Julia IDE VSCode + Jupyter

davidistreader@gmail.com

Abstract. The Julia Jupyter link is good for demos but lack the IDE so useful for program development hence the move to VSCode.

1 Setting up Julia in VSCode

Start VSCode then install Julia via VSCode. Julia in VSCode is easy to set up so that it only partially functions. The following recipe works - not sure why!

1. Open new empty folder.
2. Create an environment - From command pallet start Julia REPL
3. in REPL(in VSCode): in the repl type "]" to enter the package manger
   (a) *Pkg* >**activate .** activate current directory?
   (b) *Pkg* >**add Plots, Statistics** add packages of interest
4. **link the newly created environment to the editor** by clicking on *Julia 1.7* in the status bar at the very bottom of the VSCode screen and then selecting the environment from the drop down list.
5. the REPL will appear in the task list bottom right of VSCode
6. in the left edge of VSCode click on the Julia icon to see
7. To see plot or the results of BenchmarkTools you must select "Execute active File in REPL" in the drop down list at the top right of screen! And/Or add readline() to end of file. But beware **@benchmark ...** is run in parallel and may take a long time?
8. it is trivial to save as a Local Git repository?
9. the Julia REPL that VSCode spawns will appear both in VSCode and in a new terminal outside VSCode.
10. in the Julia REPL pwd() works as shell pwd. To change from **julia**> to **shell**> type **;** and to get back ˜ **C**

1.1 Some basic navigation tips

In terminal type > **jupyter notebook** and set the kernal to Julia. Julia uses < *tab* > completion within the notebook. Jupiter < *esc* > *l* to show line numbers

Julia package management Pkg: is a repl with in the Juila repl **julia**> Note using Julia within Jupyter is very slightly different from using the Julia REPL.

julia> sin(π)
julia> @edit sin(1) #to go to the Julia implementation of sin OR in Jupyter **?sin**
julia> import Pkg
julia>] # enter the package REPL

pkg> activate myEnv
pkg> add Flux PyPlot Plots
pkg> status # to inspect the environment myEnv
pkg> ^C to return to **julia**>

Package contents *julia >using Flux* for a list of package contents type: *julia >Flux.< tab >* for dropdown list of functions OR for output to screen *julia >Flux.< tab >< tab >* OR *julia >names(Flux)* for function you should use.

Source code from links displayed by:

1. *julia >methods fun* will show the list of all implementations
2. *julia >methods fun(2,3.5)* implementation with these parameters

help *julia >?* to enter the help system.
help? >*gradient* will then list details of the gradient function and exit the help system.

methodswith Type *julia > methodswith(DataFrame)* for list of methods that apply to objects of type *DataFrame*

Search Git Hub Julia Search! fantastic

cheat sheets 1) Julia-Cheat-Sheet and 2) Matlab-Python-Julia

Your function documentation that will show up in jupyter with >*?cont_game*:

```
"""
 cont_game(‘\delta‘, P)
- ‘\delta‘ small x inc
- ‘P‘ prob
‘‘P = \delta^n‘‘
"""
function cont_game(\delta, P)
    return P
end
```

1.2 Julia Efficiency

Julia is JIT compiled, thereby defining functions with UNtyped parameters can provide extreme Modularity of package construction without loss of efficiency. It also means that a function is compiled once for each time it is called with parameters of distinct type thus allowing code optimization over function boundaries. Thus the use of **Val** type parameters makes the value available to the compiler and allows additional efficiency via compile time code optimization.

1. define function with additional parameters over use of global variables
2. break functions into separate functions over conditions on type
3. use inplace updates over allocation of heap variable
4. avoid untyped collections use subtyping **{T¡:AbstractFloat}**
5. declare variables as **const** where applicable
6. to check efficiency us **@time, @allocated, @profile** or **using Traceur**

2 Neural Nets

2.1 Neuron

The ith neuron with j inputs and k outputs has three parameters that can be fixed or learnt:

1. W_{ij} the weight matrix
2. b_i a scalar bias
3. ρ_i an activation function

2.2 Deep Neural network

consist of an input layer, an output layer and a stack of several hidden layers. Supervised learning requires pairs of input vectors and known output vectors. In the forward pass data is input, from which the value of the first hidden layer is computed. Computation continues up the stack of hidden layers until the value of the output is computed. A

4. loss function

is applied between the computed output and the known output vector. The backward pass adjusts the networks parameters with the goal of improving the next forward computation of the output vector.

