Automatic differentiation (AD).

- What is AD? An almost too-good-to-be-true way to compute derivatives of any computationally implemented function.
  - Similar in concept to the complex step method, but relies on “dual numbers” which are a symbolic generalization of imaginary numbers.
  - AD tends to be more convenient and actually cheaper than complex step.
- Works for *any function*, not just analytical expressions!

```julia
using ForwardDiff

f(x) = sin(pi * x)
df = ForwardDiff.derivative(f, .2)
```

AD_examples.jl

```julia
function herons_method(x0)
    @assert x0 > 0
    x = x0
    for _ in 1:25
        x = 0.5 * (x + x0 / x)
    end
    return x
end

x = 0.1
@show ForwardDiff.derivative(herons_method, x)
```

Automatic differentiation, cont.

- Can create functions which represent function derivatives using ForwardDiff.jl
- These behave like any other function in Julia; you can broadcast, pass them as arguments, analyze type stability, etc.

```
julia> @code_warntype deriv_heron(.1)
MethodInstance for deriv_heron(::Float64)
  from deriv_heron(x) @ Main Untitled-1:12
Arguments
  #self#::Core.Const(deriv_heron)
  x::Float64
Body::Float64
1 ─ %1 = ForwardDiff.derivative::Core.Const(ForwardDiff.derivative)
│   %2 = Main.heron::Core.Const(heron)
│   %3 = (%1)(%2, x)::Float64
└──      return %3
```

```
using ForwardDiff
using Plots

function heron(x0)
    @assert x0 > 0
    x = x0
    for _ in 1:2
        x = 0.5 * (x + x0 / x)
    end
    return x
end

deriv_heron(x) =
    ForwardDiff.derivative(heron, x)

x = LinRange(.1, 1, 100)
plot!(x, deriv_heron.(x))
```

Higher, partial, and multi-dimensional derivatives

- AD computes arbitrarily high derivatives accurately and stably.
- Partial derivatives can be computed by using “closures”
  - Closure: an anonymous function which encloses an external variable into the function’s scope.
- Note: closures can be slow in Julia, but they can usually be made efficient.
- If you have a vector-valued input or output, you can compute the gradient or Jacobian of the function.

```
function newton_step(f, x0)
    J = ForwardDiff.jacobian(f, x0)
    δ = J \ f(x0)

    return x0 - δ
end

function newton(f, x0)
    x = x0

    for i in 1:10
        x = newton_step(f, x)
        @show x
    end

    return x
end
```

What about efficiency?

- Surprisingly, AD usually takes about the same amount of time that it takes to evaluate a function.
- Can be used within hot loops without significant overhead (at least when using JAX in Python or Julia).
- Caveat: for higher dimensional functions (e.g., many inputs or many outputs) AD becomes more expensive.
  - The cost scales with the dimension; this can be reduced using specific AD algorithms (e.g., “reverse mode” AD).

```
using BenchmarkTools
@btime heron($.1)
@btime deriv_heron($.1)
```

```
 3.000 ns (0 allocations: 0 bytes)
 4.208 ns (0 allocations: 0 bytes)
```

```
f(x) = sum(x)
@btime f($(randn(10)))
@btime ForwardDiff.gradient($f, $(randn(10)))
```

```
4.958 ns (0 allocations: 0 bytes)
346.259 ns (3 allocations: 1.92 KiB)
```

Tabular data: DataFrames.jl

- Similar to the “pandas” Python library for handling tables and time-series.
- Helpful for manipulating data from CSV (comma-separated value) files (e.g., spreadsheets).
  - CSV.jl is a separate library for reading CSV files into Julia data types such as a DataFrame.
- Can slice, join, sub-select, compute statistics, etc.

