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`.

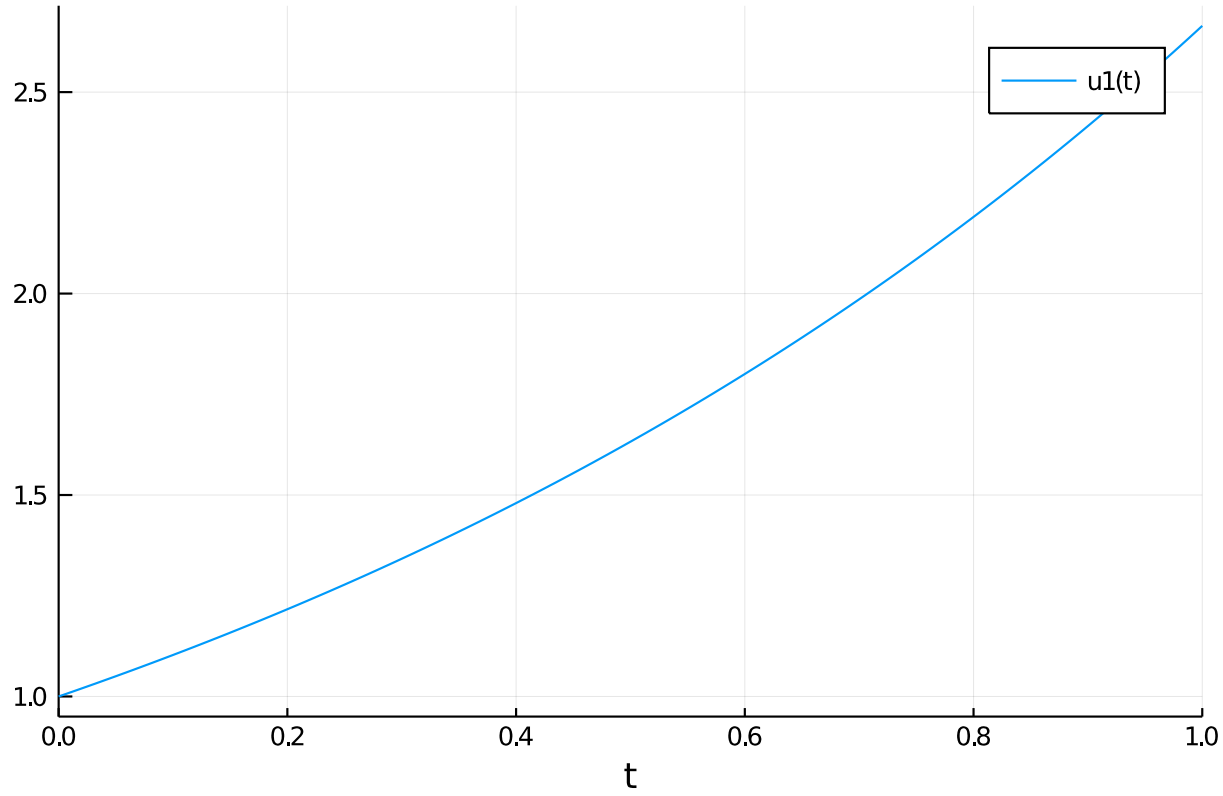
```
sol = solve(prob)
```

```
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!

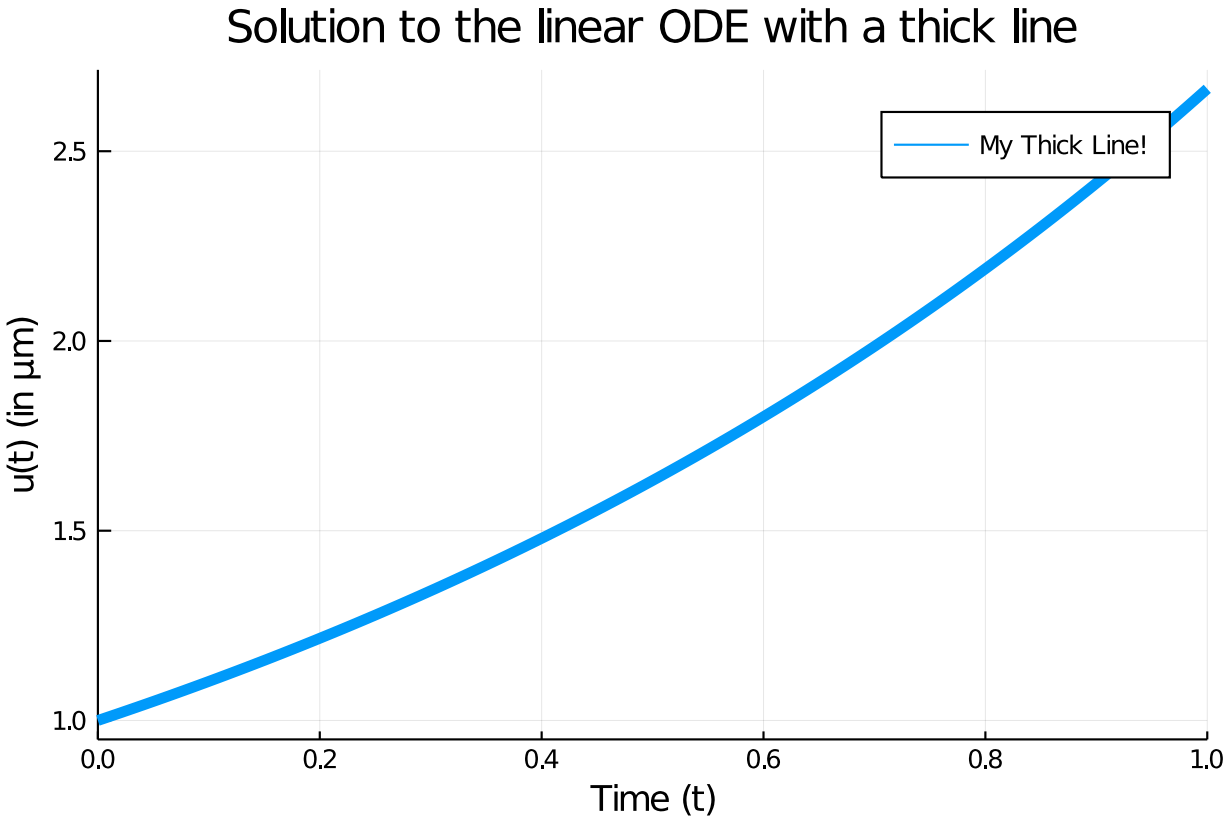
Analyzing the Solution Of course, the solution type is not interesting in and of itself. We want to understand the solution! The documentation page which explains in detail the functions for analyzing the solution