Differential Equations with Julia/SciML

Steven R. Dunbar

<2024-12-09 Mon>

What is Julia?

- https://julialang.org/ and https://docs.julialang.org/en/v1/
- Julia is a flexible dynamic language for scientific and numerical computing, with performance comparable to traditional statically-typed languages.
- Julia combines features of imperative, functional, and object-oriented programming.

Brief History

- Project from MIT Math and Computer Science
- Development started 2009
- Announced 14 February 2012
- Version 1.0 announced 8 August 2018
- Currently Version 1.11.2, 1 December 2024

Advantages of Julia

- Free and open source (MIT license)
- Many convenient features (Unicode, auto-vectorizing, natural function definition)
- Functions use multiple dispatch to select methods
- Both compiled and interactive (Notebooks (jupyter) and REPL)
- Julia has a built-in package manager
- Huge curated package collection (`https://juliahub.com/`, ~10,000 packages)
- access to CPUs, GPUs for multi-threading, parallel and distributed computing
- "Writes like Python, runs like C"

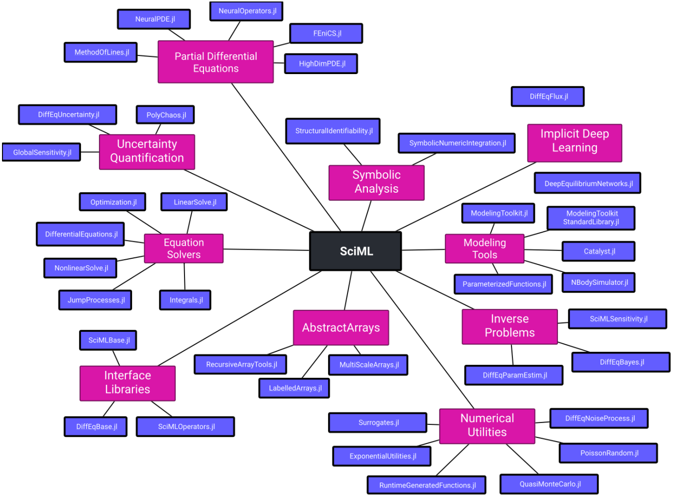
SciML Ecosystem

SciML is the combination of scientific computing techniques with machine learning.

Common-Interface High-Level software for

- Linear and nonlinear systems
- Integrals/Quadrature/Cubature
- Differential Equations, ODEs, SDEs, DAEs, DelayDEs
- Inverse Problems, and automated model discovery.
- Optimization
- Can use parallel, distributed, and GPU computing

SciML Family Tree



Lorenz Equations

Classic 3-dimensional "chaotic" ODE system:

$$
\begin{aligned}
x' &= \sigma(y - z) \\
y' &= x(\rho - z) - y \\
z' &= xy - \beta x
\end{aligned}
$$

$\sigma = 10, \rho = 28, \beta = 8/3$

$x(0) = 1, y(0) = 0, z(0) = 0$

$t \in [0, 30]$

Lorenz equations

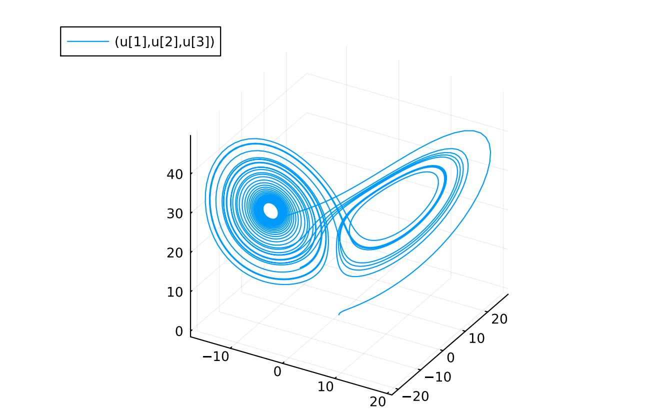
code

```
using DifferentialEquations
function parameterized_lorenz!(du, u, p, t)
    du[1] =  p[1] * (u[2] - u[1])
    du[2] =  u[1] * (p[2] - u[3]) - u[2]
    du[3] =  u[1] * u[2] - p[3] * u[3]
end
u0 = [1.0, 0.0, 0.0]
tspan = (0.0, 30.0)
p = [10.0, 28.0, 8 / 3]
prob = ODEProblem(parameterized_lorenz!, u0,
                  tspan, p)
sol = solve(prob)
```

