Chemical Engineering 541

Numerical Methods

Introduction

Family

Website

ChE 541 Numerical Methods for Engineers

- Syllabus
- Schedule
- Lecture Notes
- Homework
- Online Resources
- Reference Texts
- Programming Tips
- Personal Information Form

TAs
Notes:

- See BYU Learning Suite for all grades.

Office Hours:

- See syllabus for links to virtual office hour meetings

	M	T	W	Th	F
8:00 am					
9:00 am					
10:00 am					
11:00 am					
12:00 pm					
1:00 pm					
2:00 pm					
3:00 pm		Lignell	Lignell	Lignell	
4:00 pm					
5:00 pm					

All class material will be on the website:

- http://ignite.byu.edu/che541

All scores and homework will be recorded on Learning Suite

- Learningsuite.byu.edu

Zoom link

- https://byu.zoom.us/my/dlignell

Trends

- PDEs → ODEs → algebraic systems
- Nonlinear systems → Linear systems
- One complex equation becomes many “simpler” ones.
- Continuous problems become discrete
  - Solutions at specified locations, or times.
    - Domain split into a grid of points or cells.
- Most realistic problems require numerical solution.
  - This is the rule, not the exception
    - (Yet this topic is still relegated to an elective course ☹)

Example Applications

- Fluid Flow / Heat Transfer
  - Unsteady PDEs → discretize to large system of ODEs
  - Implicit: solve nonlinear systems at each "timestep"
    - Reduce these to iterative linear systems. Iterate to "convergence." Repeat.
- Reaction Engineering
  - Solve (maybe large) system of nonlinear equations (ODE's).
  - PFR, PSR
  - Add spatial depences/diffusion → PDEs
  - Mechanism size reduction:
    - Solve nonlinear and linear systems of equations. Eigenvalue analysis to reduce dimensions.
- Chemical Equilibrium
  - Solve systems of nonlinear equations to minimize Gibbs free energy.
- Pipeline design
  - Solve systems of nonlinear equations.
- Distillation → system of nonlinear equations for tray compositions
- h = h(T): given T, find h (easy); given h, find T (harder)
  - Often, h is nicer to work with than T (h is conserved in adiabatic systems, but T is not!)

Direct Solution of Turbulent Combustion

Unknowns:

ρ,
v,
e_o,
Y_k,
P

Auxiliary:

Flux relations:
Heat, Mass,
Momentum.
Energy/temperature
Mixing relations

Equations:

continuity:
$$\frac{\partial \rho}{\partial t} = -\frac{\partial(\rho v_i)}{\partial x_i},$$

momentum:
$$\frac{\partial(\rho v_i)}{\partial t} = -\frac{\partial(\rho v_j v_i)}{\partial x_i} - \frac{\partial P}{\partial x_i} + \frac{\partial \tau_{i,j}}{\partial x_i},$$

energy:
$$\frac{\partial(\rho e_o)}{\partial t} = -\frac{\partial(\rho e_o v_i)}{\partial x_i} - \frac{\partial(P v_i)}{\partial x_i} + \frac{\partial(\tau_{i,j} \cdot v_j)}{\partial x_i} - \frac{\partial q_i}{\partial x_i},$$

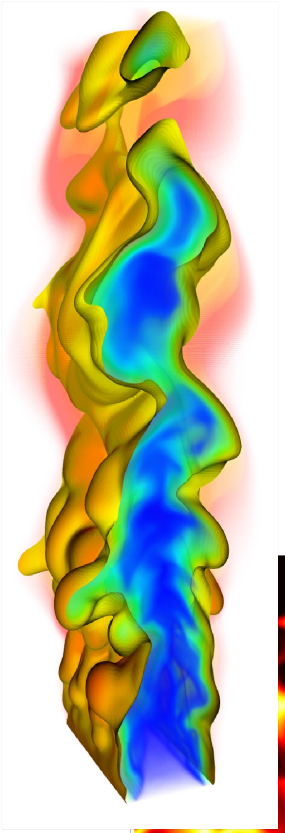
species:
$$\frac{\partial(\rho Y_k)}{\partial t} = -\frac{\partial(\rho Y_k v_i)}{\partial x_i} - \frac{\partial(j_{k,i})}{\partial x_i} + \omega_k.$$

