An Intro to DifferentialEquations.jl

Chris Rackauckas

June 23, 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)
```

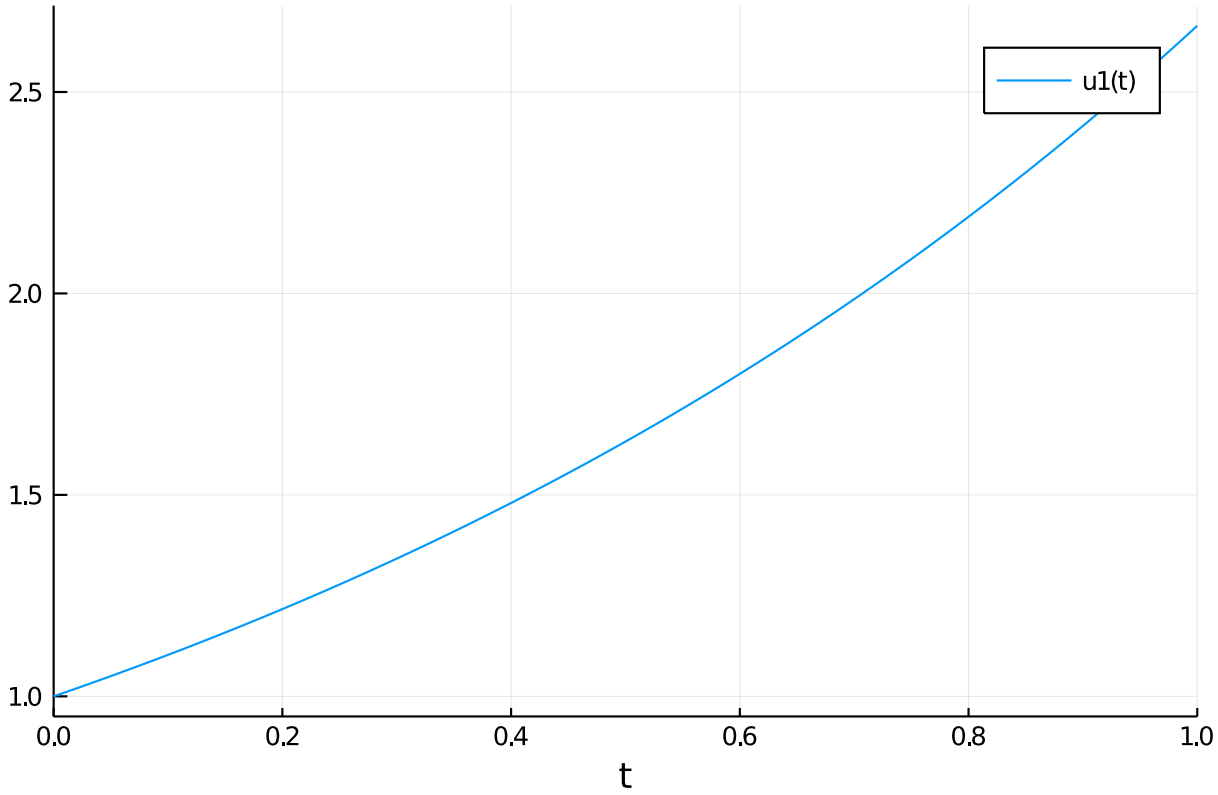
```
retcode: Success
Interpolation: Automatic order switching interpolation
t: 5-element Array{Float64,1}:
 0.0
 0.10042494449239292
 0.35218603951893646
 0.6934436028208104
 1.0
u: 5-element Array{Float64,1}:
 1.0
 1.1034222047865465
 1.4121908848175448
 1.9730384275622996
 2.664456142481451
```

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