is the Solution Handling page. Here we will describe some of the basics. You can plot the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

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



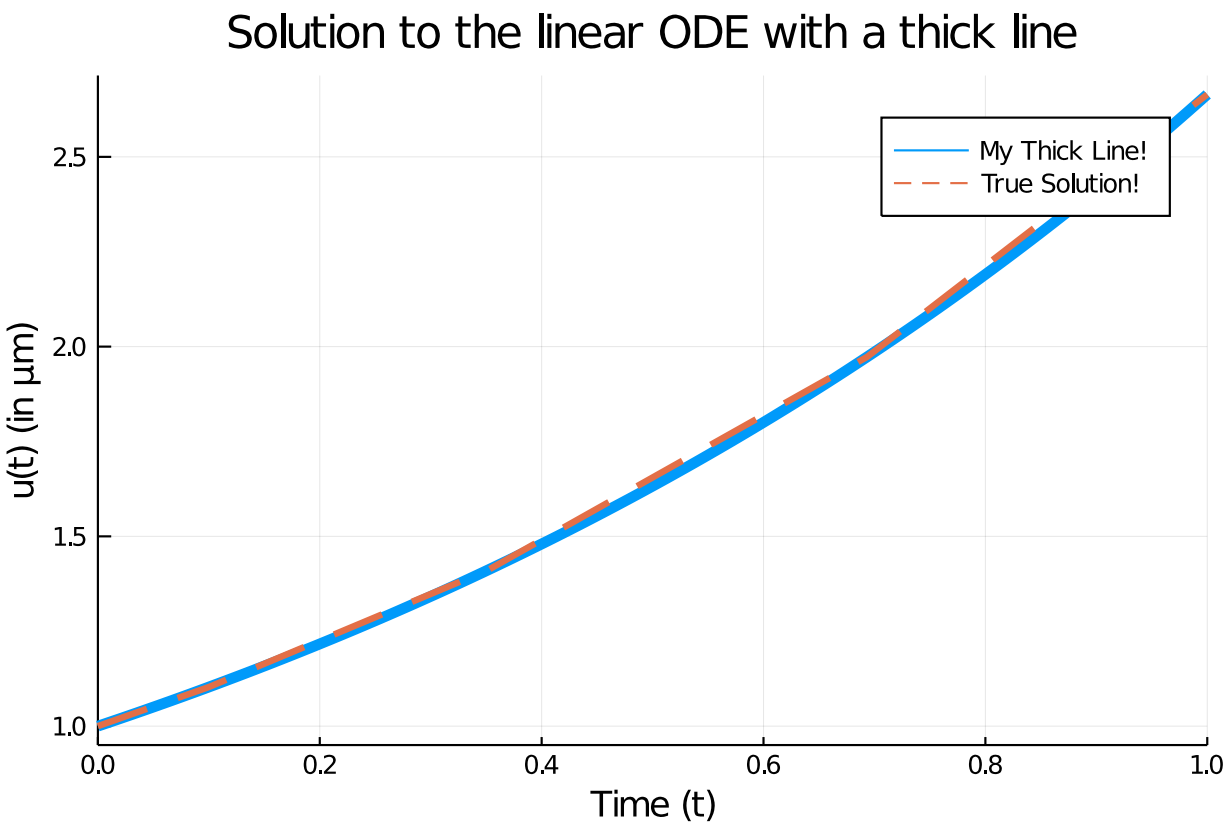
From the picture we see that the solution is an exponential curve, which matches our intuition. As a plot recipe, we can annotate the result using any of the Plots.jl attributes. For example:

```
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:

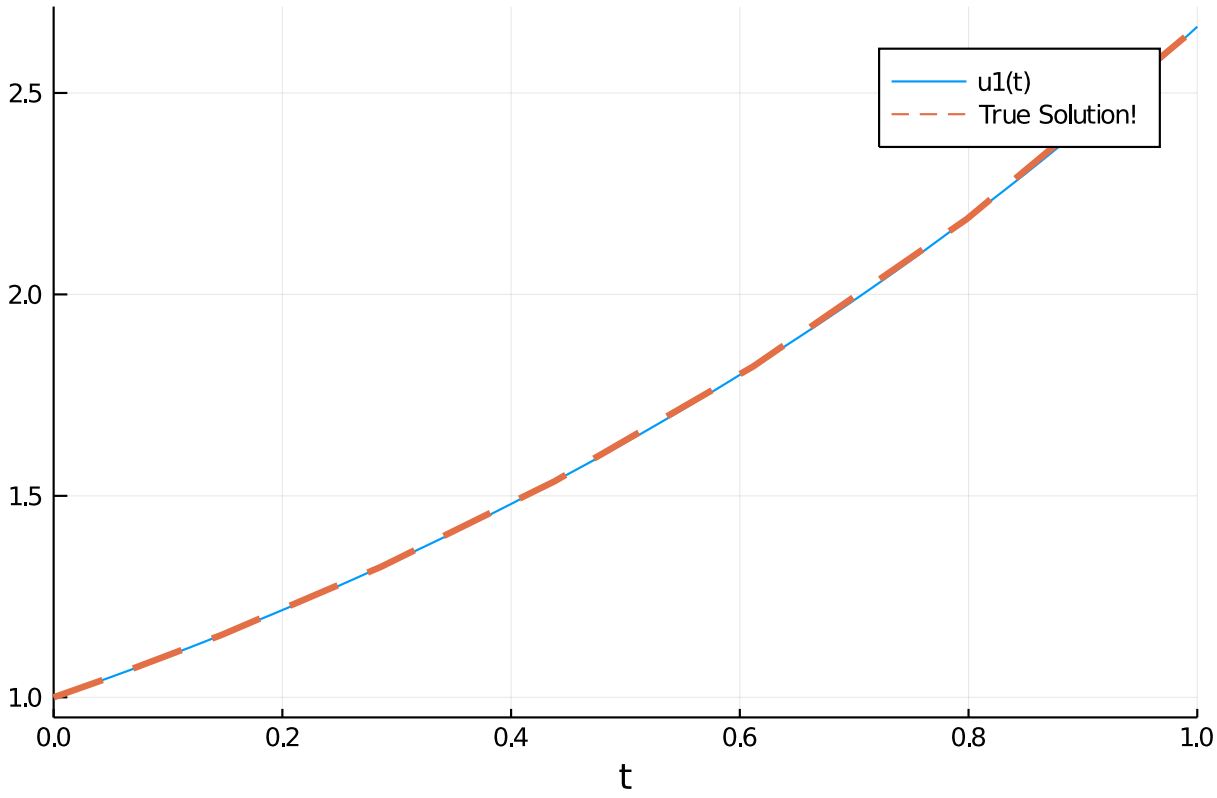
```
sol = solve(prob,abstol=1e-8,reltol=1e-8)
```

```
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
```