Other than for the input layer each $j-$neuron has a weight attached to each $i-$input W_{ij}, a scalar bias b_j value and an activation function ρ_j . In the forward pass the input values are multiplied by the weight, the results are summed along with a bias $r_j = \Sigma_{i=1}^{i=n} x_i * W_{ij} + b_j$. This is input to an activation function prior to the result $\rho_j(r_j)$ being output.

For some neural algorithms the activation function is of little significance and can be decided after the the rest of the neural architecture has been fixed. But for other algorithms the selection of the activation function is critical and must be designed along with the rest of the neural architecture. Using appropriate activation functions a NN can be used to compute multiplication and division as well as summation. This enables the NN to to compute many finite element methods, FEM. Further the activation function can be parametrised per neuron or per layer and learnt just like the weights and biases. This has been used in XPINNs. Adaptive activation functions is equivalent to second order approximation bt without computing the Hessian, which is expensive.

The basic Multi Layered Positron,MLP, consists of a sequence of layers with total connection weight matrix between any two adjacent layers.

Although given a large enough MLP and function can be modelled in practice MLPs were of little use until the physics of the problem at hand was embedded in the NN. A MLP can be thought of as having an unstructured topology whereas many more effective NNs have a physics inspired topology.

Convolution NNs were used for image recognition. The Topology of the layers captured: neighbouring neurons-pixels are more relevant than distant pixels; a

feature, face, is a face wherever it appears and at whatever scale. Recurrent Neural Nets, RNNs,sequentially process data such as text. Long Short Term Memory NNs, LSTM, are an improvement on RNNs. Hierarchical Neural Nets capture abstract and more concret e understanding. Attention Neural Nets mimic the flexibility of cognitive attention. They have three components an encoder and a decoder with an attention unit between them. Transformer Neural Nets, like RNNs and LSTMs process sequential data but unlike RNNs and LSTMs they process the data in parallel using attention to select the relevance of the data.

The integration of Physics and Neural networks has proven very effective.Physics can be encoded within Neural Networks in the:

Topology of the network such as CNN and Recurrent NN

the Loss function such as PINN. Using automatic differentiation we can evaluate the model represented by an ODE. Hence the Loss function includes not only L_D the difference between the NN computed value and the training data but also L_R, the difference between the NN computed value and the value computed by the ODE.

by generating training data L_R can be used in place of L_D, if you have the data thats good if not you can always fake it.

Solving Inverse Problems with Deep Learning by Lexing Ying

When and why physics-informed neural networks fail to train by Paris Perdikaris looks at when and why failure occurs.

2.3 Complex Valued NN

Although the flexibility of NN can be greatly beneficial it can also be the cause of problems Using Complex values in place of real values can limit this flexibility, preventing infeasible solutions from being considered and reducing over fitting. See **A Survey of Complex-Valued Neural Networks**. CVNN are not the same a a two dimensional real NN but are good fro modelling data as amplitude, phase pairs.

3 Finite Element Methods

FEM solves problems over N dimensional continuous domain by applying a discrete mesh over the domain, solving for the points on the mesh and interpolating between the points. The accuracy of many methods can be improved by adjusting the mesh to concentrate on specific regions, *r-adaptivity* or by increasing the the number of points in specific regions *h-adaptivity.*

3.1 NN-FEM

Encoding the mesh points as the bias on the input layer means that training will automatically adjust the mesh to the desired regions , automatic r-adaptivity. Mesh points are Neurons and are elements in an array hence mesh points can be

added at locations where they are most needed simply by adding elements to the array. This is h-adaptivity. With FEM-NN rh-adaptivity is learnt not designed by hand.

