DifferentialEquations.jl Workshop Exercises

Chris Rackauckas

July 15, 2019

These excersies teach common workflows which involve DifferentialEquations.jl. The designation (B) is for "Beginner", meaning that a user new to the package should feel comfortable trying this exercise. An exercise designated (I) is for "Intermediate", meaning the user may want to have some previous background in DifferentialEquations.jl or try some (B) exercises first. The additional (E) designation is for "Experienced", which are portions of exercises which may take some work.

The exercises are described as follows:

- Exercise 1 takes the user through solving a stiff ordinary differential equation and using the ModelingToolkit.jl to automatically convert the function to a symbolic form to derive the analytical Jacobian to speed up the solver. The same biological system is then solved with stochasticity, utilizing EnsembleProblems to understand 95% bounds on the solution. Finally, a Bayesian parameter estimation is performed to estimate the true parameters from given data.
- Exercise 2 takes the user through defining hybrid delay differential equation, that is a differential equation with events, and using automatic differentiation to to perform gradient-based parameter estimation.
- Exercise 3 takes the user through differential-algebraic equation (DAE) modeling, the concept of index, and using both mass-matrix and implicit ODE representations.
- Exercise 4 takes the user through optimizing a PDE solver, utilizing automatic sparsity pattern recognition, automatic conversion of numerical codes to symbolic codes for analytical construction of the Jacobian, preconditioned GMRES, and setting up a solver for IMEX and GPUs, and compute adjoints of PDEs.
- Exercise 5 focuses on a chaotic orbit, utilizing parallel ensembles across supercomputers and GPUs to quickly describe phase space.
- Exercise 6 takes the user through training a neural stochastic differential equation, using GPU-accleration and adjoints through Flux.jl's neural network framework to build efficient training codes.

This exercise worksheet is meant to be a living document leading new users through a deep dive of the DifferentialEquations.jl feature set. If you further suggestions or want to contribute new problems, please open an issue or PR at the DiffEqTutorials.jl repository.

1 Problem 1: Investigating Sources of Randomness and Uncertainty in a Stiff Biological System (B)

In this problem we will walk through the basics of simulating models with DifferentialEquations.jl. Let's take the Oregonator model of the Belousov-Zhabotinskii chemical reaction system. This system describes a classical example in non-equilibrium thermodynmics and is a well-known natural chemical oscillator.

1.1 Part 1: Simulating the Oregonator ODE model

When modeling, usually one starts off by investigating the deterministic model. The deterministic ODE formulation of the Oregonator is given by the equations

$$\frac{dx}{dt} = s(y - xy + x - qx^2) \tag{1}$$
$$\frac{dy}{dt} = (-y - xy + z)/s \tag{2}$$
$$\frac{dz}{dt} = w(x - z) \tag{3}$$

with parameter values $s = 77.27$, $w = 0.161$, and $q = 8.375 \times 10^{-6}$, and initial conditions $x(0) = 1$, $y(0) = 2$, and $z(0) = 3$. Use the tutorial on solving ODEs to solve this differential equation on the timespan of $t \in [0, 360]$ with the default ODE solver. To investigate the result, plot the solution of all components over time, and plot the phase space plot of the solution (hint: use `vars=(1,2,3)`). What shape is being drawn in phase space?

1.2 Part 2: Investigating Stiffness

Because the reaction rates of `q` vs `s` is very large, this model has a "fast" system and a "slow" system. This is typical of ODEs which exhibit a property known as stiffness. Stiffness changes the ODE solvers which can handle the equation well. Take a look at the ODE solver page and investigate solving the equation using methods for non-stiff equations (ex: `Tsit5`) and stiff equations (ex: `Rodas5`).

Benchmark using $t \in [0, 50]$ using `@btime` from BenchmarkTools.jl. What happens when you increase the timespan?

1.3 (Optional) Part 3: Specifying Analytical Jacobians (I)

Stiff ODE solvers internally utilize the Jacobian of the ODE system in order to improve the stepsizes in the solution. However, computing and factorizing the Jacobian is costly, and thus it can be beneficial to provide the analytical solution.