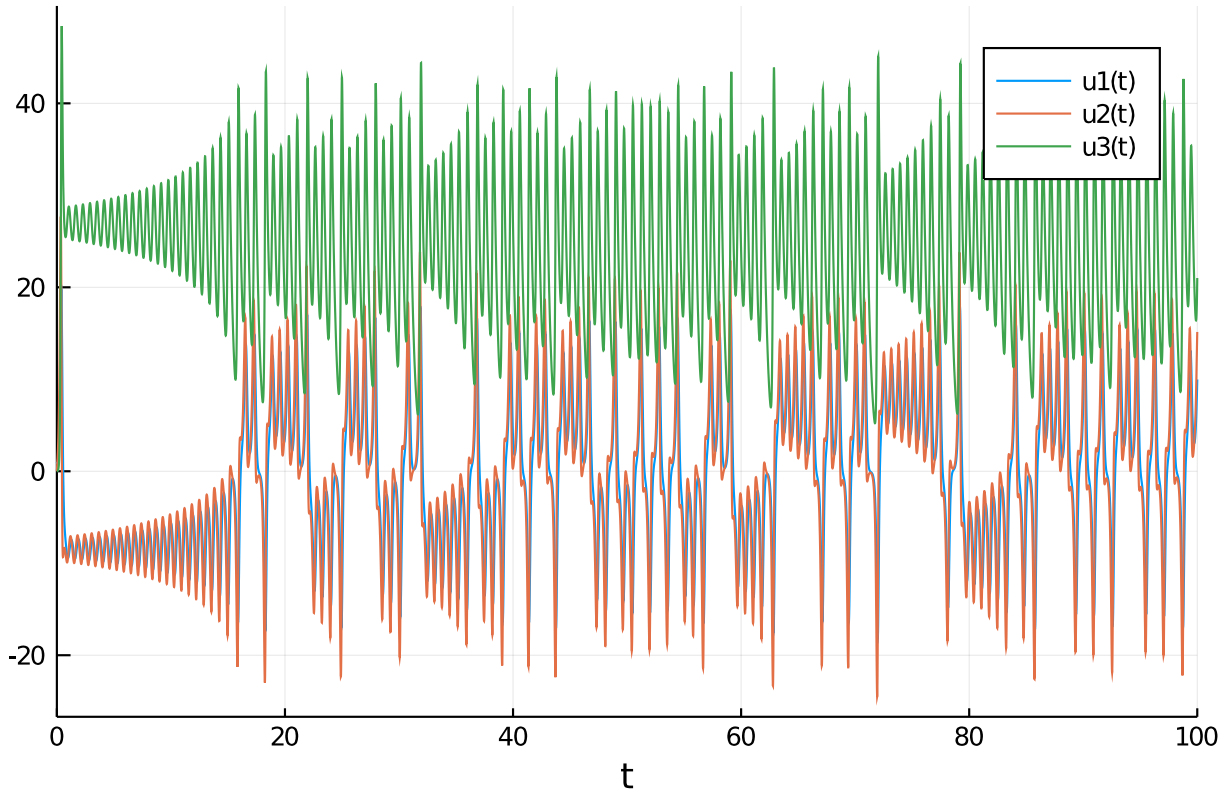
We can get a real matrix by performing a conversion:

```
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
```

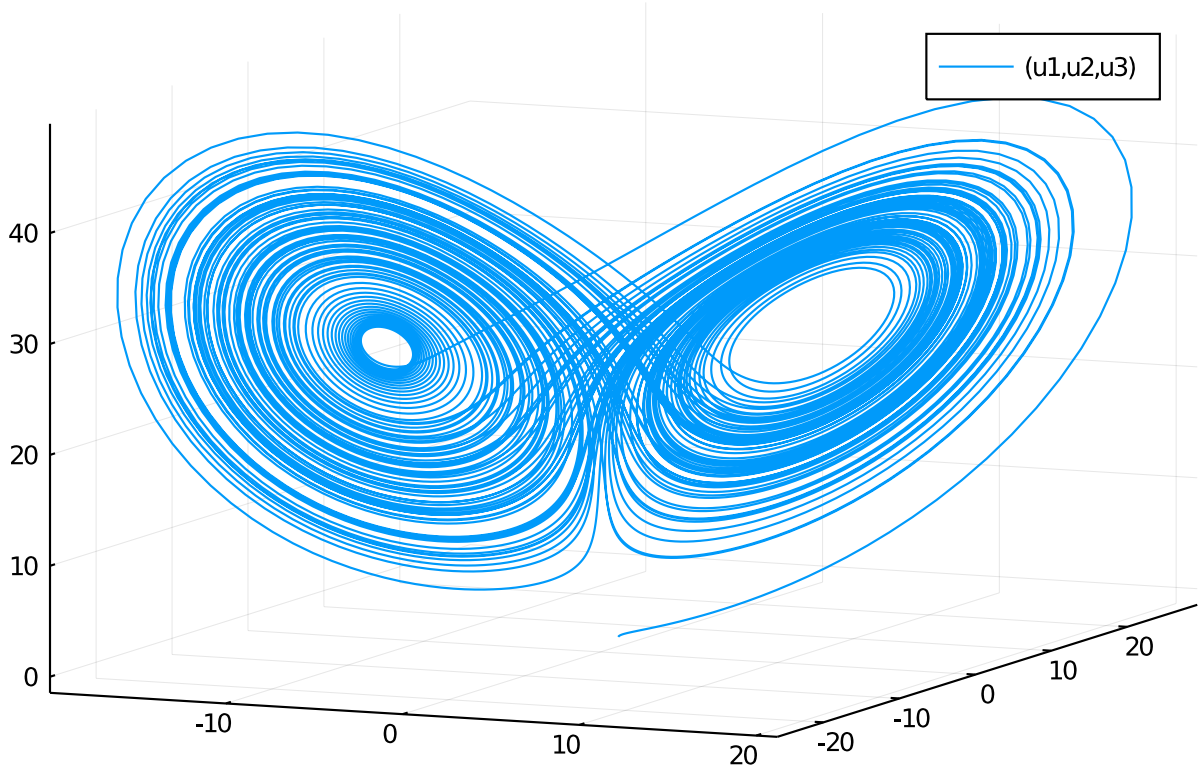
This is the same as sol, i.e. `sol[i,j] = A[i,j]`, but now it's a true matrix. Plotting will by default show the time series for each variable:

```
plot(sol)
```