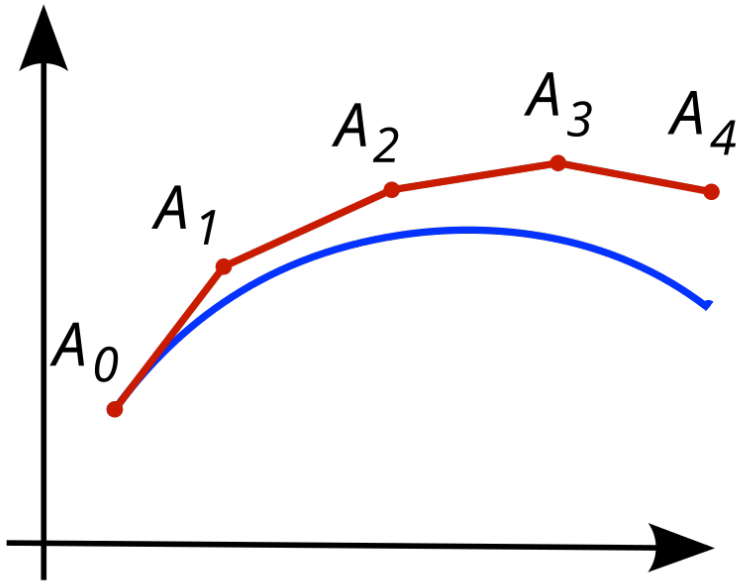
```
using CSV
using DataFrames

path = joinpath(pkgdir(DataFrames), "docs",
                "src", "assets", "german.csv");
german = CSV.read(path, DataFrame)
```

```
julia> german
1000×10 DataFrame
 Row  │ id     Age    Sex      Job    Housing
      │ Int64  Int64  String7  Int64  String7
──────┼──────────────────────────────────────
    1 │     0     67  male         2  own
    2 │     1     22  female       2  own
    3 │     2     49  male         1  own
    4 │     3     45  male         2  free
    5 │     4     53  male         2  free
    6 │     5     35  male         1  free
    7 │     6     53  male         2  own
    8 │     7     35  male         3  rent
  ⋮   │   ⋮      ⋮      ⋮        ⋮      ⋮
  993 │   992     23  male         1  rent
  994 │   993     30  male         3  own
  995 │   994     50  male         2  own
  996 │   995     31  female       1  own
  997 │   996     40  male         3  own
  998 │   997     38  male         2  own
  999 │   998     23  male         2  free
 1000 │   999     27  male         2  own
```

Ordinary differential equations (ODEs)

- "Rate of change" phenomena are typically modeled as systems of ODEs.
  - Examples: infectious disease modeling, predatory-prey population models, mechanical oscillators, neuron models, simplified climate models, etc…
  - Time-dependent partial differential equations (PDEs) are turned into very large systems of ODEs.
- Most ordinary differential equations don't have an explicit solution, so we use time-stepping (Explicit Euler, Runge-Kutta, etc)