4 Parallel Computing and Scientific Machine Learning

Many thanks for open source MIT course $18.337J/6.338J$: gores over how to build optimized libraries that can be used. So covers a lot of underlying theory. Scientific Machine Learning is the use of Machine Learning to the solution of the simulation of Scientific problems. Machine learning on data is simply this problem of finding an approximation to some unknown function!

Neural Nets, Talyor series and Fourier series are all universal function appropriators that can be used with loss function in standard parameter optimisation procedures. But Neural Nets do not scale exponentially with the dimension of the input. In theory given enough data and a large enough neural net you can solve any problem. In practice it is necessary to encode as much of the behaviour of the system as you can into the neural net prior to training it, for example convolution neural nets. Many disciplines use differential equations to capture some of the behaviour of the systems they are modelling.

Differential equations describe the behaviour whereas neural nets learns the behaviour from the data. ODEs are better at extrapolation than neural nets and are interpretable.

Neural Ordinary Differential Equations $u'(t) = f(u, p, t)$ where f is a neural network.

Dynamic systems change over time and can be defined as a set of differential equations For example the N body problem is defined by N ordinary differential equations, ODEs, $F = MA$ where A - acceleration - is the second order differential of position over time. A solution to these differential equation is a function from time to position.

ODEs are of the form $\frac{du}{dt} = f(u, p, t)$. When there is a vector $u = (u_1, u_2, \dots u_N)$ then $\frac{du}{dt}$ is the Jacobian. And the Jacobian, of such an ODE is defined in terms of partial derivatives.

PDEs, Partial Differential Equations are a generalisation of ODEs that allow terms such as $\frac{\delta u_i}{dt} \frac{\delta u_j}{dt}$ etc.

ODEs can be solved, the explicit function $u(t)$ constructed, for a given vector of the parameters p. The solution is frequently best understood by plotting the result, $u(t)$ plot. Another useful way to understand a dynamical system is the bifurcation plot.

4.1 Algebraic Dynamics

AlgebraicDynamics.jl provides a way to *"wire"* together different models, eg: the flow of people between cities, each with their own SIR infection model.

4.2 Latent Differential Equations

LatentDiffEqu.jl Encode high vector space input (video of pendulum) into low dimension space (to 2) using RNNs or LSTMs then decode back to high vector space output. Use SINDy to convert low dimension space into analytic term.

Physics-Informed ML Simulator for Wildfire Propagation

4.3 Baysian Inference

Turing.jl allows Probabilistic programming. With **Flux.jl** you have Neural ODE models.

4.4 Physics Informed Neural Nets

Machine learning is data driven and can use neural nets to model dynamic systems. These same dynamic systems can be analyticity modelled by differential equations. Physics Informed Neural Nets **PINN** attempts to combine the best of both worlds:

NN Robust to noise
NN Good with high dimensional input
NN Can infer hidden pattern, implicit basis
ODE fast to compute
ODE No input data needed
ODE Error bounds on solution

Automatic differentiation of NN used alonge with ODE encoded in Loss function.

Auto Encoder Decoder that relates high dimensional system, such as video input, to low dimension representation, such as pendulum dynamics of the single θ variable representation can be learnt. Then $\theta' = f(\theta)$ dynamics can be learnt.

4.5 Automatic Differentiation

Given a program the compiler constructs a *primal*, an executable algorithm. Using **Dual Numbers** the AD compiler also construct an executable algorithm for the differentiation of the primal.

Forward mode AD asks the question: how dose the output change when the input changes by some small amount? Forward Mode AD is efficient at computing the gradient when there is one, or a small number, of inputs but is terrible when there is a large number of inputs.

Reverse Mode AD asks the question the other way around. How should the the inputs be changed in order to change the output by a small amount. Reverse Mode AD is efficient at computing the gradient when there is a large number of inputs.

Forward Mode AD is often compared with the forward pass of a neural net and Reverse Mode AD is compared with back propagation and clearly you cannot replace back propagation with a forward pass. Forward and Reverse Mode AD both compute the gradient. But are they the same function or, ignoring efficiency, is Reverse Mode doing something the Forward Mode is not? Alternatively are there applications using Reverse Mode where, ignoring efficiency, Forward Mode could not be used?