Lorenz Equations Phase Portrait

code

```
using Plots
plot(sol, idxs = (1, 2, 3))
```

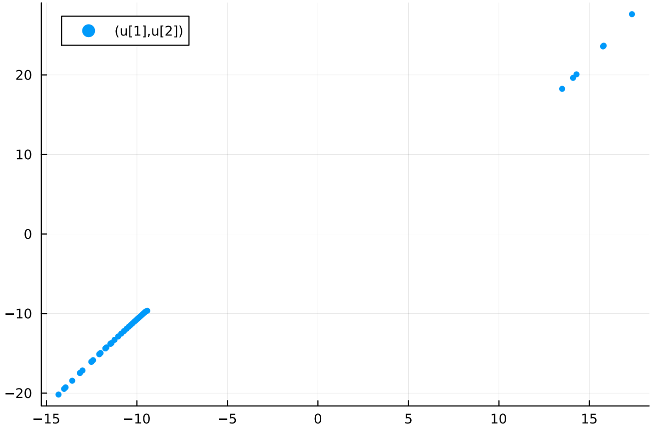


Event Handling

code

```
condition(u, t, integrator) = u[3] - 28.0
affect!(integrator) = nothing
cb = ContinuousCallback(condition, affect!,
        nothing, save_positions = (true, false))
_prob = ODEProblem(prob.f, u0, prob.tspan, p)
_sol = solve(_prob, Vern9(),
        save_everystep = false,
        save_start = false, save_end = false,
        callback = cb,
        abstol = 1e-16, reltol = 1e-16)
scatter(_sol, idxs = (1, 2),
        markersize = 3, msw = 0)
```

Poincare Map



Geometric Brownian Motion

Simplest stochastic differential equation:

$$dN_t = rN_t \,\mathrm{d}t + \alpha N_t \,\mathrm{d}B_t,$$
$$N(0) = N_0$$

$r = 1$, $\alpha = 1$, $N_0 = 1/2$.

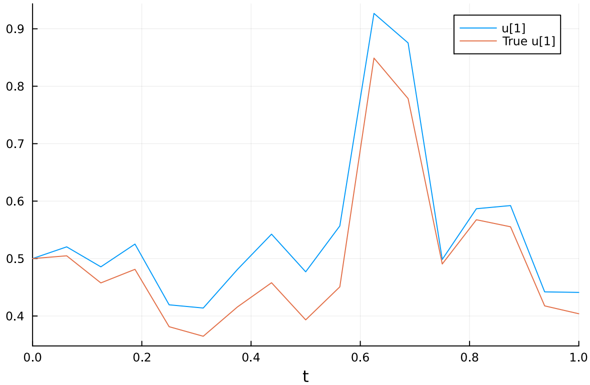
Solve with Euler-Maruyama, $h = \mathrm{d}t = 1/16$

Geometric Brownian Motion

code

```
using DifferentialEquations
r = 1, alpha = 1, N0 = 1 / 2
f(N, p, t) = r * N, g(N, p, t) = alpha * N
dt = 1 // 2^(4), tspan = (0.0, 1.0)
prob = SDEProblem(f, g, N0, tspan)
sol = solve(prob, EM(), dt = dt)
using Plots
plot(sol)
N_analytic(N0, p, t, W) =
    N0 * exp((r-(alpha^2)/2)*t+alpha*W)
ff = SDEFunction(f, g, analytic = N_analytic)
prob = SDEProblem(ff, N0, (0.0, 1.0))
sol = solve(prob, EM(), dt = dt)
plot(sol, plot_analytic = true)
```

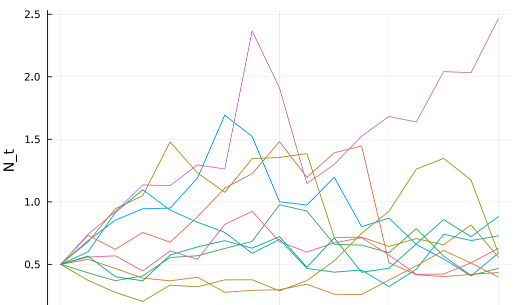
Geometric Brownian Motion



Geometric Brownian Motion Ensembles

code

```
ensembleprob = EnsembleProblem(prob)
sol10 = solve(ensembleprob, EM(), dt = 1//16,
              trajectories = 10, adaptive=false)
paths10 = [sol10.u[i].u for i in 1:10]
ts10 = sol10.u[1].t
```

