An Intro to DifferentialEquations.jl

Chris Rackauckas

October 7, 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`.

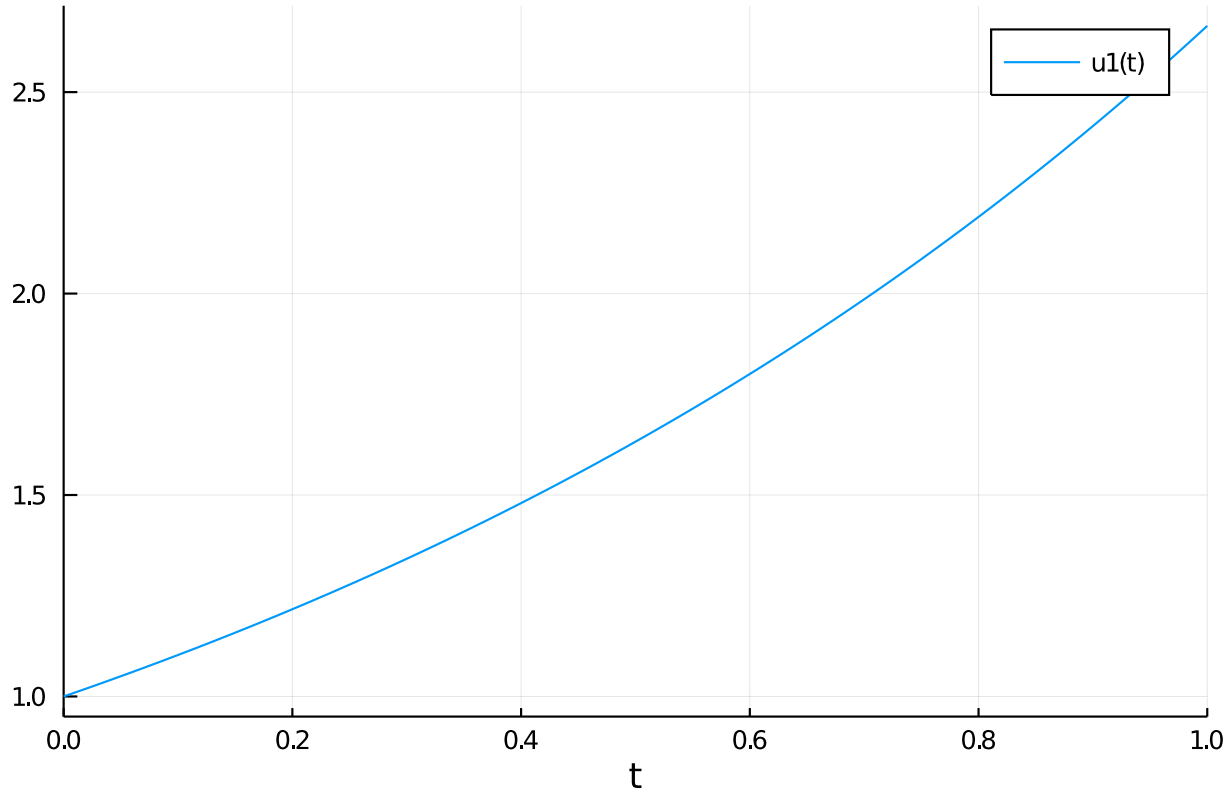
```
sol = solve(prob)
```

```
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!

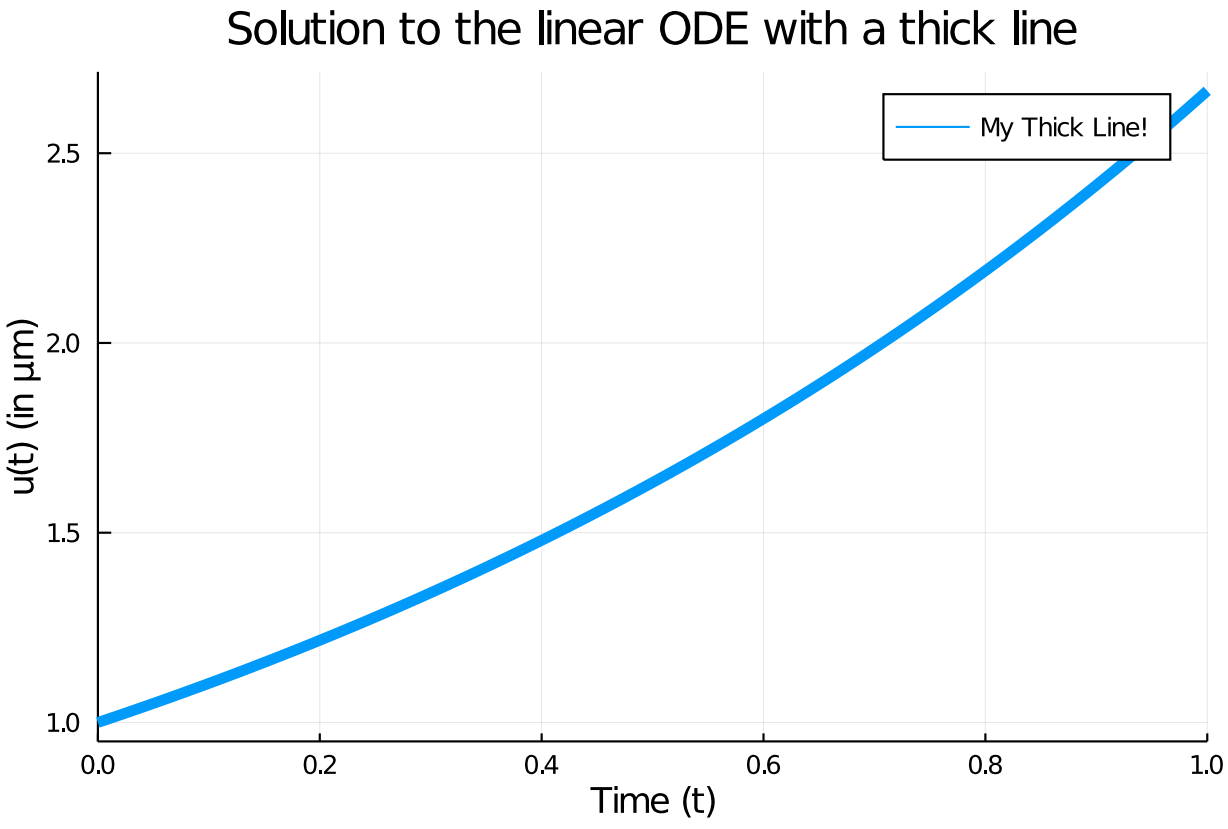
Analyzing the Solution Of course, the solution type is not interesting in and of itself. We want to understand the solution! The documentation page which explains in detail the functions for analyzing the solution is the Solution Handling page. Here we will describe some of the basics. You can plot the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

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



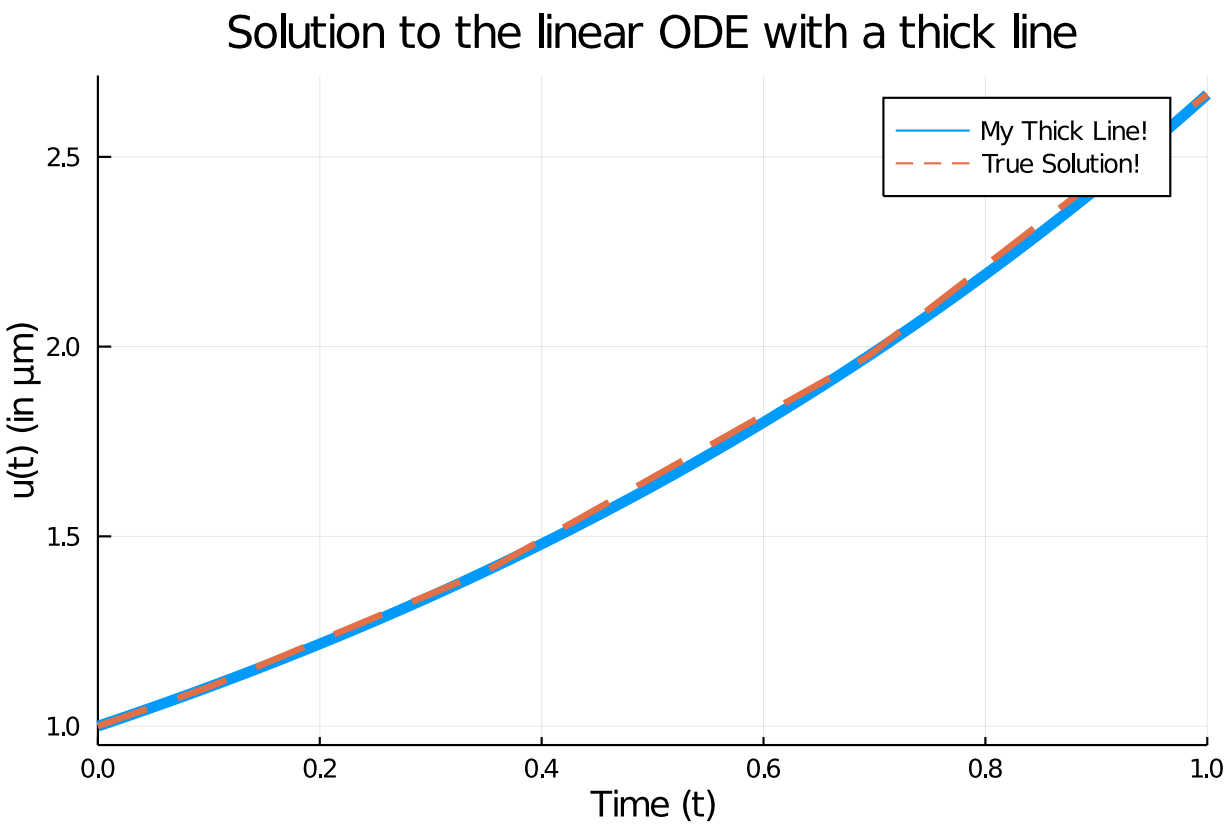
From the picture we see that the solution is an exponential curve, which matches our intuition. As a plot recipe, we can annotate the result using any of the Plots.jl attributes. For example:

```
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:

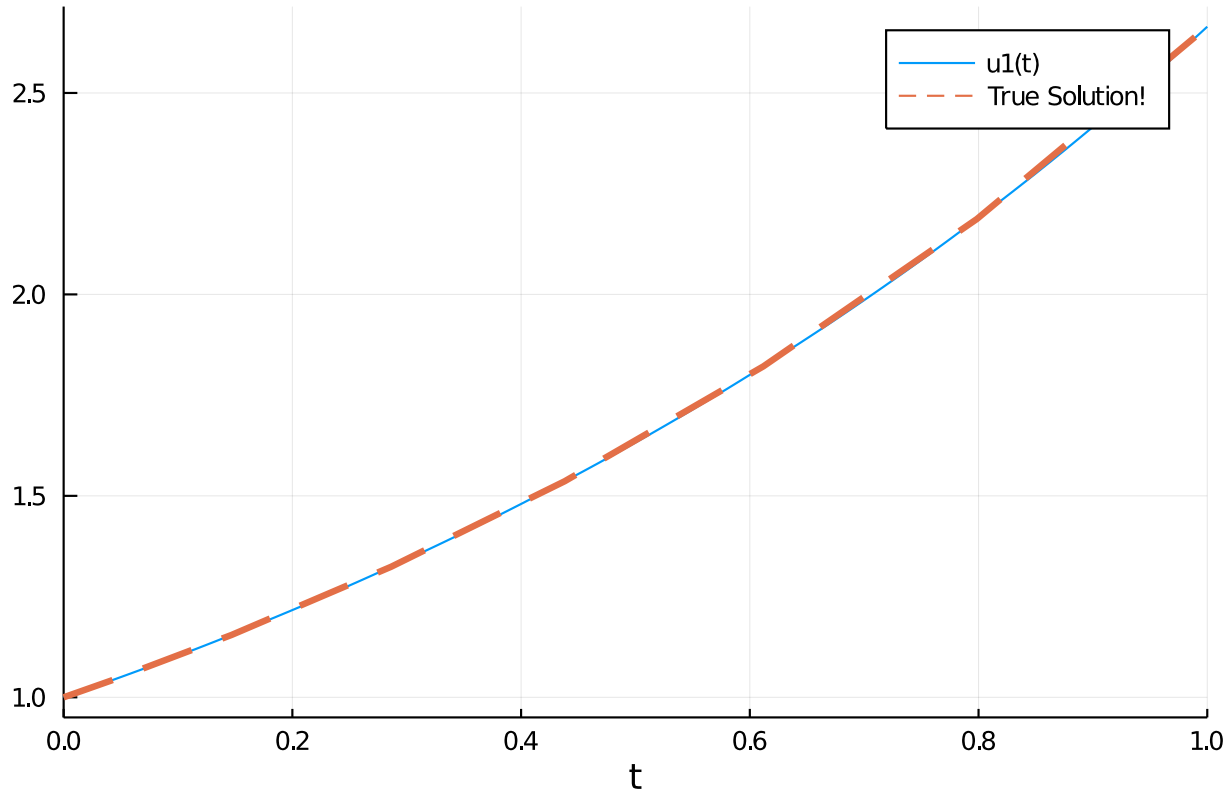
```
sol = solve(prob,abstol=1e-8,reltol=1e-8)
```

```
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

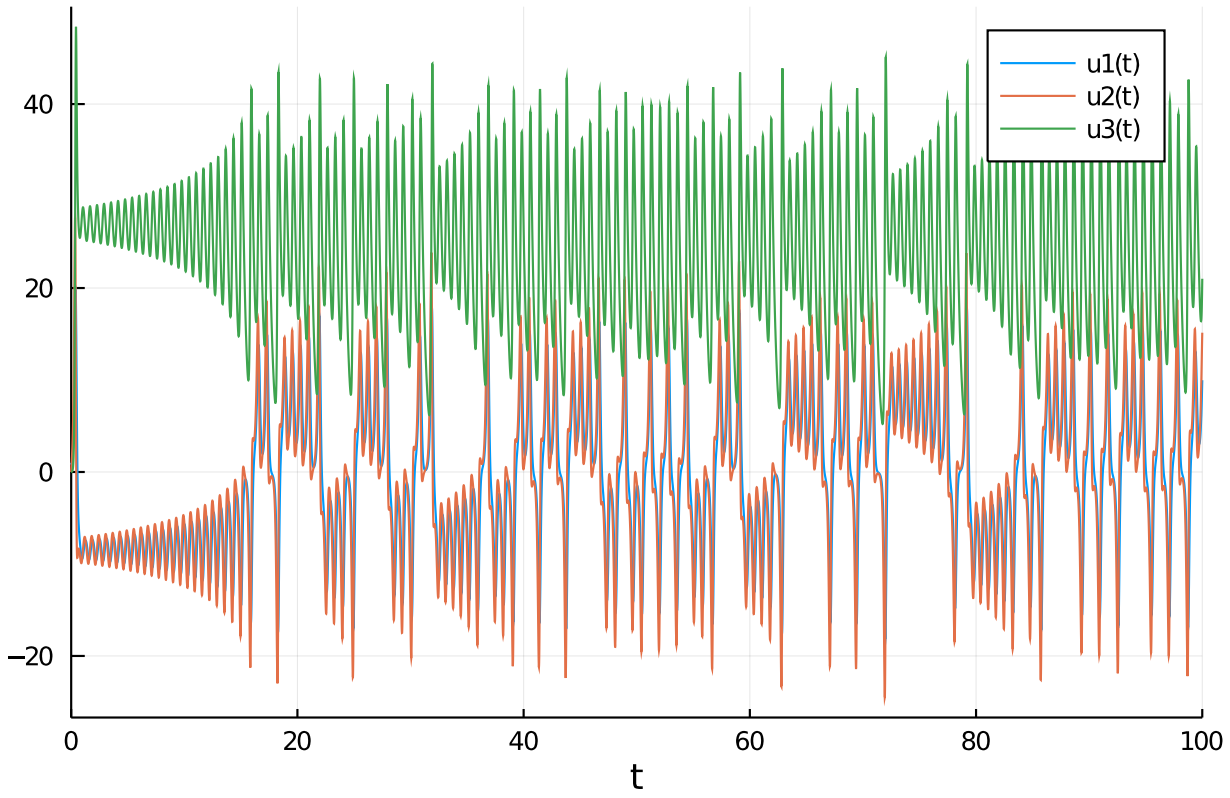
We can get a real matrix by performing a conversion:

```
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
```

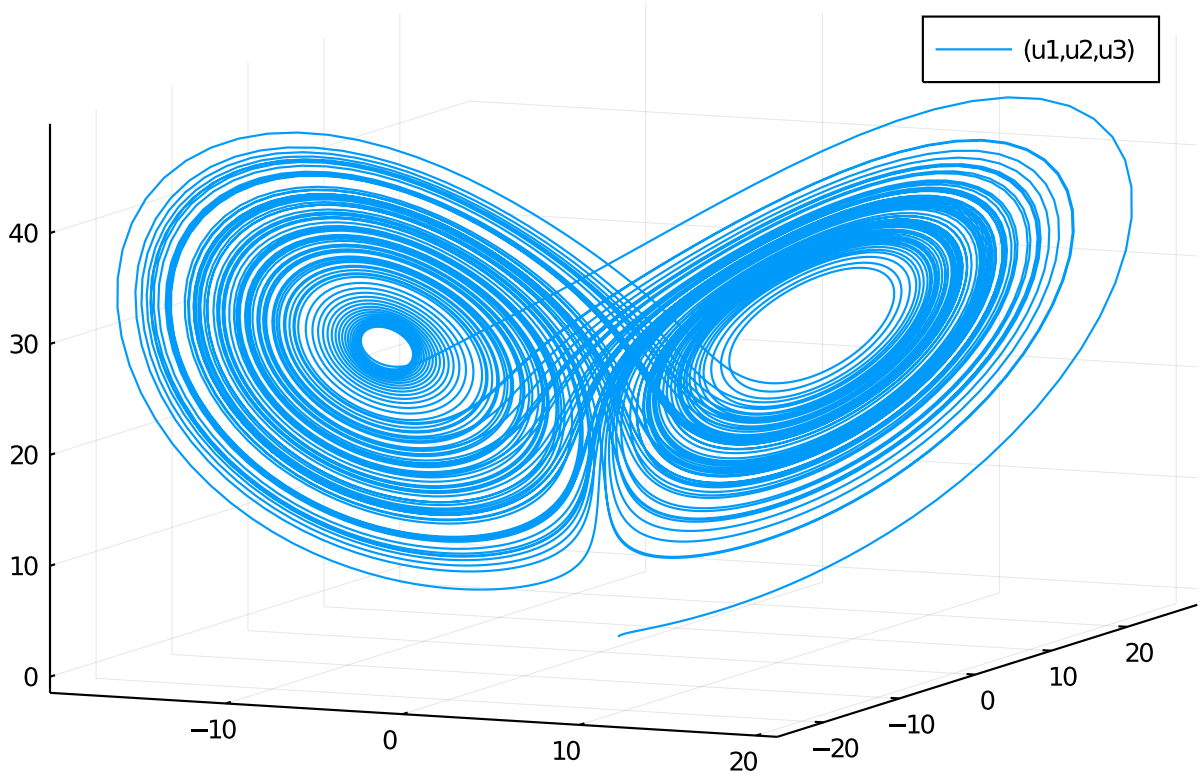
This is the same as sol, i.e. `sol[i,j] = A[i,j]`, but now it's a true matrix. Plotting will by default show the time series for each variable:

```
plot(sol)
```



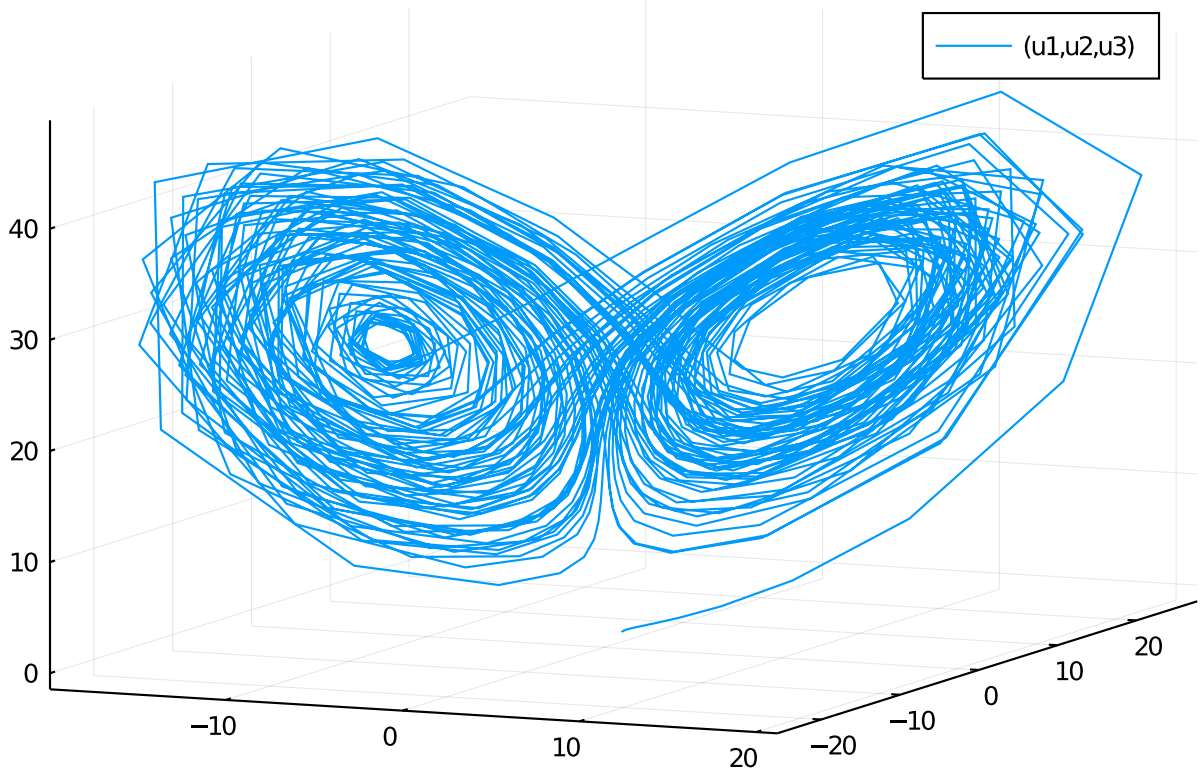
If we instead want to plot values against each other, we can use the `vars` command. Let's plot variable 1 against variable 2 against variable 3:

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



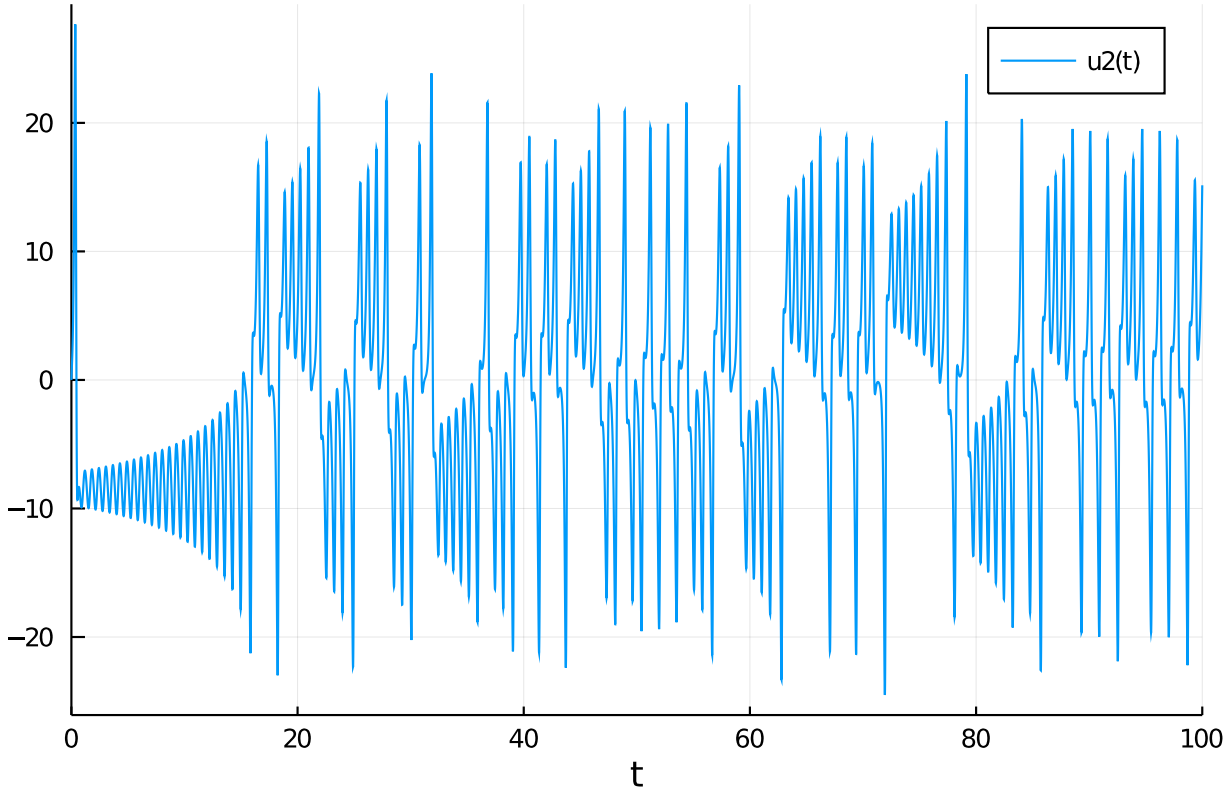
This is the classic Lorenz attractor plot, where the `x` axis is `u[1]`, the `y` axis is `u[2]`, and the `z` axis is `u[3]`. Note that the plot recipe by default uses the interpolation, but we can turn this off:

```
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: 10-element Array{Float64,1}:
 0.0
 0.05146877813401059
 0.1282527639612372
 0.20731579115404317
 0.31194657697845707
 0.4225431598147541
 0.5523447290123698
 0.6929130710324516
 0.8390639277681302
 1.0
u: 10-element Array{Array{Float64,2},1}:
 [0.9338218261404168 0.25027683696626934; 0.6880113011089077 0.506902953589
7485; 0.28992751685776486 0.29603372052070287; 0.2189820604968442 0.2920193
8310634223]
 [0.8948426636137807 0.17525069693742912; 0.7903064122469458 0.504947202325
4512; 0.11788938369023497 0.2690130051528156; 0.45134812084245574 0.3755129
5336193646]
 [0.7193322867237658 0.017848328300158783; 0.7787283709167452 0.44462855940
05449; -0.08596618899107833 0.27104811527568967; 0.7872952645502843 0.48037
868353315794]
 [0.38913424909065064 -0.19495121518957492; 0.5763782165401627 0.3271187297
261934; -0.19022504229055434 0.3390385836921268; 1.1033190209303758 0.55623
67883714682]
 [-0.272987172852885 -0.5367162625860921; 0.06637250786299154 0.12676175904
182013; -0.08904908575802231 0.5510289887642261; 1.4406242592723904 0.59227
6076515206]
 [-1.2152623883063445 -0.9339362563237532; -0.6552725287351174 -0.069600095
88756283; 0.4044789366199196 0.9391212647236922; 1.6463184935695707 0.53220
70391817578]
 [-2.5321196966566832 -1.3667942147334118; -1.4967473770474502 -0.156239524
16703127; 1.5859245470209573 1.5967175318885862; 1.6099302942619835 0.30950
920955914035]
 [-3.984027621234267 -1.656187624294827; -1.9947474758819008 0.095607008001
0418; 3.622513877511197 2.474070467842447; 1.128746260031665 -0.13564013502
98888]
 [-5.123308606501874 -1.5618097576166066; -1.5152890998010435 0.91571474211
66967; 6.408345924598824 3.3725094942512968; 0.04087731885134804 -0.8180666
081388397]
 [-5.284369363948497 -0.7291297770767666; 0.8351113144423721 2.598699526403