The reason for asking this is that the paper **Provably Correct, Asymptotically Efficient, Higher-Order Reverse-Mode Automatic Differentiation** defines Reverse Mode AD as a correct and efficient way to compute the gradient when there is a large number of inputs. Yet this is computing the exact same thing as Froward Mode AD.

4.6 Lecture 2

JIT compilation slows down first execution but allows modular development of libraries. Multiple dispatch and being a high level language allows aggressive code optimisation.

For Julia efficiency

1. only benchmark functions and always use functions
2. Avoid explicit typing - no overhead, $\mathbb{O}_{\mathbb{C}}$, with JIT compilation
3. a $@viewx[i:j]$ is a pointer to part of an array
4. Avoid heap allocation - for small size data use static vectors for large size arrays allocate prior to iteration (use mutating functions) and be aware of index order as arrays are implemented as single index linear structures.
5. broadcast and fuse operations (bounds checking not needed in loop)
6. function calls are expensive on parallel code

4.7 Lecture 3 physics informed neural networks

Neural nets are composed of many layers each parametrised on a Weight matrix, a Basis Vector and an activation function. Neural nets are composed of a stack of layers and a loss function. They are Universal function Approximators. Training a neural net requires know inputs and outputs and proceeds

1. a forward pass from input to output
2. an evaluation via the loss function
3. a correction of the parameters of each layer by back propagation

Other Universal function Approximators include polynomial expansion and the Fourier series. These two suffer from the curse of dimensionality where as NNs do not.

Physics, biology and chemistry all make use of partial differential equations to model complex dynamic systems. Hence the interest in their solution, simulation and analysis. Simple PDEs can solved in NNs by encoding the PDE in the loss function.

4.8 Lecture 4 Discrete Dynamic Systems, How Loops work

Systems that evolve in time where time is taken in a finite number of discrete steps.

$$u_{n+1} = \alpha u_n + \epsilon_n$$

With delays

$$u_{n+1} = \Sigma_{j=0}^{j=k} \alpha_j u_{n-j} + \epsilon_n$$

For scalar linear dynamical system $u_{n+1} = \alpha u_n$ we have $u_{n+1} = \alpha^n u_0$ and hence

1. $\|\alpha\| < 1$ implies stability, $u_n \to 0$
2. $\|\alpha\| > 1$ implies divergence, $u_n \to \infty$
3. $\|\alpha\| = 1$ implies $u_n \to u_0$ and complex dynamics in the complex plane

Understanding Non Linear dynamic systems

1. **Stability:**
   For such systems **Banach Theorem**: for metric space (M, d) if ff is contracting $d(f(x), f(y)) < d(x, y)$ then there exists a unique fixed point (x^* such that $f(x^*) = x^*$) and a sequence $x_{n+1} = f(x_n)$ such that $x_n \to x^*$
   If the function f of a non linear system $x_{n+1} = f(x_n)$ is *nice* (f' is continuous) then: when $f'(x) < 1$ the system is **stable** (when perturbed by a small amount the sequence still returns to the fixed point x^*).
   Multi variable systems: let $x \in \mathbb{R}^3$ then in $x_{n+1} = f(x_n)$ the function f is a vector of functions. The linear version $x_{n+1} = Ax_n$ has a matrix of constants A. The systems to analyse have diagonalizable matrices $A = PDP^{-1}$. This includes many systems of scientific interest. Stability when $\|D\|$, all eigenvalues within the unit circle. Solutions along the eigenvectors are independent of each other.
2. **Periodicity:** 1. The length of the period. 2. Stability implies that points near the periodic orbit are attracted to the periodic orbit.
3. **Chaos**

4.9 Lecture 7 Ordinary Differential Equations

General form

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

used to define Continuous Dynamic Systems

$$u(t) = \int_{t_0}^{t_f} f(u, p, t)$$

To solve the Lorenz equations: these are three ODEs

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y - x) \\ \frac{dy}{dt} &= x(\rho - z) - y \\ \frac{dz}{dt} &= xy - \beta z, \end{aligned}$$

the solution, for fixed parameters, σ, ρ, β is three functions $u_x(t), u_y(t)$ and $u_z(t)$ plotting them independently against time gives some insight. Alternatively creating one three dimensional (x, y, z) plot of $(u_x(t), u_y(t), u_z(t))$ we see the classic butterfly attractor.