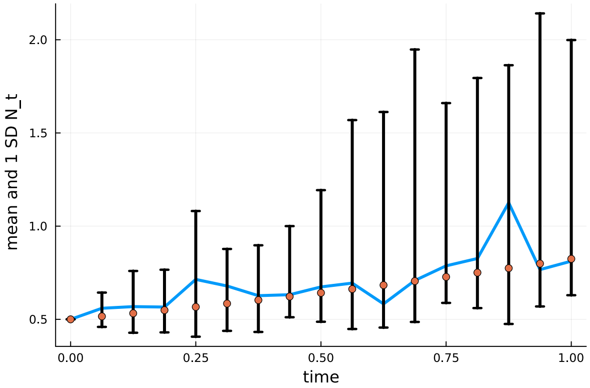


Geometric Brownian Motion Means, Variances

code

```
using DifferentialEquations.EnsembleAnalysis
summ10 = EnsembleSummary(sol10,
             quantiles=[0.16, 0.84])
plot(summ10, error_style=:bars)
theor_means = 0.5*exp.((r-alpha^2/2) .* ts10)
scatter!(ts10, theor_means,
         xlabel="time",
         ylabel=" mean and 1 SD N_t")
savefig("means_vars_example2_1.pdf")
```

Geometric Brownian Motion Means, Standard Deviations



Financial equations

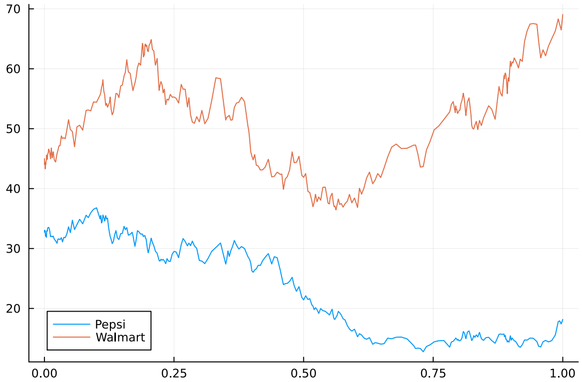
Financial Model 2.2, (4b)

Pepsi and Walmart Stock prices, 2006

$$
\begin{aligned}
\mathrm{d}X_p &= -0.0545 X_p \, \mathrm{d}t + 0.5 X_p \, \mathrm{d}B_p(t) + 0.1 X_w \, \mathrm{d}B_w(t) \\
\mathrm{d}X_w &= +0.0125 X_w \, \mathrm{d}t + 0.1 X_p \, \mathrm{d}B_p(t) + 0.5 X_w \, \mathrm{d}B_w(t)
\end{aligned}
$$

RKMilCommute - An explicit Runge-Kutta discretization

Strong Order 1.0 Milstein method, adaptive step size.



Viscous Burgers Equation

$$
\begin{aligned}
u_t + u u_x &= \nu u_{xx} \\
u(0, x) &= u_0(x)
\end{aligned}
$$

An Exact Solution:

$$
u(t, x) = \frac{2}{1 + e^{(x-t)/\nu}}
$$

Parameters in Viscous Burgers Equation

code

```
using OrdinaryDiffEq, ModelingToolkit,
    MethodOfLines, DomainSets
# Exact solution
u_exact =
 x, t) -> 2.0 ./(1.0 .+ exp.( (x.-t)/nu))
xleft = -7.0; xright = 7.0; nu = 0.5;
# Parameters, variables, and derivatives
@parameters t x
@variables u(..)
Dt = Differential(t)
Dx = Differential(x)
Dxx = Differential(x)^2
```

Equation and BCs

code

```
# 1D PDE and boundary conditions
eq = Dt(u(t, x)) + u(t,x) * Dx(u(t,x)) ~
    nu * Dxx(u(t, x))
bcs = [u(0, x) ~ exp.(-(x .- 1.0).^2 ./ 2.0 ) .-
    exp.(-(x .+ 1.0).^2 ./ 2.0 ),
       u(t, xleft) ~ 2.0, u(t, xright) ~ 0.0]
# Space and time domains
domains = [t in Interval(0.0, 10.00),
    x in Interval(xleft, xright)]
# PDE system
@named pdesys = PDESystem(eq, bcs, domains,
                          [t, x], [u(t, x)])
```

Solving the Equation

code

```
# Method of lines discretization
dx = 0.1
order = 2
discretization = MOLFiniteDifference([x => dx],
# Convert the PDE problem into an ODE problem
prob = discretize(pdesys, discretization)
# Solve ODE problem
using OrdinaryDiffEq
sol = solve(prob, Tsit5(), saveat = 1.0)
#  Compare with exact solution
discrete_x = sol[x]
discrete_t = sol[t]
solu = sol[u(t, x)]
```