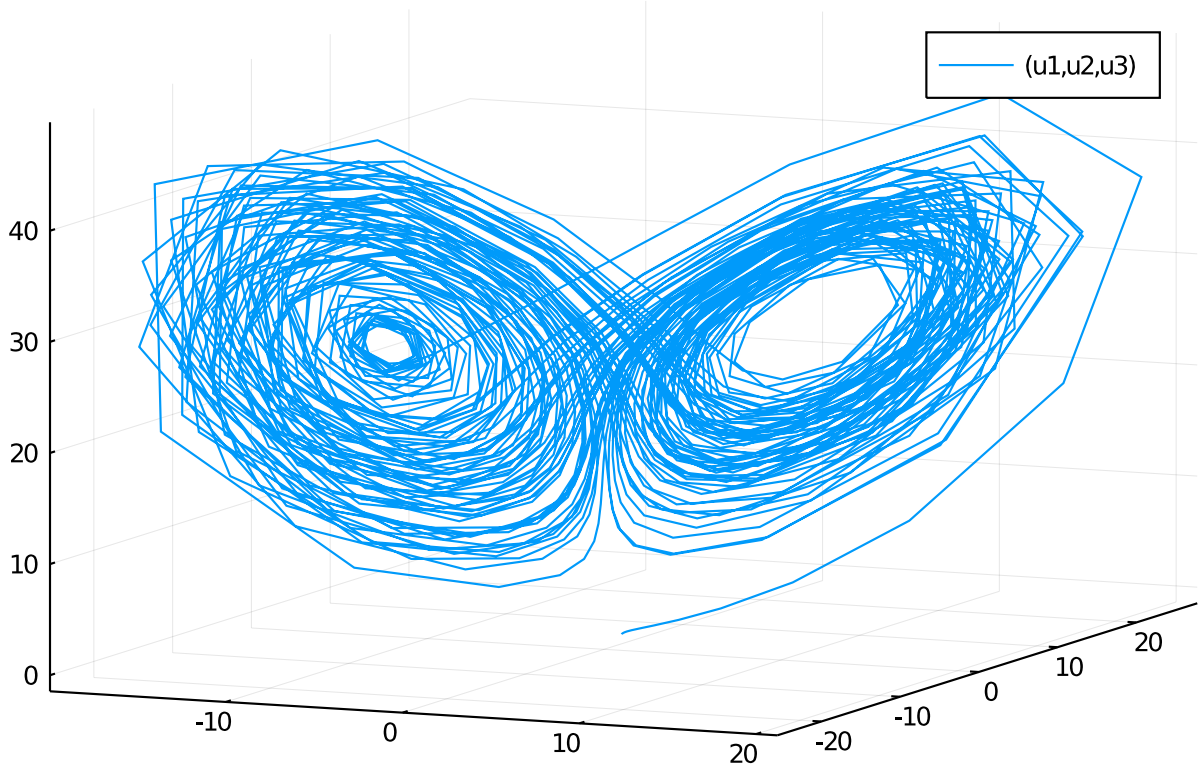
If we instead want to plot values against each other, we can use the `vars` command. Let's plot variable 1 against variable 2 against variable 3:

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



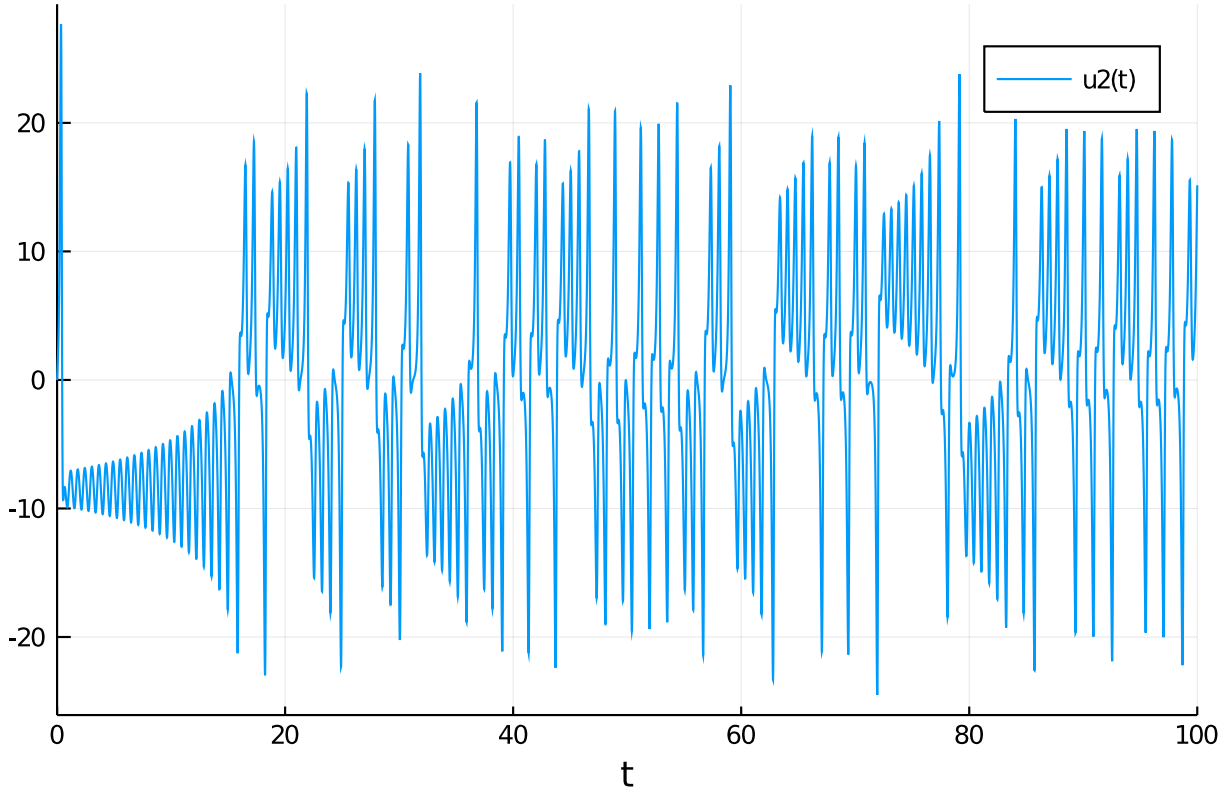
This is the classic Lorenz attractor plot, where the `x` axis is `u[1]`, the `y` axis is `u[2]`, and the `z` axis is `u[3]`. Note that the plot recipe by default uses the interpolation, but we can turn this off:

```
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
 0.042210522326169564
 0.10658495493289794
 0.18227950212113886
 0.279230630852693
 0.39435315833202683
 0.5353297592537557
 0.6906653466817446
 0.8531995952806816
 1.0
u: 10-element Array{Array{Float64,2},1}:
 [0.33276061360315956 0.9664656366454323; 0.23864821189386287 0.53858679425
93108; 0.16808549992556765 0.8023323122218611; 0.8264899800084595 0.8620864
319441555]
 [0.15127598881497242 0.7868100180143107; 0.17480664193693124 0.63329314055
58461; 0.16500045221653775 0.6966200202451989; 0.9894343094094287 1.1914232
592003868]
 [-0.20486207797071818 0.3645388686195464; -0.004963395723663444 0.59643401
2187159; 0.2408704334898219 0.6366161171627388; 1.20935080430305 1.66058794
36928443]
 [-0.7375585095566976 -0.35499301623187024; -0.3155054482566575 0.306647354
4779855; 0.48016840596101396 0.7735363988933626; 1.4088322818888805 2.13474
6395253881]
 [-1.5692834116330137 -1.5951666929331196; -0.7946491679456309 -0.350704000
13256064; 1.0670124186068946 1.3707005733553883; 1.5397353855614502 2.56404
414144361]
 [-2.690393021243958 -3.4217763215480987; -1.3149828639637309 -1.2968310106
825554; 2.219931604209072 2.822506689882954; 1.458416669391111 2.7146081760
66467]
 [-4.035228098309724 -5.857415751835514; -1.5195243401425342 -2.11548981277
74144; 4.298874940984608 5.795970197830837; 0.9169171747381611 2.1949535160
979394]
 [-5.019037181362408 -8.08529850465384; -0.6136329760300004 -1.574693766518
4402; 7.208418248488543 10.403574527366326; -0.3437222759605235 0.512635170
685642]
 [-4.759262413711839 -8.721677850942836; 2.2607923083012675 1.8721993013167
388; 10.265761238784993 15.846794295123114; -2.43482260732931 -2.6231986678
565]
 [-2.673211325861845 -6.569453015713661; 6.894698828079974 8.36064108298544
; 12.019089395003448 19.83030888763293; -4.870821243075247 -6.5550801006250
07]
```

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}:
 -0.204862   0.364539
 -0.0049634  0.596434
  0.24087    0.636616
  1.20935    1.66059
```

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}:
 0.332761  0.966466
 0.238648  0.538587
 0.168085  0.802332
 0.82649   0.862086
```