Examples: Pleiades a 7 star chaotic system, Population Ecology: Lotka-Volterra preditor prey cyclic system, Biochemistry: Robertson Equations are stiff: have periodic behaviour but with large differences in the periods.

Geometric Properties : Linear ODEs: $u' = au$ hasve general[1] solution $u(t) = u(0)e^{\alpha t}$ From this we can see

1. $Re(\alpha) > 0$ $u(t) \to \infty$ as $t \to \infty$
2. $Re(\alpha) < 0$ $u(t) \to 0$ as $t \to \infty$
3. $Re(\alpha) = 0$ $u(t)$ has constant or periodic solution

The real part is the magnitude and the imaginary part the rotation $e^{\alpha+i\beta} = e^{\alpha}e^{i\beta} = e^{\alpha}(sine(\beta) + icos(\beta))$

Steady states for discrete systems are fixed points $f(u^*) = u^*$ and for continous systems are $u'^* = 0$

Multivariable systems :
Linear multivariable systems are represented as a matrix which often can be diagonalized. A diagonal matrix decouples the equation into a system of linear ordinary differential equations which we solve individually. When all the eigenvalues are negative then $u(t) \to 0$ as $t \to \infty$. wnen an eigenvalue is positive then $u(t) \to \infty$ along the eigenvector.

Nonlinear ODEs: geometric properties extend locally: $\frac{df}{du}$ *is the Jacobian*

$$u' = \frac{df}{du}u$$

Numerical solutions of ODEs :
First descritize the equation then solve the discrete equation.

Euler's method

$$\frac{\Delta f}{\Delta t} \approx \frac{df}{dt} = u'(t) = f(u, p, t)$$

this gives a first order approximation. Works well with very small steps.

Intuitively Euler's method can be seen as selecting where, on the interval $t : t + \Delta t$, to evaluate f

$$u_{n+1} = u_n + \Delta t f(u, p, t)$$

[1] solutions to some simple algebraic equations are complex numbers $x^2 + 1 = 0$. Note you should be able to calculate the general solution by remembering that $\int \frac{dt}{t} = log(t)$

Higher order methods: Alternatively you can use Euler's method to estimate the best point on $t : t + \Delta t$, to evaluate f.

The next step is Δt times the derivative, initially evaluate the derivative at $u_n + \frac{\Delta t}{2}$

The Runge Kutter method chains this approach - use the previous derivative approximation $k_1 = f(u, p, t)$ to calculate your next derivative k_2. At each step Euler's method to estimate the best point to evaluate where to evaluate $f(u_?, p, t)$ that is what $u_?$ to use.

As the RK method chains more steps together and is equal to more terms of the Taylor series hence is more accurate or allows larger steps for the same accuracy. But it dose so at an exponentially increasing computational cost. So greater accuracy means higher order methods may be more efficient. The RK computation is dependent upon the selection of both $u_?$ and t it then averages a weighted set of these results.

Computation cost is closely related to cost of the function passed to the ODE solver. Being JIT compiled Julia can inline the function and optimize across the function call.

4.10 Lecture 8 Forward Mode AD via Higher Dimensional Algebras

Previously looked at how to solve, understand, ODEs. Here we look at how to compute and understand Forward Mode AD.

Floating point numbers are not real numbers they are **relatively scaled**. Hence introduce significant *round off error*s when computing derivatives. Note associative and commutative properties fail for floating point numbers because of the rounding errors! Float64 has 16 digit "accuracy" Float32 it's 8 digit.