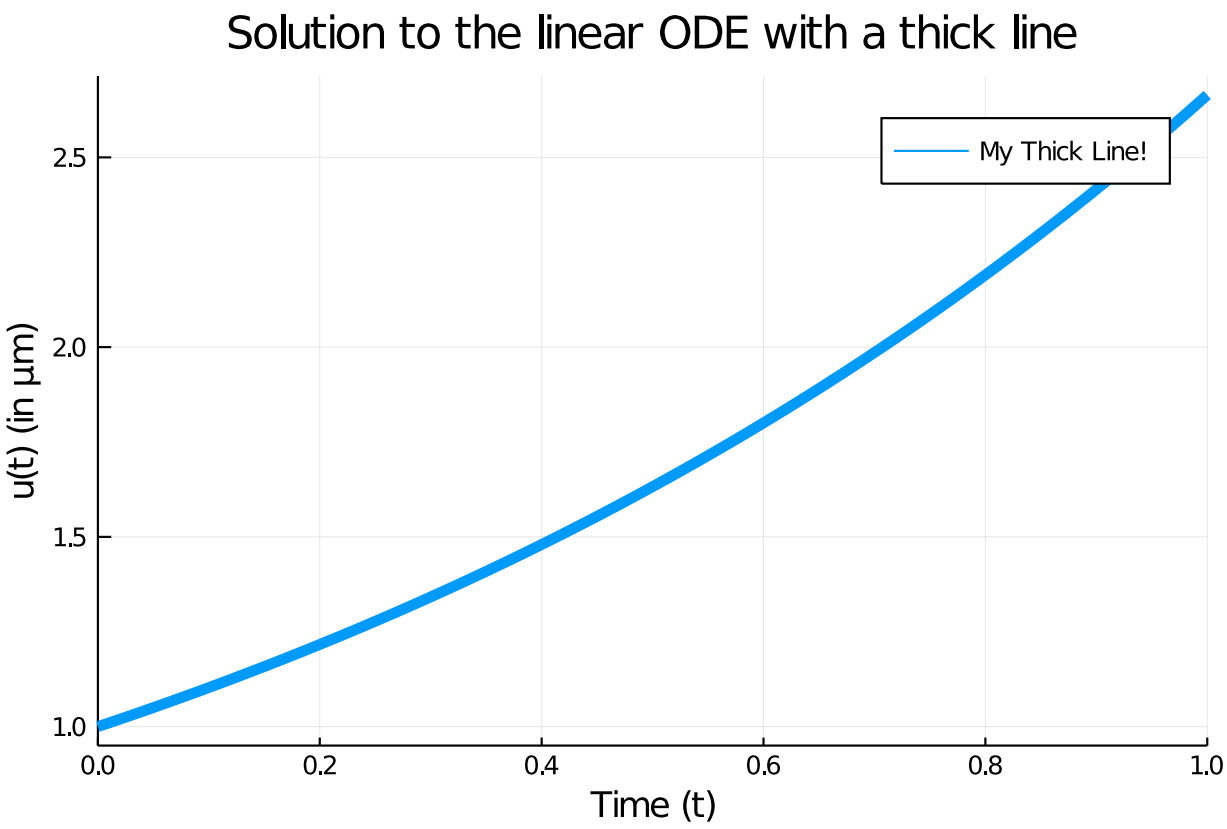
functions for analyzing the solution is the Solution Handling page. Here we will describe some of the basics. You can plot the solution using the plot recipe provided by Plots.jl:

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



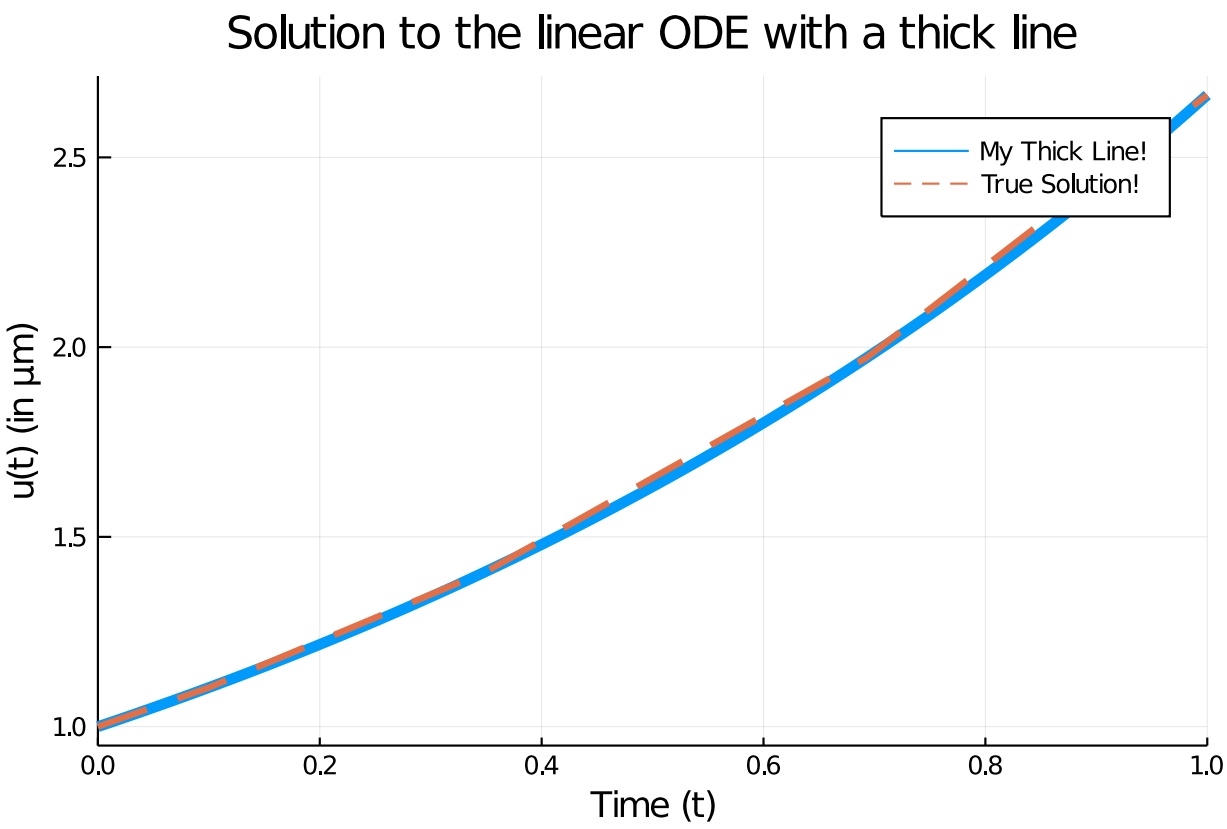
From the picture we see that the solution is an exponential curve, which matches our intuition. As a plot recipe, we can annotate the result using any of the Plots.jl attributes. For example:

```
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)
```

```
1.554261048055312
```

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

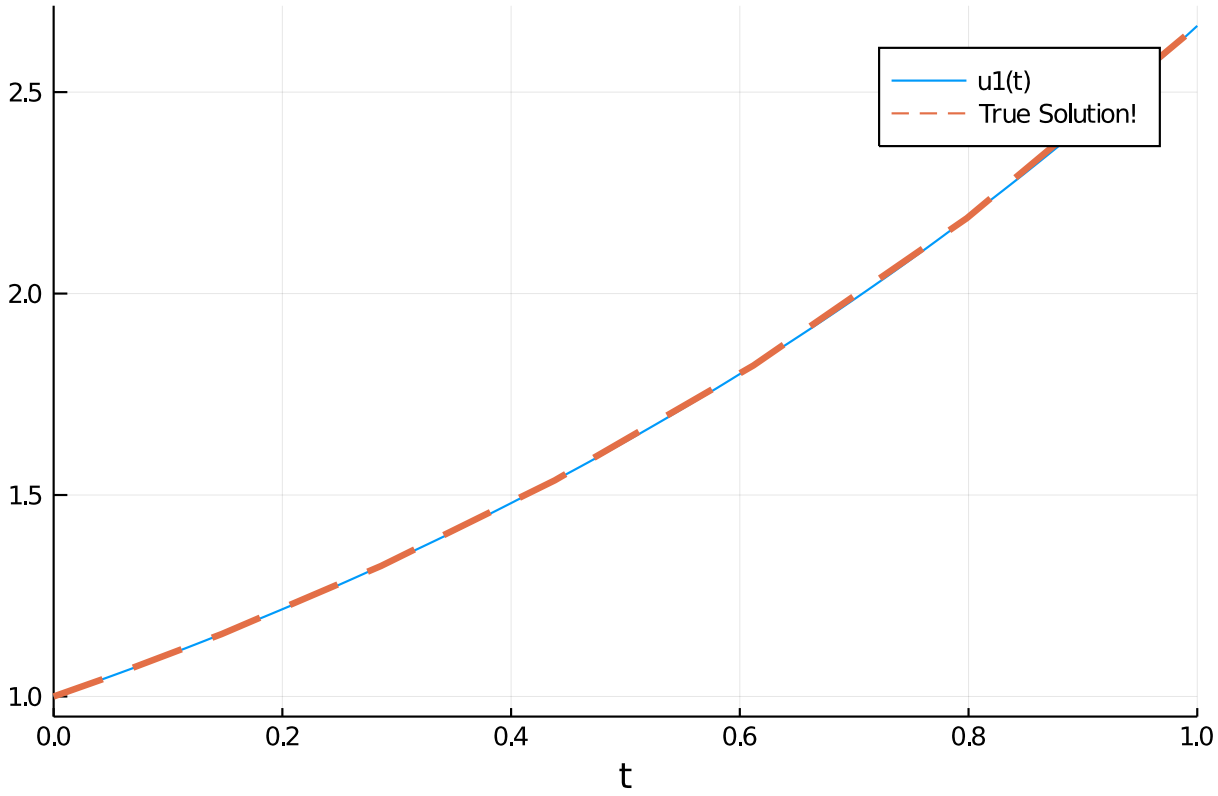
`abstol=1e-6` and `reltol=1e-3`, you can expect a global accuracy of about 1-2 digits. If we want to get around 6 digits of accuracy, we can use the commands:

```
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]
```

```
2.052326153029042
```

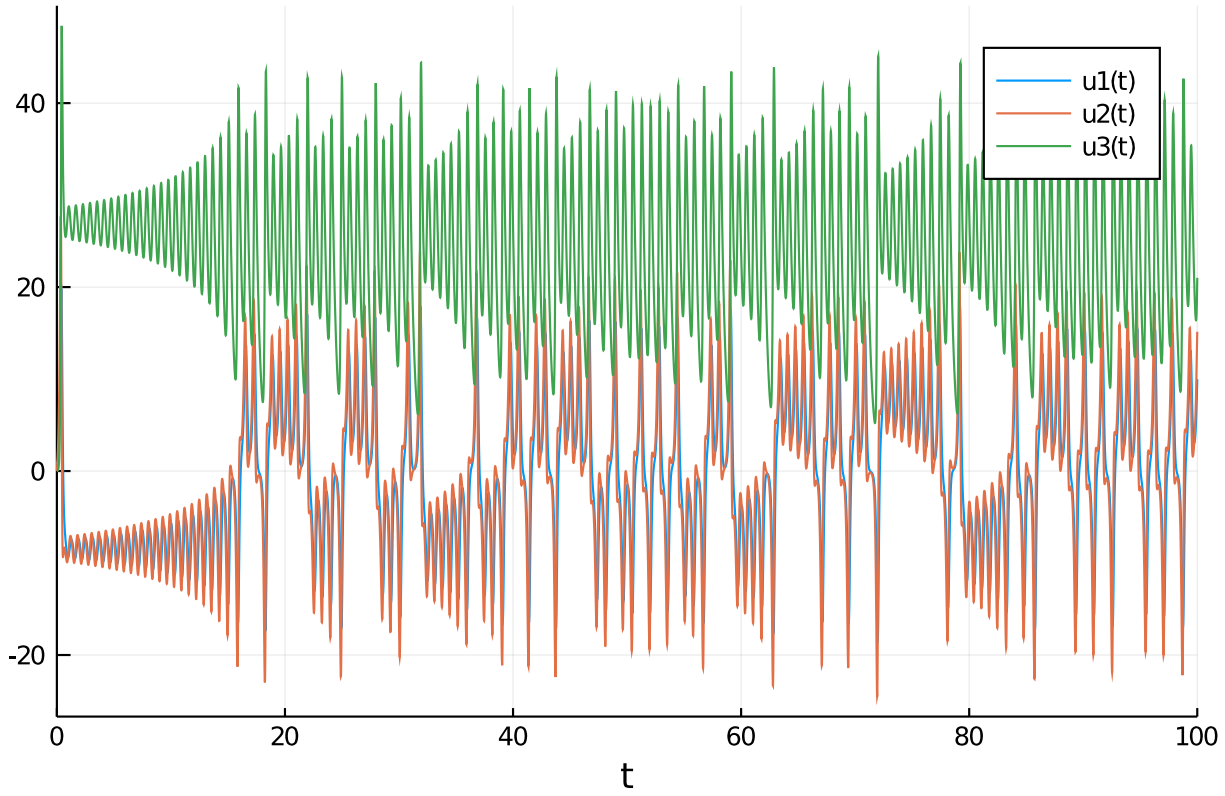
We can get a real matrix by performing a conversion:

```
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
```

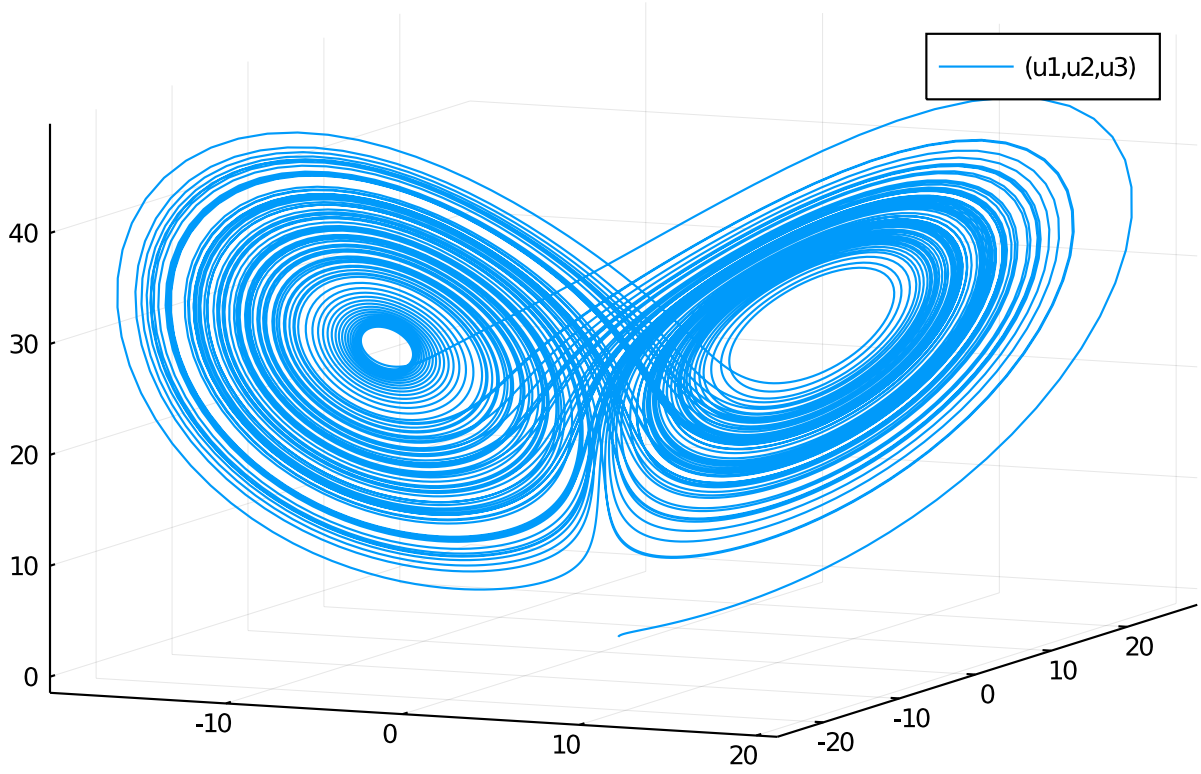
This is the same as sol, i.e. `sol[i,j] = A[i,j]`, but now it's a true matrix. Plotting will by default show the time series for each variable:

```
plot(sol)
```



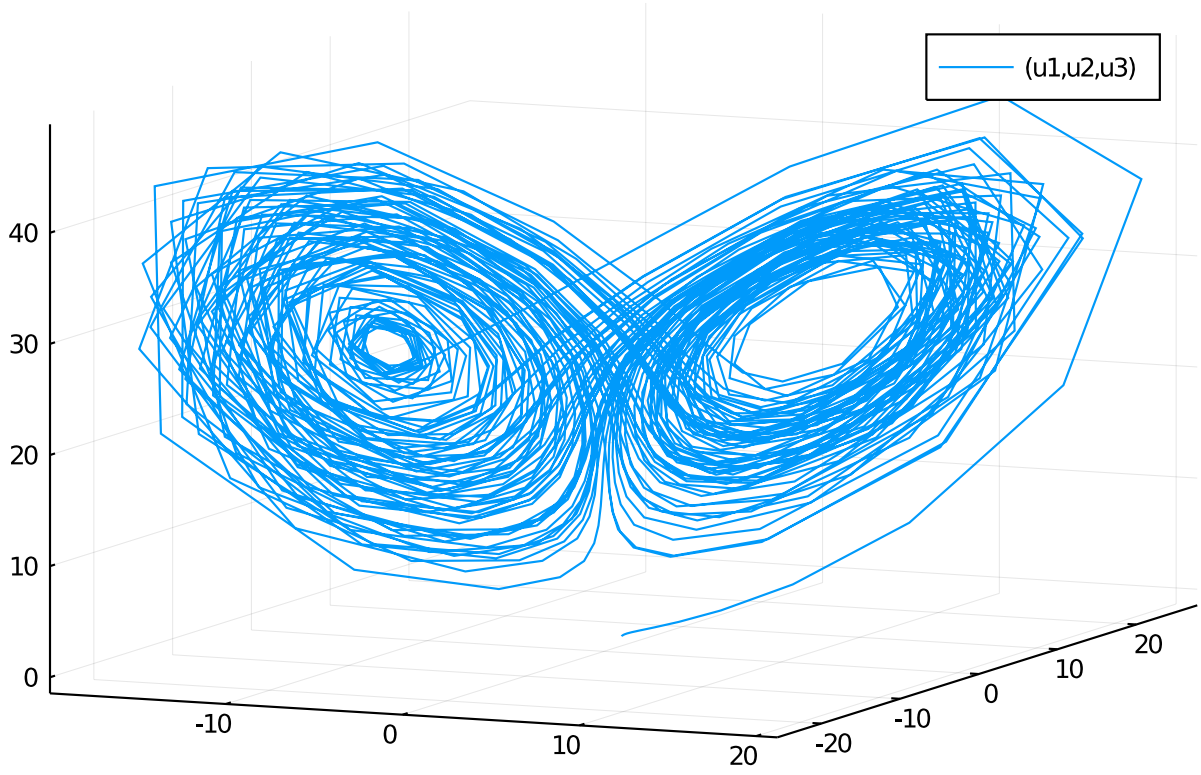
If we instead want to plot values against each other, we can use the `vars` command. Let's plot variable 1 against variable 2 against variable 3:

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



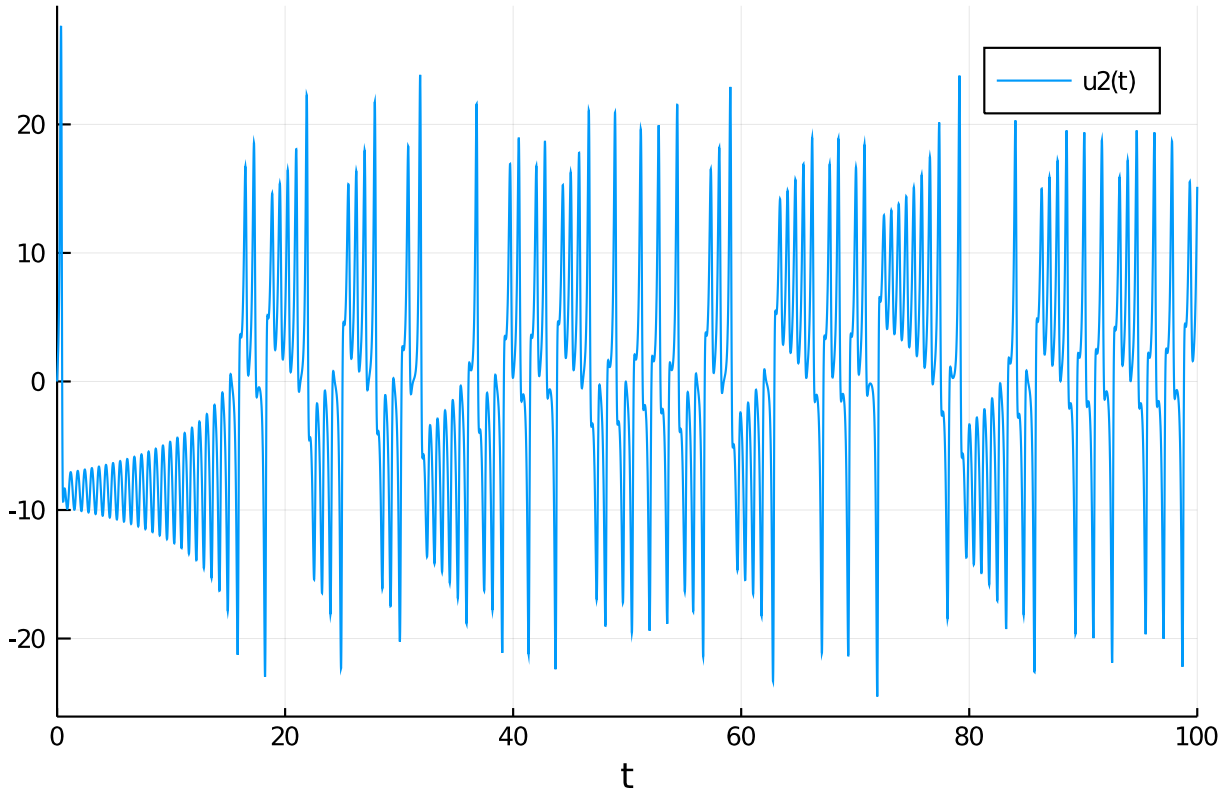
This is the classic Lorenz attractor plot, where the `x` axis is `u[1]`, the `y` axis is `u[2]`, and the `z` axis is `u[3]`. Note that the plot recipe by default uses the interpolation, but we can turn this off:

```
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: 10-element Array{Float64,1}:
 0.0
 0.04255154794632115
```

```
 0.11881055708671834
 0.20839280531685103
 0.31941197269773597
 0.44635363272971307
 0.5711945092820243
 0.6889304792116598
 0.8515479123715806
 1.0
u: 10-element Array{Array{Float64,2},1}:
 [0.27086318650993557 0.5371218905662876; 0.7360264432477825 0.169219895042
5774; 0.6117558270616796 0.8491508682908533; 0.5341381098053661 0.138442447
84194126]
 [0.15451178645087005 0.5099921009629359; 0.7371840370406817 0.344482104858
76685; 0.6002546275825115 0.7693640848810583; 0.643648851961093 0.326813370
2124807]
 [-0.12156809355155412 0.3569524691488778; 0.6580296395151627 0.51783576363
32351; 0.6486943173309617 0.6711027104049014; 0.805698249311254 0.649480924
6122479]
 [-0.5412628867117608 0.00692313868209099; 0.4679535662458876 0.50901759535
69398; 0.8428729047867825 0.6740611348476491; 0.9242221834798543 0.98133050
70215371]
 [-1.1583386246858631 -0.6607950968271938; 0.17806606069300712 0.2496982734
1445122; 1.3221886279089214 0.9371592226070913; 0.9335170803791629 1.275492
8116803548]
 [-1.8895092136155824 -1.663968230498571; -0.05057373667048817 -0.208253840
85542786; 2.205338361591188 1.690696675863989; 0.7151581806489894 1.3787493
252693552]
 [-2.4727809643980416 -2.749654374569755; 0.055487044985545395 -0.557092938
1208766; 3.3658837145226115 2.954791415005629; 0.22609766750728677 1.157401
1043595493]
 [-2.714566844050514 -3.6628387366291406; 0.6582687424573377 -0.50116409814
30461; 4.5894509232665035 4.580220704519057; -0.49739859479461457 0.5938844
814774605]
 [-2.190954572247021 -4.285401384004286; 2.5676043868520795 0.7147093178211
064; 6.062603245032168 7.2139469782622925; -1.8772825748208328 -0.808786125
100691]
 [-0.459499804298231 -3.624074623555232; 5.508399660531837 3.43898033935332
; 6.528697359409731 9.382929558341356; -3.3846816861293627 -2.6907368437845
03]
```

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}:
 -0.121568  0.356952
  0.65803   0.517836
  0.648694  0.671103
  0.805698  0.649481
```

