 0.8600837509832
548; 0.7216194993394363 0.9721196855080977; 0.09013381794506845 0.452953104
67106487]
 [0.7791011248989712 0.23795722763719734; 0.6043074854559101 0.946347015721
9003; 0.5609116027118417 0.9346022393791614; 0.3577917621076735 0.629928731
882589]
 [0.6045643557809823 -0.06688831546673446; 0.7383526170417386 0.94486042647
84006; 0.3710428160413029 0.9652028710899865; 0.7547095033268354 0.85455532
6696573]
 [0.19629561605006252 -0.5350495070281014; 0.616064250727633 0.798430566086
9488; 0.3038853675154095 1.1622231081804806; 1.1483653176382527 1.014176456
5986748]
 [-0.5645148541318915 -1.2119302615771539; 0.17599752421450537 0.5334589084
808327; 0.5240287443532461 1.6635766715903304; 1.496834658792578 1.04094264
05661544]
 [-1.7377955617037935 -2.0285151655771445; -0.5328946133602135 0.3135697605
9192376; 1.3100996553099855 2.619470384489738; 1.6544846383501055 0.8004651
006953425]
 [-3.21296399030658 -2.73426826240308; -1.1867848859471297 0.51171426901333
76; 2.94617939304878 4.068976184939837; 1.4061502280942615 0.14077857193818
633]
 [-4.589951913844382 -2.8622311412248957; -1.1362411431662625 1.64414910686
67151; 5.498462511814723 5.753720824163287; 0.5245354039901525 -1.030141462
5084422]
 [-5.164794204558272 -1.60768122960845; 0.7251371476391688 4.51211800391205
2; 8.93809463294772 7.0878764287728515; -1.3694049534809007 -2.864013002711
3084]
 [-5.01278979660893 -0.9537008553849472; 1.6283291269878206 5.5494318282594
355; 9.791505142267637 7.1811196922799345; -2.0146103542264653 -3.392792039
147243]
```

```
sol[3]
```

```
4×2 StaticArrays.SArray{Tuple{4,2},Float64,2,8} with indices SOneTo(4)×SOne
To(2):
 0.779101  0.237957
 0.604307  0.946347
 0.560912  0.934602
 0.357792  0.629929
```

0.3 Conclusion

These are the basic controls in DifferentialEquations.jl. All equations are defined via a problem type, and the `solve` command is used with an algorithm choice (or the default) to get a solution. Every solution acts the same, like an array `sol[i]` with `sol.t[i]`, and also like a continuous function `sol(t)` with a nice plot command `plot(sol)`. The Common Solver Options can be used to control the solver for any equation type. Lastly, the types used in the numerical solving are determined by the input types, and this can be used to solve with arbitrary precision and add additional optimizations (this can be used to solve via GPUs for example!). While this was shown on ODEs, these techniques generalize to other types of equations as well.

0.4 Appendix

This tutorial is part of the DiffEqTutorials.jl repository, found at: https://github.com/JuliaDiffEq/DiffEq

To locally run this tutorial, do the following commands:

```
using DiffEqTutorials
DiffEqTutorials.weave_file("introduction","01-ode_introduction.jmd")
```

Computer Information:

```
Julia Version 1.3.0
Commit 46ce4d7933 (2019-11-26 06:09 UTC)
Platform Info:
  OS: Windows (x86_64-w64-mingw32)
  CPU: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-6.0.1 (ORCJIT, skylake)
Environment:
  JULIA_EDITOR = "C:\Users\accou\AppData\Local\atom\app-1.42.0\atom.exe"  -a
  JULIA_NUM_THREADS = 4
```

Package Information:

```
Status `~\.julia\dev\DiffEqTutorials\Project.toml`
```