Use the ODEFunction definition page to define an `ODEFunction` which holds both the OREGO ODE and its Jacobian, and solve using `Rodas5`.

1.4 (Optional) Part 4: Automatic Symbolicification and Analytical Jacobian Calculations

Deriving Jacobians by hand is tedious. Thankfully symbolic mathematical systems can do the work for you. And thankfully, DifferentialEquations.jl has tools to automatically convert numerical problems into symbolic problems to perform the analysis on!

follow the ModelingToolkit.jl README to automatically convert your ODE definition to its symbolic form using `modelingtoolkitize` and calculate the analytical Jacobian. Use the compilation functions to build the `ODEFunction` with the embedded analytical solution.

1.5 Part 5: Adding stochasticity with stochastic differential equations

How does this system react in the presense of stochasticity? We can investigate this question by using stochastic differential equations. A stochastic differential equation formulation of this model is known as the multiplicative noise model, is created with:

$$dx = s(y - xy + x - qx^2)dt + \sigma_1 x dW_1 \tag{4}$$
$$dy = \frac{-y - xy + z}{s}dt + \sigma_2 y dW_2 \tag{5}$$
$$dz = w(x - z)dt + \sigma_3 z dW_3 \tag{6}$$

with $\sigma_i = 0.1$ where the `dW` terms describe a Brownian motion, a continuous random process with normally distributed increments. Use the tutorial on solving SDEs to solve simulate this model. Then, use the `EnsembleProblem` to generate and plot 100 trajectories of the stochastic model, and use `EnsembleSummary` to plot the mean and 5%-95% region over time.

Try solving with the `ImplicitRKMil` and `SOSRI` methods. Notice that it isn't stiff every single time!

(For fun, see if you can make the Euler-Maruyama `EM()` method solve this equation. This requires a choice of `dt` small enough to be stable. This is the "standard" method!)

1.6 Part 6: Gillespie jump models of discrete stochasticity

When biological models have very few particles, continuous models no longer make sense, and instead using the full discrete formulation can be required to accuracy describe the dynamics. A discrete differential equation, or Gillespie model, is a continuous-time Markov chain with Poisson-distributed jumps. A discrete description of the Oregonator model is given by a chemical reaction systems:

```
A+Y -> X+P
X+Y -> 2P
A+X -> 2X + 2Z
2X  -> A + P (note: this has rate kX^2!)
B + Z -> Y
```

where reactions take place at a rate which is propoertional to its components, i.e. the first reaction has a rate `k*A*Y` for some `k`. Use the tutorial on Gillespie SSA models to implement

the `JumpProblem` for this model, and use the `EnsembleProblem` and `EnsembleSummary` to characterize the stochastic trajectories.

For what rate constants does the model give the oscillatory dynamics for the ODE approximation? For information on the true reaction rates, consult the original paper.

1.7 Part 7: Probabilistic Programming / Bayesian Parameter Estimation with DiffEqBayes.jl + Turing.jl (I)

In many casees, one comes to understand the proper values for their model's parameters by utilizing data fitting techniques. In this case, we will use the DiffEqBayes.jl library to perform a Bayesian estimation of the parameters. For our data we will the following potential output:

```
t = 0.0:1.0:30.0
data = [1.0 2.05224 2.11422 2.1857 2.26827 2.3641 2.47618 2.60869 2.7677 2.96232 3.20711
    3.52709 3.97005 4.64319 5.86202 9.29322 536.068 82388.9 57868.4 1.00399 1.00169
    1.00117 1.00094 1.00082 1.00075 1.0007 1.00068 1.00066 1.00065 1.00065 1.00065
        2.0 1.9494 1.89645 1.84227 1.78727 1.73178 1.67601 1.62008 1.56402 1.50772
    1.45094 1.39322 1.33366 1.2705 1.19958 1.10651 0.57194 0.180316 0.431409 251.774
    591.754 857.464 1062.78 1219.05 1335.56 1419.88 1478.22 1515.63 1536.25 1543.45
    1539.98
        3.0 2.82065 2.68703 2.58974 2.52405 2.48644 2.47449 2.48686 2.52337 2.58526
    2.67563 2.80053 2.9713 3.21051 3.5712 4.23706 12.0266 14868.8 24987.8 23453.4
    19202.2 15721.6 12872.0 10538.8 8628.66 7064.73 5784.29 4735.96 3877.66 3174.94
    2599.6]
```