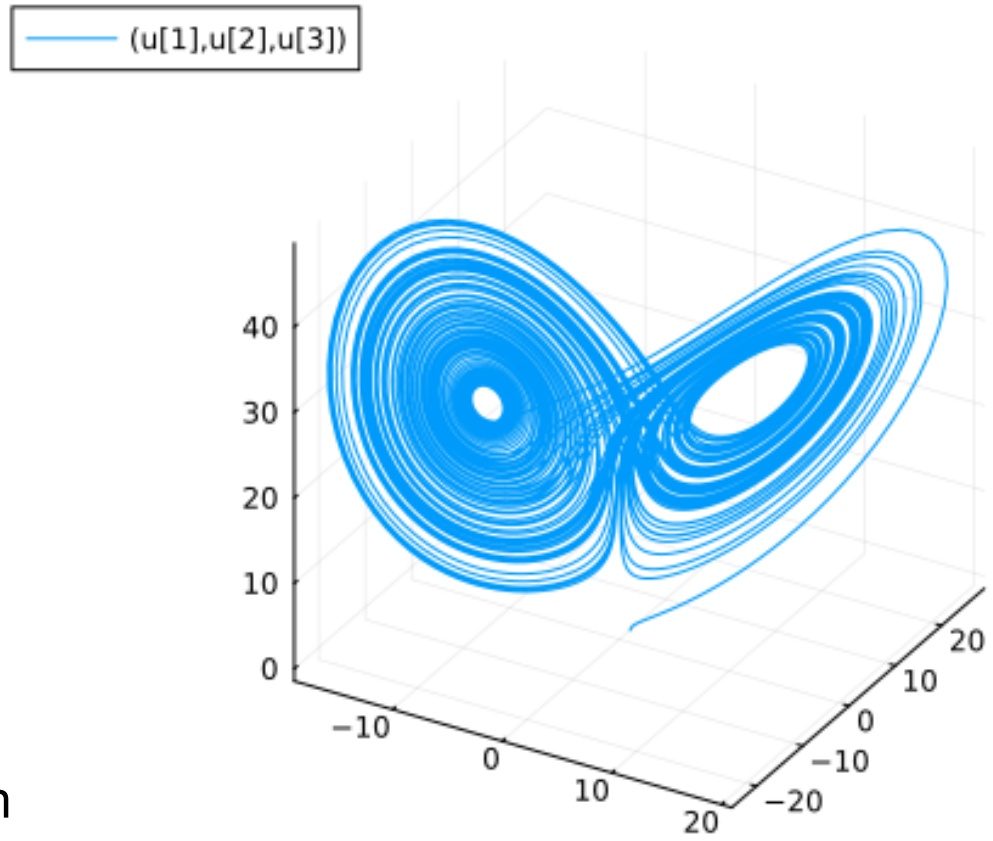


$$\frac{dx}{dt} = f(x,t)$$

$$x_{k+1} = x_k + \Delta t f(x_k, t_k)$$

Why use an ODE package?

- Take advantage of efficient but cumbersome implementations.
- Time-adaptivity is great for a lot of problems, but the algorithms tend to be more technical.
- Convenience: lots of functions are inconvenient to implement:
  - Saving intermediate solutions at arbitrary times.
  - Activating behavior at specific times which may not line up with a uniform time-stepping grid.



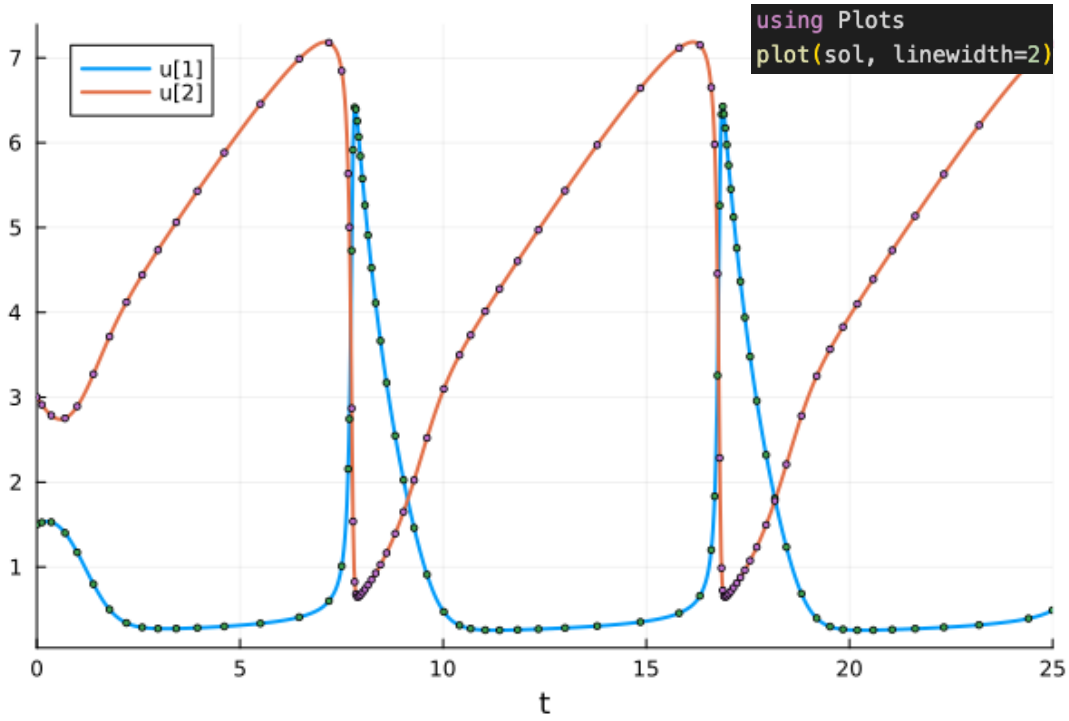
OrdinaryDiffEq.jl

- Part of the SciML and “DifferentialEquations.jl” ecosystem, which includes neural ODEs, stochastic ODEs, uncertainty quantification, etc…

$$\frac{du}{dt} = f(x, t)$$

```julia
using OrdinaryDiffEq # or DifferentialEquations

function brusselator(du, u, p, t)
    B = 4
    y1, y2 = u
    du[1] = 1 + y1^2 * y2 - (B+1) * y1
    du[2] = B * y1 - y1^2 * y2
end
u0 = [1.5, 3.0]
tspan = (0.0, 25.0)
prob = ODEProblem(brusselator, u0, tspan)
sol = solve(prob, Tsit5())
```



