Native Julia Solvers for Ordinary Differential Equations Boundary Value Problem: A GSoC proposal

Yingbo Ma

April 3, 2017

Contents

1 The Project

1.1 Basic concepts for nonexperts

An ordinary differential equation (ODE) is an equality relationship between a function $y(x)$ and its derivatives, and an nth order ODE can be written as

$$F(x, y, y', \cdots, y^{(n)}) = 0.$$

Almost every branch of science often presents ODE problems. Many physical phenomenas can be reduced into an ODE. Newton's second law of motion, namely, $F = ma$ can be rewritten into an ODE as $m\frac{d^2x}{dt^2} = F(x)$, since a can depend on time. ODEs can be very difficult to solve. There are many cases in which an ODE does not have an analytical solution. Therefore, numerical methods need to be used to solve ODEs by approximating the solution. There are two kinds of ODE problems. One is the initial value problem (IVP) and the other is the boundary value problem (BVP). For instance, the ODE

$$m\frac{d^2x}{dt^2} = -kx(t)$$

describes the motion for a harmonic oscillator. If the initial position x_0 and initial velocity $\frac{dx_0}{dt}$ are known, then it is an IVP problem. If the condition at the "boundary" is known, for instance, the initial position x_0 and final position x_1, then it is a BVP problem. My proposal is for the development of efficient and general-purpose BVP solvers for the Julia ecosystem.

1.1.1 Introduction

The project that I propose to work on in Google's Summer of Code project is the native Julia implementation of some BVP solving methods for ODE, namely, the collocation method and the shooting method. There is no solver for solving ODE BVP problems yet in the *JuliaDiffEq* organization. There are a number of advantages for Julia native solvers over calling C or Fortran's existing libraries via Julia's FFI, including the ability to use a wide variety of types like *BigFloat* for arbitrary precision, easier for people to contribute and better integrate with other packages in Julia ecosystem.

1.2 Project Goals

1.2.1 Goal 1: Implement BVP related data structure

A data structure to describe the BVP problem, namely, "BVProblem". It contains the information

$$\begin{aligned}&F(x,y,y',\cdots,y^{(n)}) = 0\\&\text{domain: } x\in[a,b]\\&\text{boundary condition: Dirichlet, Neumann, Robin.}\end{aligned}$$

The data structure can be created by

$$\text{prob} = \text{BVProblem}(f, domain, bc)$$

.

```
abstract AbstractBVProblem{dType,bType,isinplace,F} <: DEProblem

type BVProblem{dType,bType,initType,F} <: AbstractBVProblem{dType,bType,F}
  f::F
  domain::dType
  bc::bType   #boundary condition
  init::initType
end

function BVProblem(f,domain,bc,init=nothing)
  BVProblem{eltype(domain),eltype(bc),eltype(init),typeof(f)}(f,domain,bc,init)
end
```

In addition, I need to add a data structure for boundary conditions, which let users to define a boundary condition easily. The following is a template:

```
abstract AbstractBoundaryCondition

type DirichletBC <: AbstractBoundaryCondition; end
type NeumannBC <: AbstractBoundaryCondition; end
type RobinBC <: AbstractBoundaryCondition; end
```

1.2.2 Goal 2: Implement shooting method

The shooting method is a method that converts a BVP problem into an IVP problem and a root finding problem. Generally, the shooting method is efficient in simple problems because it does not need a discretization matrix. This reduces the memory overhead. The drawback is that even if the BVP problem is well-conditioned, the root finding problem that the BVP converted to can be ill-conditioned. Therefore, a more robust method like the collocation method is also needed, despite the fact that the shooting shooting method is easy to implement.

This is my current implementation of the shooting method. I will work on it to generalize it later, e.g. let the user have the ability to input a ODE solver and a minimization algorithm.

```
function solve(prob::BVProblem; OptSolver=LBFGS())
  bc = prob.bc
  u0 = bc[1]
  len = length(bc[1])
  probIt = ODEProblem(prob.f, u0, prob.domain)
  function loss(minimizer)
    probIt.u0 = minimizer
    sol = DifferentialEquations.solve(probIt)
    norm(sol[end]−bc[2])
  end
  opt = optimize(loss, u0, OptSolver)
  probIt.u0 = opt.minimizer
  opt.minimum
  DifferentialEquations.solve(probIt)
end
```

1.2.3 Goal 3: Implement collocation method

The collocation method is the idea that a solution $y(x)$ of a ODE $F(x, y, y', \cdots, y^{(n)}) = 0$ can be approximated by a linear combination of basis functions.

$$y(x) \approx \hat{y}(x) = \phi_0 + \sum_{i=1}^{n} a_i \phi_i(x)$$

And the residual $R(x, a)$ can be written as

$$F(x, \hat{y}(x), \hat{y'}(x), \cdots, \hat{y}^{(n)}(x)) = R(x, a)$$

.