812; 9.701259712382853 3.952168390619043; -1.9011949761480538 -1.7635030272
350947]
```

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}:
  0.719332   0.0178483
  0.778728   0.444629
 -0.0859662  0.271048
  0.787295   0.480379
```

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}:
 0.933822  0.250277
 0.688011  0.506903
 0.289928  0.296034
 0.218982  0.292019
```

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
 0.11057493353303068
 0.3511651471634287
 0.6079979861780698
 0.9128674287985695
 1.0
u: 6-element Array{Array{BigFloat,2},1}:
 [0.9338218261404167552797161988564766943454742431640625 0.2502768369662693
448418622210738249123096466064453125; 0.688011301108907735724073972960468381
6432952880859375 0.5069029535897484617379404880921356379985809326171875; 0
.2899275168577648553736025860416702926158905029296875 0.2960337205207028699
0499425883172079920768737792968 75; 0.2189820604968442108173576343688182532 7
87322998046875 0.2920193831063422340577062641386874020099639892578125]
 [0.7723987385574132166160647364893176596192743432168472953765721130037857 8
15566487 0.0586802681616850825750892044486655011067660232695755494823714494
9459332515385472; 0.7983190269569502508789002441004863061702450516880161534
7191978671332249471068 07 0.4639352370125701599198049507673690931339434657 87
669634111881072599730349611154 6; -0.0464327914041657077486230954001096351 69
832368529243050168511963776515923196385 98 0.2654669516080728216362141912434
2908419784932291155049681684662540597825956832 44; 0.7118183666360206751921 4
677261092254495775570402531064732783230224380696188787 04 0.4586651033315070
868834186488585127140115800036273206608866245665779243815812106]
 [-0.5816638934039468505727490361537373012184160268059692478741971354608 0
000067551 -0.6759798342587180674160243454333979875717487584730685136722084 0
4362071844060415; -0.1761491922330949961233127569174458007599952069366878 42