Adaptive time-stepping

- Handles adaptive time-stepping automatically.
  - Adaptivity is easy conceptually but tends to complicate implementation significantly.
  - Good parameter choices for adaptive time-stepping vary depending on the method as well.
- When applicable, takes advantage of *embedded* time-stepping methods which make it cheaper to compute adaptive step sizes.

Algorithm 2 High-level overview of a time integration loop using explicit RK methods with embedded error estimator and PID controllers in OrdinaryDiffEq.jl and deviations in PETSc/TS

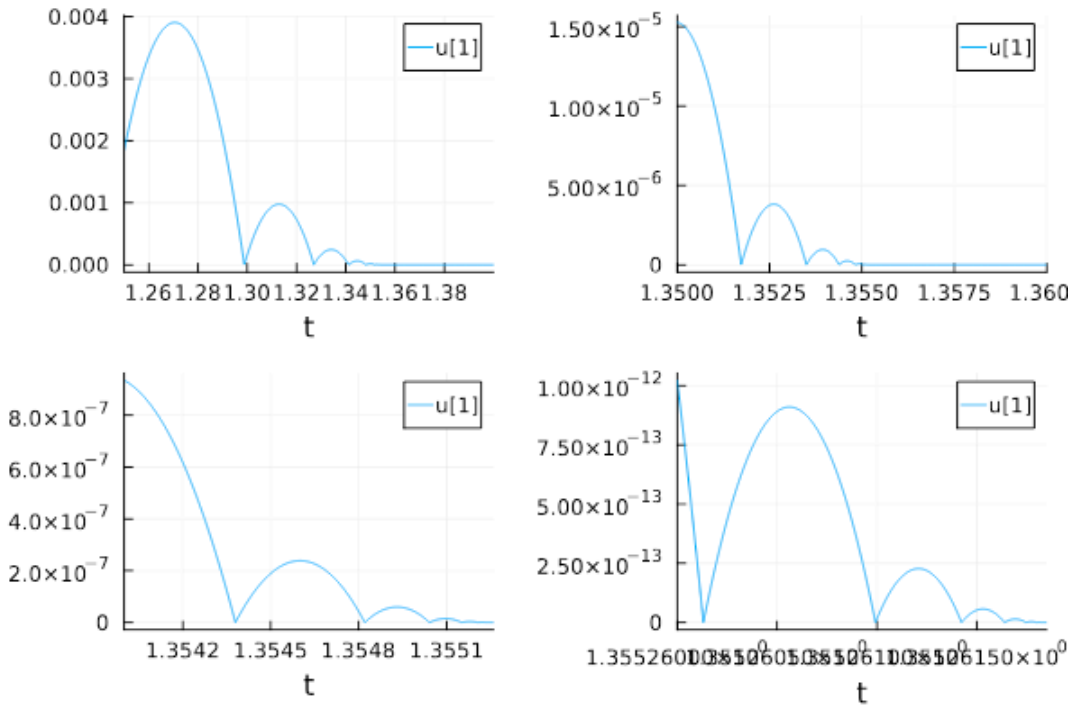
i: Initialize time $t \leftarrow 0$, time step number $n \leftarrow 0$, and initial state u^0.
ii: Initialize PID controller with $\varepsilon_0 \leftarrow 1$, $\varepsilon_{-1} \leftarrow 1$.
iii: Initialize $\Delta t_0 = \widetilde{\Delta t}$ with a given value or the algorithm of [38, p. 169].
iv: Initialize `accept_step` $\leftarrow$ `false`
v: **while** $t < T$ **do**
vi: **if** `accept_step` **then** ▷ callbacks can change `accept_step`
vii: `accept_step` $\leftarrow$ `false` ▷ prepare for the next step
viii: $t \leftarrow \widetilde{t}$
ix: $\Delta t_{n+1} \leftarrow \widetilde{\Delta t}$
x: $n \leftarrow n + 1$
xi: **else**
xii: $\Delta t_n \leftarrow \widetilde{\Delta t}$
xiii: **end if**
xiv: **if** $t + \Delta t_n > T$ **then**
xv: $\Delta t_n \leftarrow T - t$
xvi: **end if**
xvii: Compute u^{n+1} and $\widehat{u}^{n+1}$ with time step Δt_n
xviii: $w_{n+1} \leftarrow \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(\frac{u_i^{n+1} - \widehat{u}_i^{n+1}}{\tau_a + \tau_r \max\left\{\left|u_i^{n+1}\right|, \left|\widetilde{u}_i\right|\right\}}\right)^2}$
xix: ▷ $\widetilde{u} = u^n$ in OrdinaryDiffEq.jl, $\widetilde{u} = \widehat{u}^{n+1}$ in PETSc
xx: $w_{n+1} \leftarrow \max\{w_{n+1}, w_{\min}\}$
xxi: ▷ $w_{\min} \approx 2.2 \times 10^{-16}$ in OrdinaryDiffEq.jl, $= 1.0 \times 10^{-10}$ in PETSc
xxii: $\varepsilon_{n+1} \leftarrow 1/w_{n+1}$
xxiii: `dt_factor` $\leftarrow \kappa\left(\varepsilon_{n+1}^{\beta_1/k} \varepsilon_n^{\beta_2/k} \varepsilon_{n-1}^{\beta_3/k}\right)$ ▷ default $\kappa(a) = 1 + \arctan(a - 1)$
xxiv: $\widetilde{\Delta t} \leftarrow$ `dt_factor` $\cdot \Delta t_n$
xxv: **if** `dt_factor` $\geqslant$ `accept_safety` **then** ▷ default `accept_safety` $= 0.81$
xxvi: `accept_step` $\leftarrow$ `true`
xxvii: **else**
xxviii: `accept_step` $\leftarrow$ `false`
xxix: **end if**
xxx: **if** `accept_step` **then**
xxxi: $\widetilde{t} \leftarrow t + \Delta t_n$
xxxii: **if** $\widetilde{t} \approx T$ **then** ▷ within 100 units in last place in OrdinaryDiffEq.jl
xxxiii: $\widetilde{t} \leftarrow T$
xxxiv: **end if**
xxxv: Apply callbacks ▷ AMR, CFL-based control in OrdinaryDiffEq.jl
xxxvi: **end if**
xxxvii: **end while**