Collocation method forces the residual $R(x, a)$ to be 0 for n collocation points. There are different discretization methods in collocation method, and I am going to work on Simpson discretization, Gauss discretization, Radau discretization, and Lobatto discretization this summer. The "discretization" is really a matrix A that applies different quadrature rules, e.g. Simpson's rule to a vector $\vec{x}$. It forms a sparse linear system $A\vec{x} = \vec{b}$, where $\vec{b}$ is known by the boundary condition and RHS of the ODE.

A quadrature rule is a numeric method to calculate a well-behaved definite integrals (no singularity). It uses the idea that a function $f(x)$ can be represented by multiplying an orthogonal polynomial $P(x)$ and another function $W(x)$.

$$f(x) = P(x) \cdot W(x)$$

Therefore, a definite integral of a function F can be approximated by a linear combination of orthogonal polynomials p_i and their weights w_i.

$$F = \int_a^b f(x)\, dx = \int_a^b P(x)W(x) \approx \sum_{i=1}^{n} p_i(x) \cdot w_i(x)\, dx$$

There are more details about quadrature in the section 3.

1.3 Stretch Goals and Future Directions

1.3.1 Compatibility with the features of the common interface

There are many features in *JuliaDiffEq* common interface. They are listed in http://docs.juliadiffeq.org/latest/basics/common_solver_opts.html, and I plan to implement the following:

- Continuous output
- Singularity handling
- *Progressbars* for linear ODEs
- Adaptivity

1.3.2 Weighted residual method

Unlike the collocation method which forces the residual to be zero at a finite number of collection points, the weighted residual method minimizes the residual over the entire interval of integration.

1.4 Relationship with existing quadrature projects

There are some existing quadrature projects in Julia ecosystem, e.g. *QuadGK.jl* and *FastGaussQuadrature.jl*. Although they can calculate the quadrature, those packages are not particularly helpful for the collocation method. The key for collocation method is to form a discretization and apply the quadrature rules to the discretization by using a matrix (potentially sparse matrix) A, which converts a differential equation to a linear system in the form of $A\vec{x} = \vec{b}$.

1.5 Testing

The testing problems for this project will use some ODE BVP problems with analytical solutions. After I implement the collocation method, I will add more sophisticated without analytical solutions. Although there are testing problems that do not have analytical solutions, I can still make sure that I get the right results by comparing my solutions with solutions from the current existing robust solver (e.g. *bvp4c* and *bvp5c*) with very low tolerance. Here is an testing example:

```julia
using BoundaryValueDiffEq
using Base.Test

function f(t, y, du)
  (x, v) = y
  du[1] = v
  du[2] = -x
end

bc = [[0., 1.], [1., 1.]]
domin = (0.,100.)
bvp = BVProblem(f, domin, bc)
@test solve(bvp)[end] == bc[2]

function lorenz(t,u,du)
 du[1] = 10.0(u[2]-u[1])
 du[2] = u[1]*(28.0-u[3]) - u[2]
 du[3] = u[1]*u[2] - (8/3)*u[3]
end

bc_lorenz = [[1.0,0.0,0.0],[-10.,-11.,29.]]

lorenzprob = BVProblem(lorenz, (0.0,1.0), bc_lorenz)
@test norm(solve(lorenzprob)[end] - bc_lorenz[2]) < 0.5 # The tolerance here is big, because
                          # I only implemented a very simple
                          # solver that uses the shooting method,
                          # which is not robust enough.
```

1.6 Timeline

Pre-GSoC I am currently working on function of matrix (a.k.a. matrix function) in Julia Lab. At this period, I can get more experience with working in a big project and sharpen my Julia/Git skills.

Community Bonding: May 5 (Start) - May 30

- Learn more about Julia's type system and the coding style in *JuliaDiffEq*.
- Get more proficient in the Git version control system.
- Learn more about the collocation method and optimization methods that will be used in the shooting method.
- Send a pull request.

Shooting Method: May 30 - June 15

- Design a general framework for BVP problems. (e.g. *BVProblem* and *BoundaryCondition* data structure.)
- Generalize the shooting method.
- Test shooting method with some simple problems with analytical solutions.
- Learn more deeply about collocation methods.
- Send a pull request.

Discretization Algorithms: June 15 - July 10

- Design a basic framework for different types of discretization for solving BVP problems.
- Implement different kinds of discretization algorithms for collection methods.
- Optimize those discretization algorithms that are implemented with *SIMD* and some other techniques.
- Send a pull request.

