Solving Equations in With Julia-Defined Types

Chris Rackauckas

March 5, 2019

One of the nice things about DifferentialEquations.jl is that it is designed with Julia's type system in mind. What this means is, if you have properly defined a Number type, you can use this number type in DifferentialEquations.jl's algorithms! [Note that this is restricted to the native algorithms of OrdinaryDiffEq.jl. The other solvers such as ODE.jl, Sundials.jl, and ODEInterface.jl are not compatible with some number systems.]

DifferentialEquations.jl determines the numbers to use in its solvers via the types that are designated by `tspan` and the initial condition of the problem. It will keep the time values in the same type as tspan, and the solution values in the same type as the initial condition. [Note that adaptive timestepping requires that the time type is compaible with `sqrt` and `^` functions. Thus dt cannot be Integer or numbers like that if adaptive timestepping is chosen].

Let's solve the linear ODE first define an easy way to get ODEProblems for the linear ODE:

```
using DifferentialEquations
f = (u,p,t) -> (p*u)
prob_ode_linear = ODEProblem(f,1/2,(0.0,1.0),1.01);
```

First let's solve it using Float64s. To do so, we just need to set u0 to a Float64 (which is done by the default) and dt should be a float as well.

```
prob = prob_ode_linear
sol =solve(prob,Tsit5())
println(sol)
```

```
retcode: Success
Interpolation: specialized 4th order "free" interpolation
t: [0.0, 0.0996426, 0.345703, 0.677692, 1.0]
u: [0.5, 0.552939, 0.708938, 0.99136, 1.3728]
```

Notice that both the times and the solutions were saved as Float64. Let's change the time to use rational values. Rationals are not compatible with adaptive time stepping since they do not have an L2 norm (this can be worked around by defining `internalnorm`, but rationals already explode in size!). To account for this, let's turn off adaptivity as well:

```
prob = ODEProblem(f,1/2,(0//1,1//1),101//100);
```

1

```
sol = solve(prob,RK4(),dt=1//2^(6),adaptive=false)
println(sol)
```

```
retcode: Success
Interpolation: 3rd order Hermite
t: Rational{Int64}[0//1, 1//64, 1//32, 3//64, 1//16, 5//64, 3//32, 7//64, 1
//8, 9//64, 5//32, 11//64, 3//16, 13//64, 7//32, 15//64, 1//4, 17//64, 9//3
2, 19//64, 5//16, 21//64, 11//32, 23//64, 3//8, 25//64, 13//32, 27//64, 7//
16, 29//64, 15//32, 31//64, 1//2, 33//64, 17//32, 35//64, 9//16, 37//64, 19
//32, 39//64, 5//8, 41//64, 21//32, 43//64, 11//16, 45//64, 23//32, 47//64,
 3//4, 49//64, 25//32, 51//64, 13//16, 53//64, 27//32, 55//64, 7//8, 57//64
, 29//32, 59//64, 15//16, 61//64, 31//32, 63//64, 1//1]
u: [0.5, 0.507953, 0.516033, 0.524241, 0.53258, 0.541051, 0.549658, 0.55840
1, 0.567283, 0.576306, 0.585473, 0.594786, 0.604247, 0.613858, 0.623623, 0.
633542, 0.64362, 0.653857, 0.664258, 0.674824, 0.685558, 0.696463, 0.707541
, 0.718795, 0.730229, 0.741844, 0.753644, 0.765632, 0.777811, 0.790183, 0.8
02752, 0.815521, 0.828493, 0.841671, 0.855059, 0.86866, 0.882477, 0.896514,
 0.910775, 0.925262, 0.93998, 0.954931, 0.970121, 0.985552, 1.00123, 1.0171
5, 1.03333, 1.04977, 1.06647, 1.08343, 1.10067, 1.11817, 1.13596, 1.15403,
1.17239, 1.19103, 1.20998, 1.22923, 1.24878, 1.26864, 1.28882, 1.30932, 1.3
3015, 1.35131, 1.3728]
```

Now let's do something fun. Let's change the solution to use `Rational{BigInt}` and print out the value at the end of the simulation. To do so, simply change the definition of the initial condition.

```
prob = ODEProblem(f,BigInt(1)//BigInt(2),(0//1,1//1),101//100);
sol =solve(prob,RK4(),dt=1//2^(6),adaptive=false)
println(sol[end])
```