937933529881874904665089 4 0.0508801665151275486102510162880049608412610418 1
938104016200949981501608398574898; 0.0359923745982492681720577192579822676 7
495606097667100055714329209569163091010883 0.6691996437719099333702839368 56
377051576156127779509569461604366451991164198583 7; 1.5342850506967671243842
095577148890056449700121650800134489547468681757380819 2 0.5835063469846612
714405850427245194743221771964413515699264024033364016453801904]
 [-3.121128418917642856024436443869158261408131217997108295082699225868431 2
94454187 -1.513071111332936026161449734212645012165404245427658564772037987
709652991099251; -1.771994688621019248902283305287299965392805340952177372 9
032707871602179106280 63 -0.109425676668356804426032007995659395714133588848
414646641633141925799945529 0733; 2.301566607287679308553932597095763271798 2
250195239755235556979589801632672357 34 1.931087922124021706213502848485168 2
909517593054429217452347540283623735210675 34; 1.480173141110081203763672651
910420525085594523636332762759930544663512546022 4 0.159100136095995456908 96
219089268120319163933792651995861346932924602994751428 73]
 [-5.386303328223049565788217965425413589087374616699671804637370572908161 7
46065015 -1.290815387092587891861109872912716870469051788637872937647154002
765630916868849; -0.710236749136630824520178731204546068022231372245102781 0
676710014284176645176509 1.584413436954855058756533182555749764445098208042
220750232325894191522422994462; 7.941083531796850411323446421519371926142 04
```

```
5110365290231595083421616489906527109 3.722378995569464264321557839425572407212804119728732857413506299449999276828452; -0.754205719191458953191834502058314459822854052802833337198759729719190862448541 9-1.2334865328087742843927987122142703015182859891728230103301161761017728207500691]