Dual numbers consist of a pair of numbers, the first hold the value the second holds the derivative. Thus, applying f to $Dual(a, 1)$ should give $Dual(f(a), f'(a))$ and Dual number computation is accurate and has excellent performance. The normal mathematical operators "+", "*", etc are lifted to the known results for derivatives $(f * g)' = f'g + fg'$. Dual is a 2-dimensional number for calculating the derivative without floating point rounding error. **The compiler applies, Forward Mode AD on all programs.**

4.11 Lecture 9 Solving Stiff Ordinary Differential Equations

Stiff , sparce ODEs are common in all the sciences and both stiffness and sparsity influence the efficiency of algorithms used to solve the ODEs. Stiff ODEs have eigan values of very different magnitude. Algebraic Differential Equations can be seen as Stiff ODEs where the stiffness has gone to infinity.

Scientific dynamic systems are often described both at great detail, very small time steps, yet have interesting properties described at much larger time steps. This results in stiff ODEs. How do a million well understood things have some emergent property of interest one set of a million atoms are solid the other are liquid. Solving stiff ODEs to reveal the long time step dynamics with steps

small enough to cope with the small time step dynamics requires either very very many tiny **explicit steps** or fewer more complex **implicit steps**

Non-stiff ODEs can be solved via optimized Runge-Kutta methods whereas stiff ODEs ca be solved using *implicit* methods The implicit Eular method is:

$$u_{n+1} = u_n + \Delta t f(u_{n+1}, p, t + \Delta t)$$

this can be rearranged to $g(u_{n+1}) = 0$ and is the classic root finding problem. Solving via Newtons method

$$x_{n+1} = x_n - J(x_k)^{-1} g(x_k)$$

Computing the inverse of a large matrix is very expensive and the inverse of a sparce matrix is likely to be dense. Hence the direct computation of $J(x_k)^{-1}$ is to be avoided.

To solve this you iteratively:

1. Solve $Ja = g(x_k)$ for a
2. update $x_{k+1} = x_k - a$

By doing this, we can turn the matrix inversion into a problem of a linear solve and then an update.

Step one is to efficiently compute the Jacobian: AD computes the Jacobian but to do this efficiently it uses the sparcity pattern to build the compressed Jacobian and rebuild the expanding Jacobian. The full Jacobian has columns that are the partial differentiation along the basis vectors. Whereas in the compressed Jacobian columns are vectors that the sum of basis vectors. Graph colouring is used to compute the compression. But this graph colouring is np complete! Hence no ideal only guest guess - gready algorithm, weak algorithm, –.

Step two is to solve the Linear: This is the core of linear numerical analysis $Ja = b$ for known J and b find a. Decompose $J = UL$ and solve $L(Ua) = b$ by first solving for Ua then for a.

This is known as a Quasi-Newton method.

Computing LU is the most computationally expensive of solving these ODEs.

An alternative is to not compute the Jacobian but to compute the Jacobian Vector product in one step using AD?. Have a model. Have data. Fit model to data.

4.12 Lecture 10 Basic Parameter Estimation, Reverse-Mode AD, and Inverse Problems

Forward-mode AD was implementable through operator overloading and dual number arithmetic but requires one execution of the program per input variable.

Reverse-Mode AD requires one execution of the program per output. Hence, in some circumstances, can offer significant computational advantage. Whereas forward mode asks how does the output vary when the inputs change, forward chain rule $\frac{\delta w}{\delta t}$; reverse mode asks how must the inputs change to get a change in the outputs, reverse chain rule $\frac{\delta t}{\delta u}$.

Forward-Mode AD as jvp Jacobian vector product and Reverse-Mode AD as vjp vector Jacobian product.

Have a model. Have data. Fit model $f(p) = u$ to data $\{(p_i, u_i).i \in 1..n\}$. This is a problem that goes under many different names: parameter estimation, inverse problems, training, etc.

One method is to use gradient decent on a cost function $C = \|f(p_i) - u_i\|$ to adjust the models parameters:

$$p_{i+1} = p_i - \alpha \frac{dC}{dp}$$

4.13 Lecture 11: Differentiable Programming and Neural Differential Equations

Reverse-Mode AD was shown how to compute gradients in a fast manner given a **computational graph** and performing reverse-mode automatic differentiation. This is an efficient way to calculate Jacobians of objects where there are less rows than columns (think of the gradient as 1 row). But generating the computational graph can be infeasible.