Follow the exmaples on the parameter estimation page to perform a Bayesian parameter estimation. What are the most likely parameters for the model given the posterior parameter distributions?

Use the `ODEProblem` to perform the fit. If you have time, use the `EnsembleProblem` of `SDEProblem`s to perform a fit over averages of the SDE solutions. Note that the SDE fit will take significantly more computational resources! See the GPU parallelism section for details on how to accelerate this.

1.8 (Optional) Part 8: Using DiffEqBiological's Reaction Network DSL

DiffEqBiological.jl is a helper library for the DifferentialEquations.jl ecosystem for defining chemical reaction systems at a high leevel for easy simulation in these various forms. Use the descrption from the Chemical Reaction Networks documentation page to build a reaction network and generate the ODE/SDE/jump equations, and compare the result to your handcoded versions.

2 Problem 2: Fitting Hybrid Delay Pharmacokinetic Models with Automated Responses (B)

Hybrid differential equations are differential equations with events, where events are some interaction that occurs according to a prespecified condition. For example, the bouncing ball

is a classic hybrid differential equation given by an ODE (Newton's Law of Gravity) mixed with the fact that, whenever the ball hits the floor (`x=0`), then the velocity of the ball flips (`v=-v`).

In addition, many models incorporate delays, that is the driving force of the equation is dependent not on the current values, but values from the past. These delay differential equations model how individuals in the economy act on old information, or that biological processes take time to adapt to a new environment.

In this equation we will build a hybrid delayed pharmacokinetic model and use the parameter estimation techniques to fit this it to a data.

2.1 Part 1: Defining an ODE with Predetermined Doses

First, let's define the simplest hybrid ordinary differential equation: an ODE where the events take place at fixed times. The ODE we will use is known as the one-compartment model:

$$\frac{d[Depot]}{dt} = -K_a[Depot] + R \tag{7}$$

$$\frac{d[Central]}{dt} = K_a[Depot] - K_e[Central] \tag{8}$$

with $t \in [0, 90]$, $u_0 = [100.0, 0]$, and $p = [K_a, K_e] = [2.268, 0.07398]$.

With this model, use the event handling documentation page to define a `DiscreteCallback` which fires at `t` $\in$ `[24,48,72]` and adds a dose of 100 into `[Depot]`. (Hint: you'll want to set `tstops=[24,48,72]` to force the ODE solver to step at these times).

2.2 Part 2: Adding Delays

Now let's assume that instead of there being one compartment, there are many transit compartment that the drug must move through in order to reach the central compartment. This effectively delays the effect of the transition from `[Depot]` to `[Central]`. To model this effect, we will use the delay differential equation which utilizes a fixed time delay τ:

$$\frac{d[Depot]}{dt} = -K_a[Depot](t) \tag{9}$$

$$\frac{d[Central]}{dt} = K_a[Depot](t-\tau) - K_e[Central] \tag{10}$$

where the parameter $\tau = 6.0$. Use the DDE tutorial to define and solve this delayed version of the hybrid model.

2.3 Part 3: Automatic Differentiation (AD) for Optimization (I)

In order to fit parameters (K_a, K_e, τ) we will want to be able to calculate the gradient of the solution with respect to the initial conditions. One way to do this is via Automatic

Differentition (AD). For small numbers of parameters (<100), it is fastest to use Forward-Mode Automatic Differentition (even faster than using adjoint sensitivity analysis!). Thus for this problem we will make use of ForwardDiff.jl to use Dual number arithmetic to retrive both the solution and its derivative w.r.t. parameters in a single solve.