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}:
 0.270863  0.537122
 0.736026  0.16922
 0.611756  0.849151
 0.534138  0.138442
```

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
 0.06424976655859967
 0.28188954342518685
 0.5761214259532718
 0.8732406643102223
 1.0
u: 6-element Array{Array{BigFloat,2},1}:
 [0.270863186509935571777418772398959845304489135742187 5 0.5371218905662875
631179531410452909767627716064453125; 0.736026443247782546919211199565324932
23368072509765625 0.16921989504257739511672298249322921037673950195312 5; 0.
611755827061679635647806207998655736446380615234375 0.849150868290853289011
010929243639111518859863281 25; 0.534138109805366134708037861855700612068176
26953125 0.138442447841941262964837733306922018527984619140625]
 [0.084490330374339149421699963340021022464130522952121881945127305901891493
78645136 0.480168043198149506642765366687612941216235685157126019782096946
9410479346955755; 0.724507664277543834928847963070088645285653403465235277 6
344940602018540860899986 0.411938391386976795733381558349862385447164509596
305328305930768191428765143 5555; 0.604357230890031053499651091997098576979 6
103897798892801819312682118657226612131 0.734433190856616175103560634118520
964496679180609864914482159548062076201132453 8; 0.694618830076542395895969 0
998423412414462114203879548775008414039087378723024412 0.421122026384606600
443189696369714572471204206280329215640598536150560133462725]
 [-0.941991670692428134253632111795878234940210874445995865206000962229680 8
016281184 -0.408837261967847320688679133122112914826951739971675712061665 23
14340700916706039; 0.275200367301072539588730490147138742684352935418098723
202323375217169419648964 7 0.361221709611936655963246253364270799245822961 23
789190927638769089746666390808 75; 1.129242366243063011907175935531896717899
755283287937935794594245845636027802883 0.810805159851038136797606063771980
192372996532576431018243542130671442094890170 7; 0.949474165512330247114620 5
008628396834623169348628704334360633237426541733896938 1.193977197979626499
349305120563764416119257102748373207601784438030332776734843]
 [-2.490268553933259070824326182131217598543211398126780076752010741526769 1
15570987 -2.791618465988424402768787352136892545358529517783323315703437881
371259015699799; 0.069635803096272854459453270951815394816466147242066588 78
577815066467451894484 65 -0.564332051559970256187705422457504397036062646406
633433168055064123568130740564; 3.415843467173672466318354133713174014952 95
137594295917852068425830396144159348 1 3.015059124439811593272181105168005 57
478748471109903968988800530939063051891392 2; 0.200918102174983598988533109 5
```

```
1698802512308562138577987921226959232399505557106681.14107240995592149945630140723337697957912982175105396821128117837896725694865 8]
 [-2.02169437451567948295449300628456701383809133838359627416847091865333437053310 8-4.27761024918244238645653121256864060484250759355447330941848918040583292915844 7; 2.92682414166275758116188548904682319225011235567059775895690458891735566780983 71.00746957295619492576888717695331502474702291736057520009022245758868088462021 9; 6.20256966301887804714038213318847058446357064139406293459266387912954570802582 9 7.56551601466683410827398472199537492666042357245429690356861952922795798090084 6; -2.08795025752329673537136879335516966586493099724389452855792366997778955931289 8 -1.0504553690837975370051988829994761340333842754263051593443163414375807193974957]
 [-0.45949080575458991992626423585584048166319074548780898835815887840912578503020768 -3.62407203555302735745165990676699006480174767782829820278066568421162096860066 5; 5.50841851821539355853590866966240924976263527422071666154066324800144450002052 13.43899898726082627428236753135773971245602462046071704413404107388794881321289 2; 6.52870178915289382406196764744797389442148156416780216256116745173358237444472 49.38294419552648620630332751129511987717515397467323726446523304143204442314056 2; -3.38469057674669129596325206654107253406767051506253760255463224381330942575036 1 -2.690746990457916304447185337281676412886332520539077051171375668140603991718604]
```

```
sol[1,3]
```

```
-0.941991670692428134253632111795878234940210874445995865206000962229680801628118
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: 6-element Array{BigFloat,1}:
 0.0
 0.06424976655859967378842171519064510714413586292591145791021181574921876585260077
 0.28188954342518688255551320992716220253382394233576764075058032238312013450214 73
 0.57612142595327180700656806889211343748297486003429041007296210093875737605277 23
 0.87324066431022237395695889420022183809849925596813175064525748051837241817375 82
 1.0
u: 6-element Array{Array{BigFloat,2},1}:
 [0.27086318650993557177741877239895984530448913574218755371218905662875631179531410452909767627716064453125; 0.73602644324778254691921119956532493233680725097656255 0.16921989504257739511672298249322921037673950195312 5; 0.61175582706167963564780620799865573644638061523437 5 0.84915086829085328901101092924363911151885986328125; 0.53413810980536613470803786185570061206817 6
```

```
26953125 0.13844244784194126296483773330692201852798461914062 5]
 [0.08449033037433912419546061480574957753074395539179374185347649244992416
132717765 0.48016804319814949454226812494949083364327345021481910179788478 3
6417411749952611; 0.72450766427754382914093727486669504564356271799352511 3
4400687274383974434584 72 0.41193839138697681636311624344174329577651204991 6
907768778177409583795587189084 3; 0.60435723089003105615476457081293072977 9
24257224951437178059614294743118993745 5 0.73443319085661616394025771713057 4
31661254155507628809674264054080697932608763 76; 0.69461883007654241276511 9
47223278755880016988739752362877474665442522195624351 4 0.421122026384606632
52271648614166728082840865474636602049557489549278425451509 81]
 [-0.94199167069242823842870289482699870270545028453460418263793305619048 5
595023428 -0.40883726196784743748628987896224877433551638769747390187192659
13971768410698947; 0.27520036730107249106919390944664607418390483649559087 7
14365765824258840647640 01 0.36122170961193660658892118446735652977184923707
51346339682019054409435078039751; 1.12924236624306309828587563199405323147 4
55676616375631756750235081800058286442 8 0.81080515985103818860398380531688 7
35372995589929837505708408507393794766062999 4; 0.94947416551233024430484022
12591202548010603798211918392824016001853181998493012 1.1939771979796265439
70385137591451551459692542141580134208301821842270235698 8]
 [-2.49026855393325918784295900284072695029264550620433973015539435280276 6
82005757 -2.79161846598842468727384361617982749940835204194546259078665536 5
397504226514164; 0.06963580309627295357853707424260842726133290599093273731
26635972222630350386 29 -0.56433205155997030309047243724684194210812003700 8
74251083195173005569438163718 1; 3.41584346717367280657313189829806227932 5
18438082544896080634449342299794959 35 3.01505912443981200536671300094644011
82727619626856149241772849191438873119158 14; 0.20091810217498342634398371 4
05592459328454158086848251455531057305392773132554 3 1.14107240995592138641 4
32145312847978505463964227566948014762054537816741383395 4]
 [-2.02169437451567891041609194760090906667993916258975186907728557915358 4
64449819 -4.27761024918244232016636629049698254663469303063711068415419628 4
759725415440975; 2.92682414166275874647864613513184897925001448114493024411
25521166439273621786 58 1.00746957295619589754387214757053663662198978188721
03154367883694134212259125 98; 6.20256966301887845513932784888111802706633 2
97358898404229430077830593753495161 7.56551601466683520056377263215017010 6
26127216234454850897954059238878201326537; -2.08795025752329740331682589440
65770711249804257069091474439530332515149923625 79 -1.05045536908379831394 2
14730921131709402581405125518175138013423722193179365762]
 [-0.45949080575459014840961997735040347849030323521536111311462471121804 7
556141783 -3.62407203555302749386559565376271924422734528150347809705795239
2382232587778486; 5.50841851821539323360026565184387763182045303475950241 0
59467771424137962961 27 3.43899898726082593802729277379340428306337687173388
62038369792010271728914798 74; 6.52870178915289384552705814536371467703979 4
43617613584474387804803502582966261 9.38294419552648604246845377477412236 8
44551038848896667629969749726238785275871; -3.38469057674669115148214737896
96043937929258557282121383468140889115136598284 09 -2.69074699045791610592 4
29628905278732503673116902678256894322205580342556415139]
```