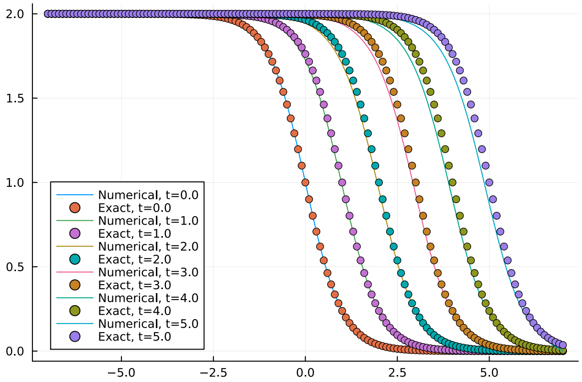
Plotting the Solution

code

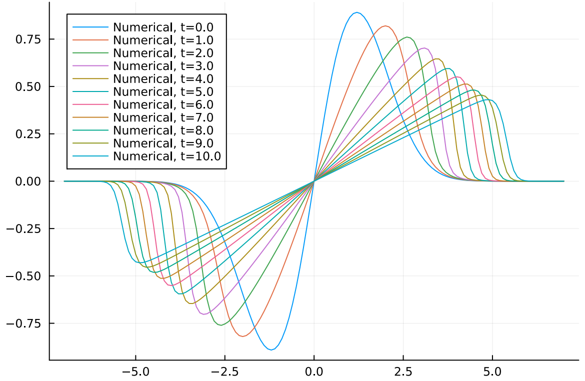
```
using Plots
plt = plot()

for i in eachindex(discrete_t)
    plot!(discrete_x, solu[i, :],
          label = "Numerical, t=$(discrete_t[i])
    scatter!(
        discrete_x,
        u_exact(discrete_x, discrete_t[i]),
        label = "Exact, t=$(discrete_t[i])")
end
plt
```

Solution of Burgers Equation

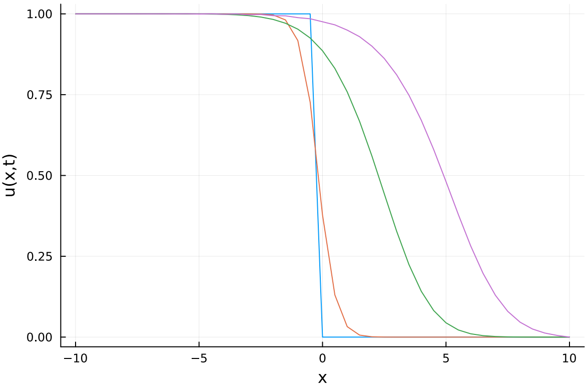


Solution of N-wave initial condition



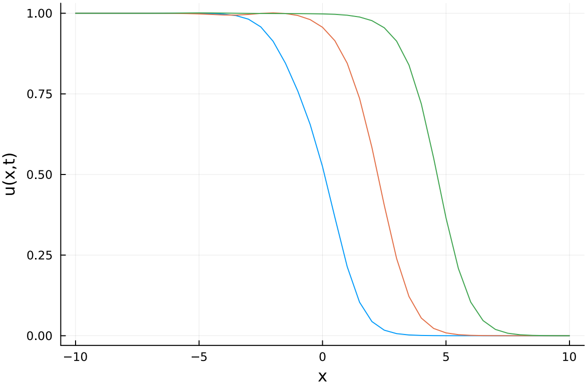
Fisher (KPP) Equation

$$u_t = \frac{D}{2} u_{xx} + ru(1-u)$$



Telegrapher Fisher Equation

$$u_{tt}+((2a+2b)-4bu)u_t = s^2 u_{xx}+(-b^2-2ab)u+(b^2+2ab)u^2$$



Other PDE Solvers

- NeuralPDE.jl is a package of neural network solvers for PDEs using physics-informed neural networks (PINNs).
- FEniCS.jl is a wrapper for the FEniCS finite element method library.
- ApproxFun.jl is a package for approximating functions in basis sets. (Spectral methods)
- Ferrite.jl is a library for finite element software.
- Gridap.jl is a package for grid-based approximation of PDEs
- Trixi.jl is a package for numerical simulation of hyperbolic conservation laws.
- VoronoiFVM.jl is a library for generating FVM discretizations.

SciML Family Tree