Use the information from the page on local sensitvity analysis to define the input dual numbers, solve the equation, and plot both the solution over time and the derivative of the solution w.r.t. the parameters.

2.4 Part 4: Fitting Known Quantities with DiffEqParamEstim.jl + Optim.jl

Now let's fit the delayed model to a dataset. For the data, use the array

```
t = 0.0:12.0:90.0
data = [100.0 0.246196 0.000597933 0.24547 0.000596251 0.245275 0.000595453 0.245511
         0.0 53.7939 16.8784 58.7789 18.3777 59.1879 18.5003 59.2611]
```

Use the parameter estimation page to define a loss function with `build_loss_objective` and optimize the parameters against the data. What parameters were used to generate the data?

2.5 Part 5: Implementing Control-Based Logic with Continuous-Callbacks (I)

Now that we have fit our delay differential equation model to the dataset, we want to start testing out automated treatment strategies. Let's assume that instead of giving doses at fixed time points, we invent a wearable which monitors the patient and administers a dose whenever the internal drug concentration falls below 25. To model this effect, we will need to use `ContinuousCallbacks` to define a callback that triggers when `[Central]` falls below the threshold value.

Use the documentation on the event handling page to define such a callback, and plot the solution over time. How many times does the auto-doser administer a dose? How much does this change as you change the delay time τ?

2.6 Part 6: Global Sensitivity Analysis with the Morris and Sobol Methods

To understand how the parameters effect the solution in a global sense, one wants to use Global Sensitivity Analysis. Use the GSA documentation page perform global sensitivity analysis and quantify the effect of the various parameters on the solution.

3 Problem 3: Differential-Algebraic Equation Modeling of a Double Pendulum (B)

Differential-Algebraic Equaton (DAE) systems are like ODEs but allow for adding constraints into the models. This problem will look at solving the double penulum problem with enforcement of the rigid body constraints, requiring that the total distance `L` is constant throughout the simulation. While these equations can be rewritten in an ODE form, in many cases it can be simpler to solve the equation directly with the constraints. This tutorial will cover both the idea of index, how to manually perform index reduction, and how to make use of mass matrix and implicit ODE solvers to handle these problems.

3.1 Part 1: Simple Introduction to DAEs: Mass-Matrix Robertson Equations

A mass-matrix ordinary differential equation (ODE) is an ODE where the left-hand side, the derivative side, is multiplied by a matrix known as the mass matrix. This is described as:

$$Mu' = f(u, p, t)$$

where M is the mass matrix. When M is invertible, there is an ODE which is equivalent to this formulation. When M is not invertible, this can have a distinctly different behavior and is as Differential-Algebraic Equation (DAE).

Solve the Robertson DAE:

$$\frac{dy_1}{dt} = -0.04y_1 + 10^4 y_2 y_3 \tag{11}$$

$$\frac{dy_2}{dt} = 0.04y_1 - 10^4 y_2 y_3 - 3 \times 10^7 y_2^2 \tag{12}$$

$$1 = y_1 + y_2 + y_3 \tag{13}$$

with $y(0) = [1, 0, 0]$ and $dy(0) = [-0.04, 0.04, 0.0]$ using the mass-matrix formulation and `Rodas5()`. Use the ODEProblem page to find out how to declare a mass matrix.

(Hint: what if the last row has all zeros?)

3.2 Part 2: Solving the Implicit Robertson Equations with IDA

Use the DAE Tutorial to define a DAE in its implicit form and solve the Robertson equation with IDA. Why is `differential_vars = [true,true,false]`?

3.3 Part 3: Manual Index Reduction of the Single Pendulum

3.4 Part 4: Single Pendulum Solution with IDA

3.5 Part 5: Solving the Double Penulum DAE System

4 Problem 4: Performance Optimizing and Parallelizing Semilinear PDE Solvers (I)

This problem will focus on implementing and optimizing the solution of the 2-dimensional Brusselator equations. The BRUSS equations are a well-known highly stiff oscillatory system of partial differential equations which are used in stiff ODE solver benchmarks. In this tutorial we will walk first through a simple implmentation, then do allocation-free implementations and looking deep into solver options and benchmarking.