41540329193865588834329442483803434837620440892198858242938619636906682801338006242715455644424606411004214780699571277051331391310531713199392899156247221954032417368713407455895193878334931538719947505505071664247676041703383322539596306975163054442487962501064886965528244257746528910317816381566346406657267065535626957947163676467986365664901255951417127203808674858689165314566488145289175776934175339650492795688798018631672121713891280290797883948897127735148367985433842763265610542943428517082820508767909688690651283605841517700007145151945514976141613421193476681879508561664377833381251072429460943851264680808184907550924696148357487675219668709370901737689298872020868991281326892017125669358214535685688517619073103608890094548192332030192615116464220451220434614279630678314198226327612575654853082442761181633339340786106693548856458888067417892290768065865070728444712497528988407828353188165924149224845068564398578520709288052499443029691709003030830449621399085676058244288918720817202870441353593800457556211211//30259552635700191640185022778698533980585437459631263972837074707758927127042324370300439207400330261988472164262649512891884983076335911224711118741639261573749898146108785742255065717130085209408458055585794298557073823141968752578356478828562187174172508561251022846835469120207095441551882473797168595729508112819379447023076766794533658143285933059578542748675535941434604752014899870847257974750322570077399294677581910523695792606813529078759274589264848923154827578713239056475245050253159810279037690534441254912000

00000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000

That's one huge fraction!

0.1 Other Compatible Number Types

BigFloats

```
prob_ode_biglinear = ODEProblem(f,big(1.0)/big(2.0),(big(0.0),big(1.0)),big(1.01))
sol =solve(prob_ode_biglinear,Tsit5())
println(sol[end])
```

```
1.37280044090380872778928318231411552985333601446142133501450986619466611676
11229
```

DoubleFloats.jl   There's are Float128-like types. Higher precision, but fixed and faster than arbitrary precision.

```
using DoubleFloats
prob_ode_doublelinear =
    ODEProblem(f,Double64(1)/Double64(2),(Double64(0),Double64(1)),Double64(1.01))
sol =solve(prob_ode_doublelinear,Tsit5())
println(sol[end])
```

```
1.3728004409038075
```

ArbFloats   These high precision numbers which are much faster than Bigs for less than 500-800 bits of accuracy.

```
using ArbNumerics
prob_ode_arbfloatlinear =
    ODEProblem(f,ArbFloat(1)/ArbFloat(2),(ArbFloat(0.0),ArbFloat(1.0)),ArbFloat(1.01))
sol =solve(prob_ode_arbfloatlinear,Tsit5())
println(sol[end])
```

```
1.37280044090380872778928318231
```

0.2 Incompatible Number Systems

DecFP.jl   Next let's try DecFP. DecFP is a fixed-precision decimals library which is made to give both performance but known decimals of accuracy. Having already installed DecFP with]add DecFP, I can run the following:

```
using DecFP
prob_ode_decfplinear =
    ODEProblem(f,Dec128(1)/Dec128(2),(Dec128(0.0),Dec128(1.0)),Dec128(1.01))
sol =solve(prob_ode_decfplinear,Tsit5())
```

```
Error: StackOverflowError:
```

```
println(sol[end]); println(typeof(sol[end]))
```

```
1.37280044090380872778928382314
ArbNumerics.ArbFloat{128}
```

Decimals.jl Install with `]add Decimals`.

```
using Decimals
prob_ode_decimallinear =
    ODEProblem(f,[decimal("1.0")]./[decimal("2.0")],(0//1,1//1),decimal(1.01))
sol =solve(prob_ode_decimallinear,RK4(),dt=1/2^(6)) #Fails
```

```
Error: MethodError: Decimals.Decimal(::Rational{Int64}) is ambiguous. Candi
dates:
  (::Type{T})(x::Rational{S}) where {S, T<:AbstractFloat} in Base at ration
al.jl:92
  Decimals.Decimal(num::Real) in Decimals at C:\Users\accou\.julia\packages
\Decimals\Qfcas\src\decimal.jl:13
Possible fix, define
  Decimals.Decimal(::Rational{S})
```

```
println(sol[end]); println(typeof(sol[end]))
```

```
1.37280044090380872778928382314
ArbNumerics.ArbFloat{128}
```

At the time of writing this, Decimals are not compatible. This is not on DifferentialEquations.jl's end, it's on partly on Decimal's end since it is not a subtype of Number. Thus it's not recommended you use Decimals with DifferentialEquations.jl

