Solving Equations in With Julia-Defined Types

Chris Rackauckas

February 25, 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 Δt, 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);
```

```
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])
```

```
41540329193865588834329442483803434837620440892198858242938619636906682801
38006242715455644424606411004214780699571277051331391310531713199392899156 2
47221954032417368713407455895193878334931538719947505505071664247676041703 3
83322539596306975163054442487962501064886965528244257746528910317816381566 3
46406657267065535626957947163676467986365664901255951417127203808674858689 1
65314566488145289175776934175339650492795688798018631672121713891280290797 8
83948897127735148367985433842763265610542943428517082820508767909688690651 2
83605841517700007145151945514976141613421193476681879508561664377833381251 0
72429460943851264680808184907550924696148357487675219668709370901737689298 8
72020868991281326892017125669358214535685688517619073103608890094548192332 0
30192615116464220451220434614279630678314198226327612575654853082442761181 6
33339340786106693548856458888067417892290768065865070728444712497528988407 8
28353188165924149224845068564398578520709288052499443029691709003030830449 6
21399085676058244288918720817202870441353593800457556211 21//302595526357001
91640185022778698533980585437459631263972837074707758927127042324370300439 2
07400330261988472164262649512891884983076335911224711118741639261573749898 1
46108785742255065717130085209408458055585794298557073823141968752578356478 8
28562187174172508561251022846835469120207095441551882473797168595729508112 8
19379447023076766794533658143285933059578542748675535941434604752014899870 8
47257974750322570077399294677581910523695792606813529078759274589264848923 1
54827578713239056475245050253159810279037690534441254912000000000000000000 0
00000000000000000000000000000000000000000000000000000000000000000000000000 0
00000000000000000000000000000000000000000000000000000000000000000000000000 0
00000000000000000000000000000000000000000000000000000000000000000000000000 0
```

```
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
```

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])
```

```
1.372800440903808727789283182314115529853336014461421335014509866194661167611229
```

DoubleFloat.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
```

0.2 Incompatible Number Systems

ArbFloats These high precision numbers which are much faster than Bigs for less than 500-800 bits of accuracy. Having already installed Nemo and ArbFloats, I can use them in DifferentialEquations.jl via:

```
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())
```

```
Error: MethodError: no method matching decompose(::ArbNumerics.ArbFloat{128
})
Closest candidates are:
  decompose(!Matched::BigFloat) at hashing2.jl:133
  decompose(!Matched::Float64) at hashing2.jl:122
  decompose(!Matched::Float32) at hashing2.jl:111
  ...
```

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

```
1.3728004409038075
```

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]))
```

```
1.3728004409038075
DoubleFloats.DoubleFloat{Float64}
```

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]))
```

```
1.3728004409038075
DoubleFloats.DoubleFloat{Float64}
```

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, DoubleFloats.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.

0.4 Appendix

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

```
These benchmarks are part of the DiffEqTutorials.jl repository, found at:

https://github.com/JuliaDiffEq/DiffEqTutorials.jl

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\environments\v1.1\Project.toml`
  [7e558dbc] ArbNumerics v0.3.5
  [c52e3926] Atom v0.7.14
  [6e4b80f9] BenchmarkTools v0.4.2
  [336ed68f] CSV v0.4.3
  [be33ccc6] CUDAnative v1.0.1
  [3a865a2d] CuArrays v0.9.1
  [a93c6f00] DataFrames v0.17.1
```

```
  [55939f99] DecFP v0.4.8
  [abce61dc] Decimals v0.4.0
  [39dd38d3] Dierckx v0.4.1
  [459566f4] DiffEqCallbacks v2.5.2
  [f3b72e0c] DiffEqDevTools v2.6.1
  [aae7a2af] DiffEqFlux v0.2.0
  [c894b116] DiffEqJump v6.1.0+ [`C:\Users\accou\.julia\dev\DiffEqJump`]
  [1130ab10] DiffEqParamEstim v1.6.0+ [`C:\Users\accou\.julia\dev\DiffEqPar
amEstim`]
  [055956cb] DiffEqPhysics v3.1.0
  [225cb15b] DiffEqTutorials v0.0.0 [`C:\Users\accou\.julia\external\DiffEq
Tutorials.jl`]
  [0c46a032] DifferentialEquations v6.3.0
  [497a8b3b] DoubleFloats v0.7.5
  [587475ba] Flux v0.7.3
  [f6369f11] ForwardDiff v0.10.3+ [`C:\Users\accou\.julia\dev\ForwardDiff`]
  [7073ff75] IJulia v1.17.0
  [c601a237] Interact v0.9.1
  [b6b21f68] Ipopt v0.5.4
  [4076af6c] JuMP v0.18.5
  [e5e0dc1b] Juno v0.5.4
  [eff96d63] Measurements v2.0.0
  [76087f3c] NLopt v0.5.1
  [429524aa] Optim v0.17.2
  [1dea7af3] OrdinaryDiffEq v5.2.1+ [`C:\Users\accou\.julia\dev\OrdinaryDif
fEq`]
  [65888b18] ParameterizedFunctions v4.1.0
  [91a5bcdd] Plots v0.23.0
  [71ad9d73] PuMaS v0.0.0 [`C:\Users\accou\.julia\dev\PuMaS`]
  [d330b81b] PyPlot v2.7.0
  [731186ca] RecursiveArrayTools v0.20.0
  [90137ffa] StaticArrays v0.10.2
  [789caeaf] StochasticDiffEq v6.1.1+ [`C:\Users\accou\.julia\dev\Stochasti
cDiffEq`]
  [c3572dad] Sundials v3.0.0
  [1986cc42] Unitful v0.14.0
  [44d3d7a6] Weave v0.7.1 [`C:\Users\accou\.julia\dev\Weave`]
```