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

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

```
retcode: Success
Interpolation: Automatic order switching interpolation
t: 5-element Array{Float64,1}:
 0.0
 0.15569746975608992
 0.4114315671969955
 0.7188583947256556
 1.0
u: 5-element Array{Array{BigFloat,2},1}:
 [0.33276061360315956072497556306188926100730895996093750 0.96646563664543228000525232346262782812118530273437
5; 0.23864821189386287159095445531420409679412841796875 0.53858679425931077489053677709307521581649780273437
5; 0.16808549992556764962614579417277127504348754882812
5 0.80233231222186107700622414995450526475906372070312
5; 0.82648998000845952383031089993892237544059753417968
75 0.86208643194415546240350067819235846400260925292968
75]
 [-0.53738064466816326157304300552293379517650820749014542230865967370509148191222
31 -0.07574240339791059820462245946533521754008552431121417012487824163218974682571
38; -0.19677160409995635349826710937347274007750311212888807041288815606447798116705
24 0.43546191796215556742470961164753532066262051841415290898314014336068284197714
45; 0.37585469798175655474106634125531541685164288973060021608203319218216417009506
21 0.69616747781274494559656942766298991817769790730412024578718466620078395399957;
1.34721960102903876997229199703396953051866146896230926173375282727483934649102
7 1.9798363659762326550074166369445795426211066927532306144550588705816919868910
32]
 [-2.86076880706064372062706640579739049462078774429658447127584990200473841151059
7 -3.71381956388042539222275977104182564395665158752819692518245949144973917370565
4; -1.37353697971930327418193169045945114008190051407913623686437678802792917797917
 -1.43001698625625159348502967573379151040821738834571685269714423764931484732722
8; 2.43413859833123067968230193120937190365429984465132562735436208196985809036852
1 3.11244035090347101523992366321152649480301913914707936574557895174220715716375
4; 1.42057347756536585306638883191409572755901648868122276903176086147495451477927
7 2.69652925421366625949689282289736773296861780036330247486456707720419320720102
4]
 [-5.09181423663311863065381763896891262752849913633416792646743924079597172485544
7 -8.35881887492598120514621235559754543180299601251093383924212766030643541379533
6; -0.27264691471412895497929984179251235440032597050810941014054190841480154887594
01 -1.22223853830082342215852449910582594736102689214633693025340331323058091527447
2; 7.76495818438430583108665608303077358507875341431577929537917033599236261386651
1 11.33964071186575242951654360813422126850056980580352217308894990645447861895984;
 -0.65133865169162206979651071779142866792344424188870211672925135873496341830623
11 0.07012101928064699288581155354531910546373930134962422692968197355932578484239
705]
 [-2.67319929436858630854916722580860496201505001727246696096178977938833947265843
2 -6.56944261194326196091547505757747391624346606737695879657591495164777690292337;
 6.89473635579476396415762539253247096185532622100529657425398394216484432573925
2 8.36069503544991722156851336498917752487898983032431597174202091165055760930216
2; 12.01910885231974097697854286340506491183946332441269676326623585091092034661445
4 19.83034792628503124138395982077508536291
```

```
9586322051339230219938854223267491899773; -4.870840243502210169953910451045
0186318181762160426688630020234941106286215028 22 -6.5551092615762742040733 7
29632030484337132417266684770482454930651607787671147 21]

sol[1,3]

-2.860768807060643720627066405797390494620787744296584471275849902004738411
510597
```

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

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

retcode: Success
Interpolation: Automatic order switching interpolation
t: 5-element Array{BigFloat,1}:
 0.0
 0.155697469756089913588807718568687516619688443112387138096523559112913162
4665296
 0.411431567196995395287430115253440834082421886883887496394704171067508115
4462666
 0.718858394725655456825861447853687198929136290477776287070521672345872242
5132711
 1.0
u: 5-element Array{Array{BigFloat,2},1}:
 [0.33276061360315956072497556306188926100730895996093 75 0.9664656366454322
800052523234626278281211853027343 75; 0.2386482118938628715909544553142040 96
79412841796875 0.5385867942593107748905367770930752158164978027343 75; 0.168
085499925567649626145794172771275043487548828125 0.8023323122218610770062 24
149954505264759063720703125; 0.8264899800084595238303108999389223754405975 3
41796875 0.8620864319441554624035006781923584640026092529296875]
 [-0.5373806446681632301957369968813674376033650227245056333145124413120476
911645982 -0.0757424033979105551734588101794435358238512138146065794459287 1
59795230485389777; -0.1967716040999563349732886135222382754548080838024513 3
898314953511091174312732 52 0.4354619179621555860985760022551601783299962942
790089125520293087573736883737941; 0.3758546979817565396563517635958257463 1
939884969013565638103711442018863680728 48 0.6961674778127449357486953566865
102505258136709811647104312294996860082449244531; 1.3472196010290387591874 2
198036515729500440250937315424239786443991716256236606 1.979836365976232628
769172585166499774929760220278351641807377213910755567233245]
 [-2.8607688070606432176607598160258975502504550431767435029141046675063171
94033689 -3.7138195638804245241417949887023443543098029783245555669617605792
432037009398796; -1.3735369797193031115767814346076508754086782852496043031
569167843651766983426 38 -1.4300169862562512080651781113215301859011073177 75
390447310126158477731719686018; 2.4341385983312300303227907542151024845780 8
9104689304552099369538726296224153 68 3.1124403509034701292212344522780173 8
479810212988067078316549195773311313039184 3; 1.4205734775653659755703405006
016544956918396247766850244999182239751264003164 26 2.6965292542136663298963
449899006832097444394609736281148366801611717580687154 88]
 [-5.0918142366331184136730258055224553030867077550314189761279738679645705
05921411 -8.3588188749259801753621854876712076821039752995960857860720761 55
054239411320666; -0.2726469147141305147604953898126807171197832387788785311
959518831583089380551696 -1.2222385383008250961003695892829495580654747186 9
143855301075147101815950586814 5; 7.7649581843843034999175742389580384863036
987528603085599218747870699280880147 14 11.3396407118657484679082654433598 16
```