 [-5.2843718170745453627594806414479785710755588685166059644495300817256896843640620 -0.729125757399840225362707661322376109637650754250178335206557470250971442127054 8; 0.8351258792055375755805729897231989563720499242580608462503422477226895659714712 .598709777257066774218611609934298817619837264689300435925101049272157344734321; 9.701279433034310021289888233497294429262382902009906421067438237990814612732321 3.952172372578267379676806534923306786534931155805109468337046923668578119284993; -1.901204706789334809777183741560540038563960759691440474481969191826012351064192 -1.763508200082036976016522737979390998358223539291664992244896286197180448650274]

sol[1,3]
```

$$-0.581663893403949766850559500017777934571055559270897678058464965933906908351736 2-0.675979834258718103029975503746666995584511850023937301274187987088073869185709 9; -0.176149192233095059880779757708444727469661571356043334$$

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
 0.1105749335330306767670449922914104607366471320738197043929743713614291144588063
 0.3511651471634287496067493149165869155919347177780433860399620496186551457260 78
 0.6079979861780697985274170818263700591963695292895154572610688618088003278252208
 0.9128674287985695517722373100995524702204026141953968264969392169851082841544 87
 1.0
u: 6-element Array{Array{BigFloat,2},1}:
 [0.9338218261404167552797161988564766943454742431640625 0.2502768369662693448418622210738249123096466064453125; 0.6880113011089077357240739729604683816432952880859375 0.5069029535897484617379404880921356379985809326171875; 0.2899275168577648553736025860416702926158905029296875 0.2960337205207028699049942588317207992076873779296875; 0.2189820604968442108173576343688182532787322998046875 0.2920193831063422340577062641386874020099639892578125]
 [0.7723987385574132128156556371143422120985597264319257091941597279261972009619641 0.0586802681616850795275465264844029171261532413860976973210435732509603409422369; 0.7983190269569502497493856872745392969591828801537354646054843421323922317052816 