Solution interpolation and callbacks

- Handles solution interpolation, which is non-trivial to implement since the accuracy of the interpolation depends on the time-stepper!
  - Can use “saveat” or just “sol(t)” to evaluate the solution at arbitrary times.
- Includes at least 300 unique named time-stepping methods, over 1000 solvers (explicit, implicit, multi-step, stabilized, stochastic, etc)
- Can insert event data or extract observable data using *callbacks*. Algorithms for handling event data are quite nuanced!
  - Events are triggered if a condition (which is a function of the solution) is met. OrdinaryDiffEq.jl solves a root finding problem (using bisection, Newton’s method, etc) to determine when *exactly* the condition is met so that there is a time-step precisely on this condition.

Event handling example: Zeno's paradox

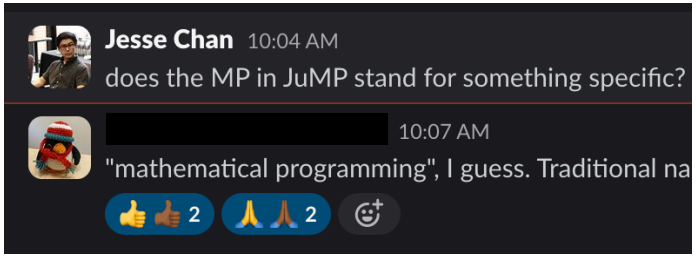
- The ODE is the equation for acceleration under gravity $x''(t) = -9.8x(t)$.
- If the height of a ball goes below zero, the ball bounces.
- A bounce is assumed to set the velocity to -1/2 * (original velocity).