4.1 Part 1: Implementing the BRUSS PDE System as ODEs

The Brusselator PDE is defined as follows:

$$\frac{\partial u}{\partial t} = 1 + u^2 v - 4.4u + \alpha(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}) + f(x, y, t) \tag{14}$$

$$\frac{\partial v}{\partial t} = 3.4u - u^2 v + \alpha(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}) \tag{15}$$

where

$$f(x, y, t) = \begin{cases} 5 & \text{if } (x - 0.3)^2 + (y - 0.6)^2 \leq 0.1^2 \text{ and } t \geq 1.1 \\ 0 & \text{else} \end{cases}$$

and the initial conditions are

$$u(x, y, 0) = 22 \cdot y(1 - y)^{3/2} \tag{16}$$

$$v(x, y, 0) = 27 \cdot x(1 - x)^{3/2} \tag{17}$$

with the periodic boundary condition

$$u(x + 1, y, t) = u(x, y, t) \tag{18}$$

$$u(x, y + 1, t) = u(x, y, t) \tag{19}$$

on a timespan of $t \in [0, 22]$.

To solve this PDE, we will discretize it into a system of ODEs with the finite difference method. We discretize `u` and `v` into arrays of the values at each time point: `u[i,j] = u(i*dx,j*dy)` for some choice of `dx`/`dy`, and same for `v`. Then our ODE is defined with

`U[i,j,k] = [u v]`. The second derivative operator, the Laplacian, discretizes to become the `Tridiagonal` matrix with `[1 -2 1]` and a `1` in the top left and right corners. The nonlinear functions are then applied at each point in space (they are broadcast). Use `dx=dy=1/32`.

You will know when you have the correct solution when you plot the solution at `x=0.25` and see a periodic orbit.

If you are not familiar with this process, see the Gierer-Meinhardt example from the DiffEqTutorials.

Note: Start by doing the simplest implementation!

4.2 Part 2: Optimizing the BRUSS Code

PDEs are expensive to solve, and so we will go nowhere without some code optimizing! Follow the steps described in the the Gierer-Meinhardt example from the DiffEqTutorials to optimize your Brusselator code. Try other formulations and see what ends up the fastest! Find a trade-off between performance and simplicity that suits your needs.

4.3 Part 3: Exploiting Jacobian Sparsity with Color Differentiation

Use the `sparsity!` function from SparseDiffTools to generate the sparsity pattern for the Jacobian of this problem. Follow the documentations on the DiffEqFunction page to specify the sparsity pattern of the Jacobian. Generate an add the color vector to speed up the computation of the Jacobian.

4.4 (Optional) Part 4: Structured Jacobians

Specify the sparsity pattern using a BlockBandedMatrix from BlockBandedMatrices.jl to accelerate the previous sparsity handling tricks.

4.5 (Optional) Part 5: Automatic Symbolicification and Analytical Jacobian

Use the `modelingtoolkitize` function from ModelingToolkit.jl to convert your numerical ODE function into a symbolic ODE function and use that to compute and solve with an analytical sparse Jacobian.

4.6 Part 6: Utilizing Preconditioned-GMRES Linear Solvers

Use the linear solver specification page to solve the equation with `TRBDF2` with GMRES. Use the Sundials documentation to solve the equation with `CVODE_BDF` with Sundials' special internal GMRES. To both of these, use the AlgebraicMultigrid.jl to add a preconditioner to the GMRES solver.

4.7 Part 7: Exploring IMEX and Exponential Integrator Techniques (E)

Instead of using the standard `ODEProblem`, define a `SplitODEProblem` to move some of the equation to the the "non-stiff part". Try different splits and solve with `KenCarp4` to see if the solution can be accelerated.