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
 0.03807305383837506
 0.10865970358989581
 0.1922688299722135
 0.29093935592681297
 0.4091629626615316
 0.5497577276830756
 0.7015974058631393
 0.864021513946766
 1.0
u: 10-element Array{StaticArrays.SArray{Tuple{4,2},Float64,2,8},1}:
 [0.8136950851470042 0.9281405200452042; 0.12302981574750915 0.931712626893
3418; 0.584252356511457 0.9548231665244864; 0.3972812626860944 0.2718726865
6711083]
 [0.745580119075267 0.8913351860022116; 0.25012900720820963 1.0863606698888
95; 0.4845110365261298 0.830072468775261; 0.6311816259771358 0.479302142397
83087]
 [0.4929835679543567 0.7140901168688831; 0.3250847445792962 1.2173041180526
418; 0.3652252301450483 0.647035133538198; 1.0487674214773106 0.84606126574
19855]
 [-0.019078460545491782 0.3224959077444266; 0.16594821700574802 1.131071195
989219; 0.3863159363320796 0.5555648778832759; 1.491254139599501 1.22976557
11055226]
 [-0.9034647714819304 -0.3753228227197931; -0.2956868957972568 0.7569360274
3343; 0.7272160602217572 0.6975741240784037; 1.8924475801645295 1.573041792
5587543]
 [-2.2902829056674014 -1.4822966754902678; -1.0525889452599204 0.0828139400
2079301; 1.7108353786766675 1.3271055729253551; 2.1160794534786165 1.762414
0066434186]
 [-4.184150765626665 -2.9965393236880007; -1.8140502237181542 -0.6750527058
210278; 3.814652235266429 2.8229270210060555; 1.8759655056876348 1.56689171
21837462]
 [-6.049657867829181 -4.482146093826883; -1.7386353060622242 -0.83300426690
64213; 7.16475321037902 5.2976374445790135; 0.8241919136509193 0.7176690915
269411]
 [-6.991993453429237 -5.226499620592426; 0.35132408353931655 0.553577733107
2316; 11.460776679180615 8.50857939466555; -1.3509834789575388 -1.012011277
6640533]
 [-6.129103856662679 -4.545388812702615; 4.34344454316372 3.463624264862365
; 14.760444208300921 10.96157972057534; -3.9887226801966 -3.080507082735765
]
```

```
sol[3]
```

```
4×2 StaticArrays.SArray{Tuple{4,2},Float64,2,8} with indices SOneTo(4)×SOne
To(2):
 0.492984  0.71409
```

```
0.325085  1.2173
0.365225  0.647035
1.04877   0.846061
```

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.

```
Error: ArgumentError: Package DiffEqTutorials not found in current path:
- Run `import Pkg; Pkg.add("DiffEqTutorials")` to install the DiffEqTutoria
ls package.
```

```
Error: UndefVarError: DiffEqTutorials not defined
```