Collocation Method: July 10 - July 31

- Implement the collocation method.
- Add more sophisticated testing BVP problems to test against the collocation method.
- Optimize the collocation method.
- Send a pull request.

Documentation: July 31 - August 15

- Write a documentation page for the BVP solvers that I have written.
- Add more tests for BVP problem and tune the algorithm.
- Send a pull request.

Review & Stretch Goals: August 15 - August 29 (End)

- Start to work on the stretch goals.
- Send a pull request.
- Review and test all the code that I have written in GSoC project.

1.7 Potential Hurdles

The potential hurdles I see are mostly because I have not worked in a big project like this before, and I may need to put in some effort to be familiar with the coding style in *JuliaDiffEq* organization. I need to be proficient with Git. I also need to be more fluent in Julia's type system. I used to work in linear algebra which does not require much familiarity about the software engineering side. To overcome these hurdles I am going to put in effort and time to learn Julia's type system and Git as I planed in 1.6.

1.8 Mentor

My mentor will be Christopher Rackauckas.

1.9 Julia Coding Demo

Here is a selection of GitHub repositories and other Julia projects I have contributed code towards:
https://github.com/JuliaDiffEq/BoundaryValueDiffEq.jl
https://github.com/obiajulu/ODE.jl/tree/radau Worked with with Joseph Obiajulu.
https://github.com/YingboMa/BVP.jl
https://github.com/YingboMa/Funm.jl

1.10 About me

My name is Yingbo Ma, and I am currently a senior at Lexington High School. I have been admitted by University of California, Irvine (UCI). I am interested in mathematics and physics and willing to learn new things about them. I worked with Joseph Obiajulu on *ODE.jl* last summer in the MIT Julia Lab. I still go to Julia Lab regularly now, and now I am working on fixing the *logm* function in Julia base.

1.11 Contact Information

Email: mayingbo5@gmail.com

GitHub: YingboMa

2 Summer Logistics

Work hours: I expect to be able to work over 35 hours per week throughout the summer. I do not have much other things to do besides working in this project, so I can put most of my attention toward it. Overall, I am able to put 400-500 hours into this project.

3 Quadrature

3.1 Simpson Quadrature

Simpson's rule is a three-point Newton-Cotes quadrature rule. Its formula is

$$\int_a^b f(x)\,dx \approx \tfrac{b-a}{6}\left[f(a) + 4f\left(\tfrac{a+b}{2}\right) + f(b)\right].[1]$$

The error for the Simpson's rule is

$$\frac{1}{90}\left(\frac{b-a}{2}\right)^5 \left|f^{(4)}(\xi)\right|,$$

where the $\xi \in (a,b)$ by Lagrange error bound. [2]
Simpson's 3/8 rule is

$$\int_a^b f(x)\,dx \approx \tfrac{3h}{8}\left[f(x_0)+3f(x_1)+3f(x_2)+2f(x_3)+3f(x_4)+3f(x_5)+2f(x_6)+...+f(x_n)\right], [3]$$

where $h = (b-a)/h$ and $x_i = a + ih$.

3.2 Gauss–Legendre quadrature

The Gauss–Legendre quadrature's weight is

$$w_i = \frac{2}{\left(1-x_i^2\right)[P_n'(x_i)]^2} = \frac{2\left(1-x_i^2\right)}{((n+1))^2\,[P_{n+1}(x_i)]^2}[4]$$

3.3 Lobatto Quadrature

The Lobatto quadrature on the interval $[-1, 1]$ is

$$\int_{-1}^1 f(x)\,dx = \frac{2}{n(n-1)}[f(1)+f(-1)] + \sum_{i=2}^{n-1} w_i f(x_i) + R_n, [5]$$

with the weights

$$w_i = \frac{2}{n(n-1)[P_{n-1}(x_i)]^2}, \qquad x_i \neq \pm 1.[5]$$

P_n is the nth order Legendre polynomial, and it can be written as

$$P_n(x) = 2^n \cdot \sum_{k=0}^n x^k \binom{n}{k}\binom{\frac{n+k-1}{2}}{n}.$$

References

[1] I.P. Mysovskikh. Simpson formula. Last visited on 04/01/2017.

[2] David Süli, Endre & Mayers. *Fundamental Principles of Optical Lithography*. 2003.

[3] Eric W. Weisstein. Simpson's 3/8 rule. From MathWorld—A Wolfram Web Resource. Last visited on 04/01/2017.

[4] Eric W. Weisstein. Legendre-gauss quadrature. From MathWorld—A Wolfram Web Resource. Last visited on 04/01/2017.

[5] Eric W. Weisstein. Lobatto quadrature. From MathWorld—A Wolfram Web Resource. Last visited on 04/01/2017.