Next, use `DiffEqArrayOperator` to define part of the equation as linear, and use the `ETDRK4` exponential integrator to solve the equation. Note that this technique is not appropriate for this equation since it relies on the nonlinear term being non-stiff for best results.

4.8 Part 8: Work-Precision Diagrams for Benchmarking Solver Choices

Use the `WorkPrecisionSet` method from DiffEqDevTools.jl to benchmark multiple different solver methods and find out what combination is most efficient. Take a look at DiffEqBenchmarks.jl for usage examples.

4.9 Part 9: GPU-Parallelism for PDEs (E)

Fully vectorize your implementation of the ODE and use a `CuArray` from CuArrays.jl as the initial condition to cause the whole solution to be GPU accelerated.

4.10 Part 10: Adjoint Sensitivity Analysis for Gradients of PDEs

In order to optimize the parameters of a PDE, you need to be able to compute the gradient of the solution with respect to the parameters. This is done through sensitivity analysis. For PDEs, generally the system is at a scale where forward sensitivity analysis (forward-mode automatic differentiation) is no longer suitable, and for these cases one uses adjoint sensitivity analysis.

Rewrite the PDE so the constant terms are parameters, and use the adjoint sensitivity analysis documentation to solve for the solution gradient with a cost function being the L2 distance of the solution from the value 1. Solve with interpolated and checkpointed adjoints. Play with using reverse-mode automatic differentiation vs direct computation of vector-Jacobian products using the `autojacvec` option of the `SensitivityAlg`. Find the set of options most suitable for this PDE.

If you have compute time, use this adjoint to optimize the parameters of the PDE with respect to this cost function.

5 Problem 5: Global Parameter Sensitivity and Optimality with GPU and Distributed Ensembles (B)

In this example we will investigate how the parameters "generally" effect the solution in the chaotic Henon-Heiles system. By "generally" we will use global sensitivity analysis methods

to get an average global characterization of the parameters on the solution. In addition to a global sensitivity approach, we will generate large ensembles of solutions with different parameters using a GPU-based parallelism approach.

5.1 Part 1: Implementing the Henon-Heiles System (B)

The Henon-Heiles Hamiltonian system is described by the ODEs:

$$\frac{dp_1}{dt} = -q_1(1 + 2q_2) \tag{20}$$
$$\frac{dp_2}{dt} = -q_2 - (q_1^2 - q_2^2) \tag{21}$$
$$\frac{dq_1}{dt} = p_1 \tag{22}$$
$$\frac{dq_2}{dt} = p_2 \tag{23}$$

with initial conditions $u_0 = [0.1, 0.0, 0.0, 0.5]$. Solve this system over the timespan $t \in [0, 1000]$

5.2 (Optional) Part 2: Alternative Dynamical Implmentations of Henon-Heiles (B)

The Henon-Heiles defines a Hamiltonian system with certain structures which can be utilized for a more efficient solution. Use the Dynamical problems page to define a `SecondOrderODEProblem` corresponding to the acceleration terms:

$$\frac{dp_1^2}{dt} = -q_1(1 + 2q_2) \tag{24}$$
$$\frac{dp_2^2}{dt} = -q_2 - (q_1^2 - q_2^2) \tag{25}$$

Solve this with a method that is specific to dynamical problems, like `DPRKN6`.

The Hamiltonian can also be directly described:

$$H(p, q) = \frac{1}{2}(p_1^2 + p_2^2) + \frac{1}{2}(q_1^2 + q_2^2 + 2q_1^2 q_2 - \frac{2}{3}q_2^3)$$

Solve this problem using the `HamiltonianProblem` constructor from DiffEqPhysics.jl.

5.3 Part 3: Parallelized Ensemble Solving

To understand the orbits of the Henon-Heiles system, it can be useful to solve the system with many different initial conditions. Use the ensemble interface to solve with randomized initial conditions in parallel using threads with `EnsembleThreads()`. Then, use `addprocs()` to add more cores and solve using `EnsembleDistributed()`. The former will solve using all

of the cores on a single computer, while the latter will use all of the cores on which there are processors, which can include thousands across a supercomputer! See Julia's parallel computing setup page for more details on the setup.