https://docs.sciml.ai/DiffEqDocs/stable/features/callback_functions/

Optimization and modeling

- JuMP.jl is an "algebraic modeling language for mathematical optimization” (good to know if you’re an operations research (OR) major!)
- Like Plots.jl, JuMP interfaces with several different solver *backends*, but focuses on providing an *algebraic modeling language*.
  - Modeling language: simplifies complex problem formulations, provide a unified solver interface, enables pre-solve simplification and reformulation.
- Focused on differentiable optimization with constraints; can handle continuous, integer and binary variable types.



Note: I am very much not an expert in this area.

Some examples

- Linear programs in standard form; can also use individual variable syntax instead of linear algebra syntax if you prefer.
- Knapsack example: easy way to specify which variables are constrained to be {0, 1}

```
@variable(model, x[1:n], Bin)
```

- More complex examples, e.g., quadratic constraints, bilevel/nested optimization

$$
\begin{aligned}
\min_{x,z} \quad & x_1^2 + x_2^2 + z \\
s.t. \quad & z = \max_{y} \quad x_1^2 y_1 + x_2^2 y_2 - x_1 y_1^4 - 2 x_2 y_2^4 \\
& \qquad s.t. \quad (y_1 - 10)^2 + (y_2 - 10)^2 \leq 25 \\
& x \geq 0.
\end{aligned}
$$

```
using JuMP
using HiGHS, Ipopt

# max c' * x
# s.t. A * x = b, x ≥ 0
model = Model(HiGHS.Optimizer)
# model = Model(Ipopt.Optimizer)
A = [1 1 9 5; 3 5 0 8; 2 0 6 13]
b = [7, 3, 5]
c = [1, 3, 5, 2]

@variable(model, x[1:4] >= 0)
@constraint(model, A * x .== b)
@objective(model, Min, c' * x)
print(model)
```

```
Min x[1] + 3 x[2] + 5 x[3] + 2 x[4]
Subject to
 x[1] + x[2] + 9 x[3] + 5 x[4] = 7
 3 x[1] + 5 x[2] + 8 x[4] = 3
 2 x[1] + 6 x[3] + 13 x[4] = 5
 x[1] ≥ 0
 x[2] ≥ 0
 x[3] ≥ 0
 x[4] ≥ 0
```

JuMP_examples.jl

How do you create a Julia package?

- It turns out that it is remarkably easy to create your own Julia package!
- Use “generate” in the REPL’s Pkg mode to initialize the minimum structure necessary for a package.
  - Note: PkgTemplates.jl can be used to perform a much more in-depth initialization (e.g., setting up CodeCov, CI, documentation, and more).
- The Project.toml generated for a package has more information than for a standard local Project.toml file.

```
(@v1.8) pkg> generate HelloWorld
```

```
shell> tree HelloWorld/
HelloWorld/
├── Project.toml
└── src
    └── HelloWorld.jl
```

```
name = "HelloWorld"
uuid = "b4cd1eb8-1e24-11e8-3319-93036a3eb9f3"
version = "0.1.0"
authors = ["Some One <someone@email.com>"]

[deps]
```

The content of `src/HelloWorld.jl` is:

```
module HelloWorld

greet() = print("Hello World!")

end # module
```

Modules in Julia

- Modules are used to organize types, code, and create packages in Julia.
- They create a new namespace, and you can specify which variables are exported if you load a module.
- You can create submodules within a module, though these are not often necessary in Julia.

```
module ExampleModule

export ExampleStruct
export example_function

struct ExampleStruct{T}
    x::T
end

function example_function(x::ExampleStruct)
    return x
end

end
```

```
julia> include("module_example.jl")
Main.ExampleModule

julia> using .ExampleModule

julia> example_function(ExampleStruct(1))
Int64
```

Example: creating a Julia package

- Lets create a Julia package “CanonicalBasisVectors.jl”.
- Main steps:
  - Create a module to hold and export the structure CanonicalBasisVector, as well as as the function “inner”.
  - Create a Github repository to host the package.
- We’ll test to see if we can then then load the package from Github.
- Demo…