```
3173180585382438633917738096087217459317787l; -0.651338651691620729229508796524908852816849010332737535729110834557115925819608 6 0.0701210192806489390787941096990660810926270348197477552332754601510548513882930 2]
 [-2.673199294368585706780503363065665701786026119896642604483078433514275738188287 -6.5694426119432612335500247087926566438456408284365005160591577508624690366495 24; 6.894736355794765024606190504418558671080912476742392987598423070313019719273052 8.360695035449918775535423712058707570649693594441894022567170101837894827159873; 12.019108852319741138570350161103582867343410013294928068764968260579588073921 48 19.830347926285031788797836157424049814063737168252636486772801618622497340501 69; -4.870840243502210662053949557281059103406065202209283288780969317208196521676674 -6.5551092615762750248955405838323463011410437721533556164720966278838138154384 51]
```

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
 0.03983644748901097
 0.10516507649258494
 0.1780101580335393
 0.2746286799432771
 0.39342242932136395
 0.5356489457870719
 0.692554939694907
 0.8639577665478819
 1.0
u: 10-element Array{StaticArrays.SArray{Tuple{4,2},Float64,2,8},1}:
 [0.056537708494834193 0.6713118734306907; 0.5096544031684589 0.16884981579008507; 0.8191825025779034 0.25874726196327336; 0.09625813613644074 0.8078590597003692]
 [0.03466736159176746 0.510750939192393; 0.5891934408512067 0.17518087725722098; 0.8165299121332957 0.2007234147482807; 0.1409836690290364 1.0409751192081136]
 [-0.021089434692789046 0.13328613582740645; 0.68798151646258 0.05495365559208955; 0.8255153955899386 0.1933559070115514; 0.2008627512175691 1.3972606217267174]
 [-0.10744527101519863 -0.4503195980833916; 0.759707511974133 -0.24465379839957418; 0.8600700014409388 0.35011236207808594; 0.24376695253326774 1.7375794711849573]
 [-0.24697523740830732 -1.463870764362083; 0.8141800349525307 -0.8316120274132464; 0.9524701717366313 0.8965254405609322; 0.25409911411481834 2.054257004885963]
 [-0.42146423072481065 -2.9934254007244503; 0.8682270791656763 -1.640502897
```

```
724014; 1.1390469464231583 2.1994425755752247; 0.18274338945287313 2.150487
0390480555]
 [-0.5537087568234312 -4.969265325442927; 1.0135144043701827 -2.25637267346
24523; 1.435114958985646 4.7481282766983535; -0.0344063067918606 1.69595925
50531195]
 [-0.4652488218921178 -6.746739901786867; 1.409815352146107 -1.668326171278
593; 1.737327983699928 8.63696735420938; -0.43027355825363367 0.29484376553
96201]
 [0.09676324971553418 -7.142122549795769; 2.2314308720415013 1.594847290412
1709; 1.7761173185708607 13.362240611809934; -0.9865058851932497 -2.4332901
139330385]
 [0.9901454195212699 -5.3877636982721295; 3.146189962989191 6.7294157643363
96; 1.3347024896374164 16.347603006980304; -1.4272805407031395 -5.407932471
238807]

sol[3]

4×2 StaticArrays.SArray{Tuple{4,2},Float64,2,8} with indices SOneTo(4)×SOne
To(2):
 -0.0210894  0.133286
  0.687982   0.0549537
  0.825515   0.193356
  0.200863   1.39726
```

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 1.4.2
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_DEPOT_PATH = /builds/JuliaGPU/DiffEqTutorials.jl/.julia
  JULIA_CUDA_MEMORY_LIMIT = 536870912
  JULIA_PROJECT = @.
  JULIA_NUM_THREADS = 4
```

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 6.14.0
[65888b18-ceab-5e60-b2b9-181511a3b968] ParameterizedFunctions 5.3.0
[91a5bcdd-55d7-5caf-9e0b-520d859cae80] Plots 1.4.3
[90137ffa-7385-5640-81b9-e52037218182] StaticArrays 0.12.3
[c3572dad-4567-51f8-b174-8c6c989267f4] Sundials 4.2.3
[37e2e46d-f89d-539d-b4ee-838fcccc9c8e] LinearAlgebra
```