5.4 Part 4: Parallelized GPU Ensemble Solving

Setup the CUDAnative.jl library and use the `EnsembleGPUArray()` method to parallelize the solution across the thousands of cores of a GPU. Note that this will efficiency solve for hundreds of thousands of trajectores.

6 Problem 6: Training Neural Stochastic Differential Equations with GPU acceleration (I)

In the previous models we had to define a model. Now let's shift the burden of model-proofing onto data by utilizing neural differential equations. A neural differential equation is a differential equation where the model equations are replaced, either in full or in part, by a neural network. For example, a neural ordinary differential equation is an equation $u' = f(u, p, t)$ where f is a neural network. We can learn this neural network from data using various methods, the easiest of which is known as the single shooting method, where one chooses neural network parameters, solves the equation, and checks the ODE's solution against data as a loss.

In this example we will define and train various forms of neural differential equations. Note that all of the differential equation types are compatible with neural differential equations, so this is only going to scratch the surface of the possibilites!

6.1 Part 1: Constructing and Training a Basic Neural ODE

Use the DiffEqFlux.jl README to construct a neural ODE to train against the training data:

```
u0 = Float32[2.; 0.]
datasize = 30
tspan = (0.0f0,1.5f0)

function trueODEfunc(du,u,p,t)
    true_A = [-0.1 2.0; -2.0 -0.1]
    du .= ((u.^3)'true_A)'
end
t = range(tspan[1],tspan[2],length=datasize)
prob = ODEProblem(trueODEfunc,u0,tspan)
ode_data = Array(solve(prob,Tsit5(),saveat=t))
```

6.2 Part 2: GPU-accelerating the Neural ODE Process

Use the `gpu` function from Flux.jl to transform all of the calculations onto the GPU and train the neural ODE using GPU-accelerated `Tsit5` with adjoints.

6.3 Part 3: Defining and Training a Mixed Neural ODE

Gather data from the Lotka-Volterra equation:

```
function lotka_volterra(du,u,p,t)
  x, y = u
  α, β, δ, γ = p
  du[1] = dx = α*x - β*x*y
  du[2] = dy = -δ*y + γ*x*y
end
u0 = [1.0,1.0]
tspan = (0.0,10.0)
p = [1.5,1.0,3.0,1.0]
prob = ODEProblem(lotka_volterra,u0,tspan,p)
sol = Array(solve(prob,Tsit5())(0.0:1.0:10.0))
```

Now use the mixed neural section of the documentation to define the mixed neural ODE where the functional form of $\frac{dx}{dt}$ is known, and try to derive a neural formulation for $\frac{dy}{dt}$ directly from the data.

6.4 Part 4: Constructing a Basic Neural SDE

Generate data from the Lotka-Volterra equation with multiplicative noise

```
function lotka_volterra(du,u,p,t)
  x, y = u
  α, β, δ, γ = p
  du[1] = dx = α*x - β*x*y
  du[2] = dy = -δ*y + γ*x*y
end
function lv_noise(du,u,p,t)
  du[1] = p[5]*u[1]
  du[2] = p[6]*u[2]
end
u0 = [1.0,1.0]
tspan = (0.0,10.0)
p = [1.5,1.0,3.0,1.0,0.1,0.1]
prob = SDEProblem(lotka_volterra,lv_noise,u0,tspan,p)
sol = [Array(solve(prob,SOSRI())(0.0:1.0:10.0)) for i in 1:20] # 20 solution samples
```

Train a neural stochastic differential equation $dX = f(X)dt + g(X)dW_t$ where both the drift (f) and the diffusion (g) functions are neural networks. See if constraining g can make the problem easier to fit.

6.5 Part 5: Optimizing the training behavior with minibatching (E)

Use minibatching on the data to improve the training procedure. An example can be found at this PR.