Static computation Graph AD When you can write one (Tensorflow - ??? is based on this) the problem is not so difficult. But it requires rewriting all programs in this style which reduces library modularity.

Tracing-Based AD and Wengert Lists Reverse-mode AD for composed functions is through pullbacks on the Jacobiansalso called "adjoints" but these methods have problems:

Source-to-Source AD

5 Overview SciML

This package covers the overarching strategy. Central to this is the use of *Automatic Differentiation of Julia programs.* Neural nets and Deep learning in particular have proven very successful at solving many problems. Significant disadvantages include

1. they require vast amount of training data
2. they are not as good at extrapolation as at interpolation

Significant improvements in Deep NN occurred when properties of the problem were *encoded* in the NN prior to learning, for example convolution NN.

The automatic differentiation refers to the fact that the Julia compiler builds both an implementation of the functions specified by the code and the differentiation of the function. This permits the solution to physical models define in various forms of Differential Equations. This can be easily extended to parametrised ODEs by preprocessing with relatively simple NN. But, maybe more surprisingly, this can be extended to the common situation where only a partial model is available. Solving this partial problem can be achieved by extending the symbolic partial physical model with NN black boxes for the unknown component.

5.1 Physics Informed Neural Networks, PINN

see **Knowledge Integration into deep learning in dynamical systems: an overview and taxonomy** for 2021 overview. The combination of some model of physics of a problem along with the data observed has proven extremely beneficial. Whereas a Multi Layer Perceptron, with its densely connected layers, requires an infeasibly vast amounts of data for many problems. The knowledge of the physics can be incorporated in various ways:

1. Change the architecture, CNN, RNN, Attention NN, ...
2. add regularisation to the loss function
3. pre train a neural net on data from related problem (PI initialisation) optionally the NN may be extended
4. increase the data volume by using data from a simulation model

5.2 Interpret ability

A symbolic model can be extracted from a NN solution by use of sparse matrices of candidate functions **SINDy** (sparse identification of non-linear dynamics). These symbolic models can have much improved capacity for extrapolation and have proven able to extract known physics results simply from data observations.

6 Differential Equations

A differential equation for a dynamic system has independent variable time t, parameters p and equation $u' = f(u, p, t)$. For some initial condition $u0$ you can use **DifferentialEquations.jl** and its default methods for numeric solution of $u(t)$. Numerical problems arise when outputs dimensions have very different periods. Such equations are referred to as **stiff**.

```
prob = ODEProblem(f, u0, tspan)
sol = solve(prob)
```

`slove(prob,  )` can take Many options including:
`Tsits5, relTol=1e-8, absTol=1e-6,alg_hints=[:stiff],`
Algorithm selection:

1. `Tsit5` for nonstiff equations (stiff - think unbounded derivative at certain points)
2. `Vern7()` and `Vern8()` for low tolerance, less than `1e-8`
3. high tolerance `BS3()`
4. stiff try `Rosenbrock23()`, `Rodsa4()` or `CVODE_BDF`

Pragmatically solve twice once with default and another with hint[:stiff] then choose best. You will need to do this a lot!

6.1 ODE optimization

Small ODE ¡100 equations:

Details are covered in documentation on ODE Optimization but a simple overview is first optimize the ODE by 1. avoiding use of *global variables* 2. avoid variable allocation with in the ODE problem definition 3. inline matrix computation using `@.` macro.

1. make non allocating
2. use static arraies where possible
3.

Large ODE You have to use large Arrays (dynamic not static)

1. use broadcast, Julia streams, `@. sin(x) + x^2`
2. non allocating functions `A_mul_B!(D,A,B)` thsi is particularly useful is D is a cash built once and written to many times in a loop. Remember might be good to pass cash in as parameter to keep function type agnostic.
3. use BLAS Julia uses open BLAS look for sugarBLAS.jl
4. with command `p = @view A[:,:,1]` p is a pointer to part of the array A,
5. choice of algorithm `Tsit5` not good for stiff ODEs Try `Newton Kry...` or `IMAX` .. Try `CVODE_BDF(linearsolver=:GMRES)`

6.2 Callbacks

See the CallBack Library in DifferentialEquations.jl

An Integrator contains the state of the system at the current step, solutions contain the history of the system state.