0.4639352370125701585461106883987053968119638701149677746995522723882007339667809; -0.0464327914041657109913640334547738146569581294959473195766220898926485168873278 4 0.2654669516080728219416223948288770624011162947519684356033013409257841160628962; 0.7118183666360206810671941547489126036106026739371138217110541581467796039222019 0.4586651033315070886187618324812081640279844200163632378379402391388166324210852]
 [-0.581663893403949766850559500017777934571055559270897678058464965933906908351736 2-0.675979834258718103029975503746666995584511850023937301274187987088073869185709 9; -0.176149192233095059880779757708444727469661571356043334
```

```
719261565368662431253555422 0.05088016651512752998421833535678705767102361098
834362688155417574601785174489741 8; 0.035992374598249306436398753799593595 7
727101584377170490607647892892675480129152 8 0.669199643771909965950732300609
715870472391625962095898787910250087625663066746 87; 1.534285050696767145383
031531664073807647457901045083108553085160713214005130 83 0.5835063469846612
675482738408630815091731767945588199229826259680228167366278924]
 [-3.121128418917642995306227003496710774199899549905276115460915431544388 7
34010964 -1.513071111332936056720486902199187243591471615925701885344744119
423369147077328; -1.771994688621019304164664089935206470181363018071479642 5
495907729581196572373 36 -0.109425676668356785713678046705182763321444210359
605457083164059287946420955607; 2.301566607287679493407352079766142509435 83
854411114643736086585697646308482187 2 1.931087922124021788434872335099295 83
514144258886469106725674192961363253032668 7; 1.480173141110081163813787047 2
015378837118894695831772381096401484057765169671 85 0.159100136095995417104 3
172107343855158018475235586113665691748337176267667579001]
 [-5.386303328223049587270396256485153903521976656719789212143998119729625 5
59922762 -1.290815387092587827005039547768160000273194195159802363442007475
83568641195931; -0.710236749136630639629146618799537023918776541854914721 50