0.3   Conclusion

As you can see, DifferentialEquations.jl can use arbitrary Julia-defined number systems in its arithmetic. If you need 128-bit floats, i.e. a bit more precision but not arbitrary, Double-Floats.jl is a very good choice! For arbitrary precision, ArbNumerics are the most feature-complete and give great performance compared to BigFloats, and thus I recommend their

use when high-precision (less than 512-800 bits) is required. DecFP is a great library for high-performance decimal numbers and works well as well. Other number systems could use some modernization.

```
using DiffEqTutorials
DiffEqTutorials.tutorial_footer(WEAVE_ARGS[:folder],WEAVE_ARGS[:file])
```

0.4 Appendix

These benchmarks are part of the DiffEqTutorials.jl repository, found at: https://github.com/JuliaDiffEq/

To locally run this tutorial, do the following commands:

```
using DiffEqTutorials
DiffEqTutorials.weave_file("type_handling","number_types.jmd")
```

Computer Information:

```
Julia Version 1.1.0
Commit 80516ca202 (2019-01-21 21:24 UTC)
Platform Info:
  OS: Windows (x86_64-w64-mingw32)
  CPU: Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-6.0.1 (ORCJIT, skylake)
Environment:
  JULIA_EDITOR = "C:\Users\accou\AppData\Local\atom\app-1.34.0\atom.exe" -a
  JULIA_NUM_THREADS = 6
```

Package Information:

```
Status `C:\Users\accou\.julia\external\DiffEqTutorials.jl\Project.toml`
[7e558dbc-694d-5a72-987c-6f4ebed21442] ArbNumerics 0.3.6
[6e4b80f9-dd63-53aa-95a3-0cdb28fa8baf] BenchmarkTools 0.4.2
[be33ccc6-a3ff-5ff2-a52e-74243cff1e17] CUDAnative 1.0.1
[3a865a2d-5b23-5a0f-bc46-62713ec82fae] CuArrays 0.9.1
[55939f99-70c6-5e9b-8bb0-5071ed7d61fd] DecFP 0.4.8
[abce61dc-4473-55a0-ba07-351d65e31d42] Decimals 0.4.0
[459566f4-90b8-5000-8ac3-15dfb0a30def] DiffEqCallbacks 2.5.2
[f3b72e0c-5b89-59e1-b016-84e28bfd966d] DiffEqDevTools 2.6.1
[1130ab10-4a5a-5621-a13d-e4788d82bd4c] DiffEqParamEstim 1.6.0
[055956cb-9e8b-5191-98cc-73ae4a59e68a] DiffEqPhysics 3.1.0
[0c46a032-eb83-5123-abaf-570d42b7fbaa] DifferentialEquations 6.3.0
[497a8b3b-efae-58df-a0af-a86822472b78] DoubleFloats 0.7.5
[f6369f11-7733-5829-9624-2563aa707210] ForwardDiff 0.10.3
```

```
[7073ff75-c697-5162-941a-fcdaad2a7d2a] IJulia 1.17.0
[eff96d63-e80a-5855-80a2-b1b0885c5ab7] Measurements 2.0.0
[429524aa-4258-5aef-a3af-852621145aeb] Optim 0.17.2
[1dea7af3-3e70-54e6-95c3-0bf5283fa5ed] OrdinaryDiffEq 5.3.0
[65888b18-ceab-5e60-b2b9-181511a3b968] ParameterizedFunctions 4.1.1
[91a5bcdd-55d7-5caf-9e0b-520d859cae80] Plots 0.23.1
[731186ca-8d62-57ce-b412-fbd966d074cd] RecursiveArrayTools 0.20.0
[90137ffa-7385-5640-81b9-e52037218182] StaticArrays 0.10.3
[c3572dad-4567-51f8-b174-8c6c989267f4] Sundials 3.1.0
[1986cc42-f94f-5a68-af5c-568840ba703d] Unitful 0.14.0
[2a06ce6d-1589-592b-9c33-f37faeaed826] UnitfulPlots 0.0.0
[44d3d7a6-8a23-5bf8-98c5-b353f8df5ec9] Weave 0.7.2
[b77e0a4c-d291-57a0-90e8-8db25a27a240] InteractiveUtils
[37e2e46d-f89d-539d-b4ee-838fcccc9c8e] LinearAlgebra
[44cfe95a-1eb2-52ea-b672-e2afdf69b78f] Pkg
```