Continuous callbacks are executed at the actual specified condition whereas DescriteCallbacks are executed at the step when the condition is first satisfied.

Numerical errors can cause the model to *drift* but if known physics defines a manifold that the solution must be on, e.g. conservation of energy, then a ManifoldProjection callback will project the numerical solution back to the manifold.

6.3 ODEs with parameters

To help you can use the `@odf_def` macro

Although Differential Equations is at the center of SciML. The ModelingToolkit is becoming as important partially

6.4 ModelToolkit

does symbolic simplification **structural_simplify**. It provides both Analytic and Numeric Model optimization, Mixed Differnence and Differential equation solvers, transform stocastic Diferential equations into Differential equations of mean and ,

7 Flux

8 Quantum computing is an interesting exemplar

Qiskit by IBM uses python and j?? to produce Quantum assembler code:

Classical bits are used for output and Quantum bits, Qbits, for computation. A single Qbit can be in a superposition state, that is with some probability of being in state 0 or 1. A Qbit posses both *amplitude*, that may be negative, and *phase*. The probability a Qbit is in any state, say 0/1, is the square of the amplitude. Two or more Qbits can be entangled.

Only the probability can be measured, seen, from outside the quantum code. Both the amplitude can phase of a Qbit can be seen by, can influence, another entangled Qbit. In particular only the relative phase differecne between the Qbits can be seen, *sometimes described by saying that "global" phase factors are unphysical, but "relative" phase factors are physical and important.*

A Qbit is represented by a vector of two complex numbers[2] and , by superposition, can be in two primitive states at the same time. The state of an entangled Qbits can be represented by a 2^n vector (of complex numbers). For a vector to be a valid representation the sum of squares must be 1 and only the the relative phase can be observed hence state can be represented by two real numbers ϕ and θ:

$$|q> = cos(\theta/2)|0> + e^{i\phi}sin(\theta/2)|1>$$

Interpreting ϕ and θ as angular coordinates the sate can be represented as a point n the **bloch sphere** with radius 1. *The Bloch vector is a visualisation tool that maps the 2D, complex statevector onto real, 3D space.*

A quantum logic gate must have the same number of inputs as outputs. Each of Qgate can be represented by a 2^n square matrix and the effect of the gate can be computed my matrix multiplication.There are $(2^n)^2$ possible matrixes bu not all are valid as they must preserve the validity of the states they are transforming. To do this the matrix must be reversible, *unitary*.

Qbit states are represented as complex vectors $< x|$ a row vector, a *bra* and $|x >$ is a column vector, a *ket*.

The state of a single Qbit is represented by a two place vector (amplitude of being in state 0 and 1)

[2] Although state is both amplitude and phase in Qiskit this, at least initially, is simplified to $\pm Amplitude$.

$$[0> \stackrel{\text{def}}{=} \begin{bmatrix}1\\0\end{bmatrix} \quad [1> \stackrel{\text{def}}{=} \begin{bmatrix}0\\1\end{bmatrix}$$

But many such vectors are not valid Qbit states:

The state of n Qbits is represented by a 2^n place vector! To find the probability of measuring a state $[\phi>$ in the state $[x>$ we do: $p([x>) = |<x|\psi>|^2$ (the square of the scalar product)

Hence $|<\psi|\psi>|^2 = 1$ if $[\psi> = \alpha[0> + \beta[1>$ then $\alpha^2 + \beta^2 = 1$

Measuring the state of Qbits collapse the probabilities fixing the Qbits to always be in the initially observed state.

Whatever state our quantum system is in, there is always a measurement that has a deterministic outcome.

$\frac{\sqrt{2}}{2}|000\rangle + \frac{\sqrt{2}}{2}|111\rangle$