6831380397060105707706 6 1.584413436954855195180247880598684908622941131620 5
639552656065713094207442329 15; 7.941083531796850687831310488201025632631650
268148628234558797237950784509390913 3.722378995569464316585195039331591425
748584877472456625996100149793514541930 3; -0.754205719191459111341525292284
625468059285619435108083912415767807861345567135 -1.233486532808774362574 7
063400385519778738955530568942638176661113175767803296 65]
 [-5.284371817074545421346665527473583462026426149005053251316159233713529 7
8735029 -0.729125757399840337612017764534421859710817790582452440484566163 6
910427367813847; 0.835125879205537274418799523829914875810370751552853466 78
369064836476583493464 8 2.598709777257066594011490738358063313103019879343 32
590753641235447665984213815 9; 9.701279433034309754332837346478889672171182 1
158979807980855634986576082949849 34 3.952172372578267363675787470632170764 6
594752582813533486704244092161777362699 91; -1.901204706789334610031878198 68
629657495336688664684857517456778123555475316194 4 -1.763508200082036889568 6
150845759006575666274772782909974466787494215020387051 35]
```

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: 12-element Array{Float64,1}:
 0.0
 0.01128478320136641
 0.04432586691283831
 0.1006982762581945
 0.1707099286973292
 0.255116869837141
```

```
 0.35338226190049626
 0.4792279457595038
 0.6232734912548586
 0.7769167439253583
 0.9425102420651629
 1.0
u: 12-element Array{StaticArrays.SArray{Tuple{4,2},Float64,2,8},1}:
 [0.3207548495240642 0.8958480860767932; 0.05145889888617261 0.463421267499
63297; 0.7617439410311491 0.011218901523266211; 0.7149661509423113 0.607770
8489159379]
 [0.2821808873524267 0.8698055278320413; 0.07274078864557351 0.470889734075
241; 0.7565178980737441 -0.02144342377855661; 0.7729001066888023 0.66853448
89058307]
 [0.14799189452720288 0.7720115937701326; 0.11013857521294577 0.46500569436
952344; 0.7560196542049209 -0.10529216495518895; 0.9373025225069743 0.84416
12987564938]
 [-0.15298478118424083 0.530261321928576; 0.0937020971578775 0.362997545605
63124; 0.8151500049895408 -0.19811650473157774; 1.1948583008078497 1.131754
4601518477]
 [-0.6470185470615244 0.09894436058777056; -0.04708620891365348 0.089581214
20544468; 1.017543669945439 -0.19873737519735588; 1.4604617143392435 1.4556
789185454904]
 [-1.3981984202990074 -0.6056311495767073; -0.3403349806960074 -0.414028076
9664418; 1.491675288341294 0.01826623939914243; 1.67313935255188 1.77246727
10082308]
 [-2.430195268099975 -1.6479738035463514; -0.7363369574880048 -1.1430303311
050458; 2.4088462895729705 0.6408571670291584; 1.727491249897271 1.99683194
68649864]
 [-3.8375532555960423 -3.2222844779981794; -1.0560404439682476 -2.056525446
782232; 4.189761002242042 2.1128749351847858; 1.4121971533517272 1.97263538
131972]
 [-5.195026884126617 -5.069506694347904; -0.6504154509301505 -2.58058075975
38326; 6.957192800911287 4.75883152208732; 0.4115677673020941 1.38106006313
05563]
 [-5.692173737110967 -6.536562249871675; 1.3878905551898326 -1.780397632094
5232; 10.30187225351298 8.483102536572735; -1.4835205569181809 -0.061474574
136404]
 [-4.162349577465936 -6.611270586777157; 6.119459024595115 1.54557509062384
26; 13.236546073136779 12.74550520847462; -4.402701390200441 -2.63023565159
13834]
 [-2.9364381041711614 -6.062757991968209; 8.433589842386493 3.4299216604888
72; 13.774156521541126 14.026629420549309; -5.569041351252945 -3.7502670408
7143]

sol[3]

4×@*(2 StaticArrays.SArray(*@{Tuple{4,2},Float64,2,8} with indices SOneTo(4)
×@*(SOneTo(2):0.147992 0.7720120.110139 0.4650060.75602 -0.1052920.937303
0.844161
```

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 1.6.9
[90137ffa-7385-5640-81b9-e52037218182] StaticArrays 0.12.4
[c3572dad-4567-51f8-b174-8c6c989267f4] Sundials 4.3.0
[37e2e46d-f89d-539d-b4ee-838fcccc9c8e] LinearAlgebra
```