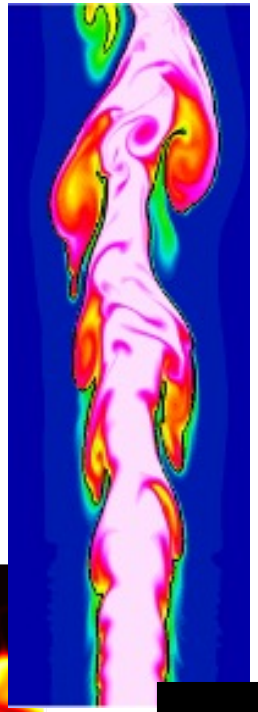
EOS:
$$P = \frac{\rho R T}{W},$$

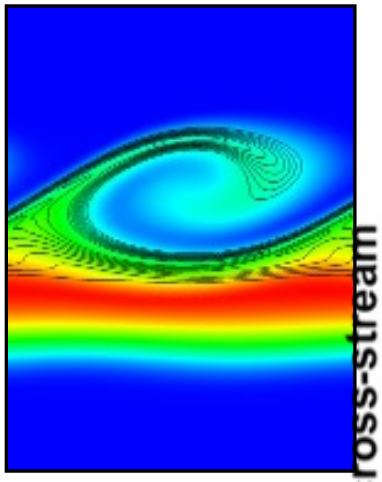
Numerical Solution

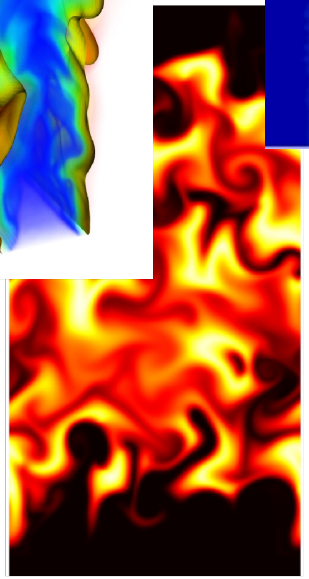
- Method of lines.
- 8th order finite difference discretization
- 4th order explicit Runge Kutta integration
- Nonlinear solution of enthalpy, temperture relationship.
- Optional implicit reaction integration with explicit diffusion, convection
- Processing of data involves many other numerical techniques (e.g., interpolation, integration).

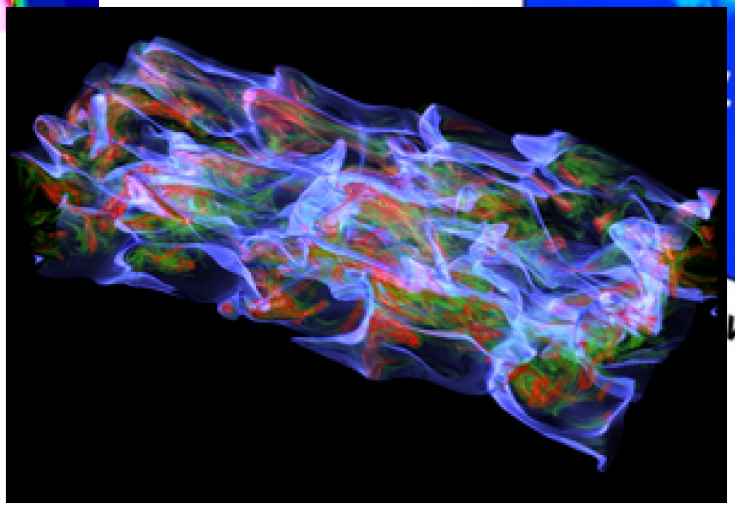
Selected Results

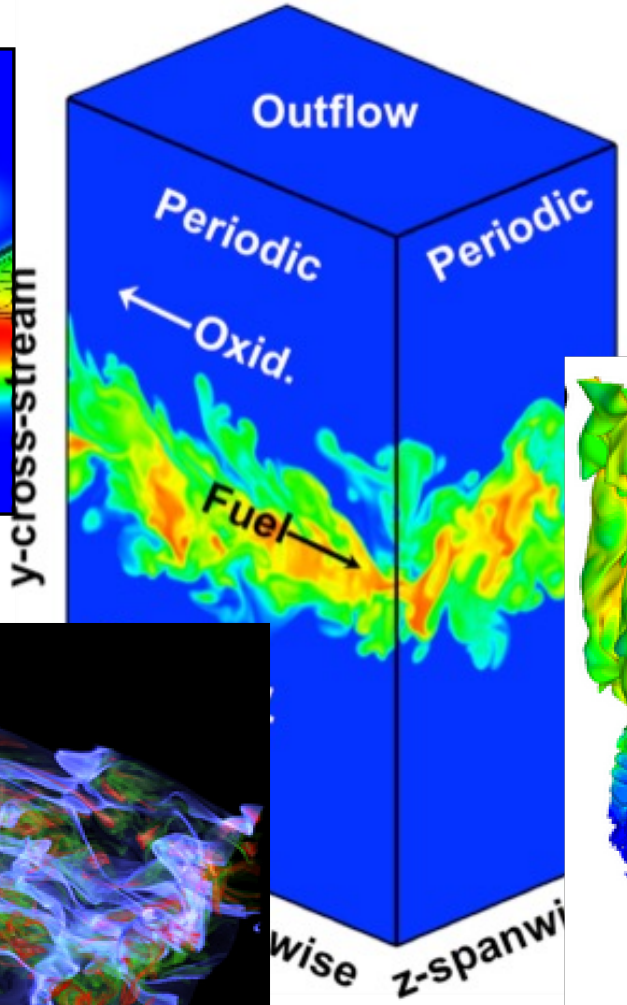


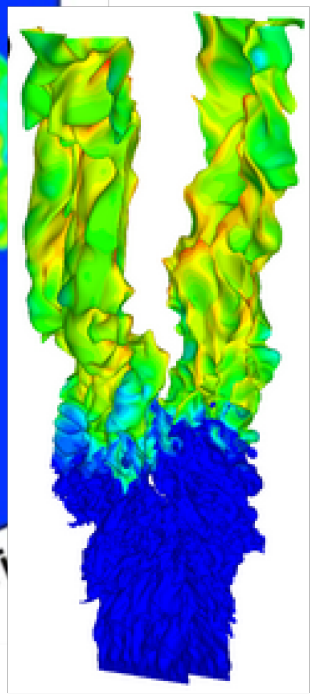




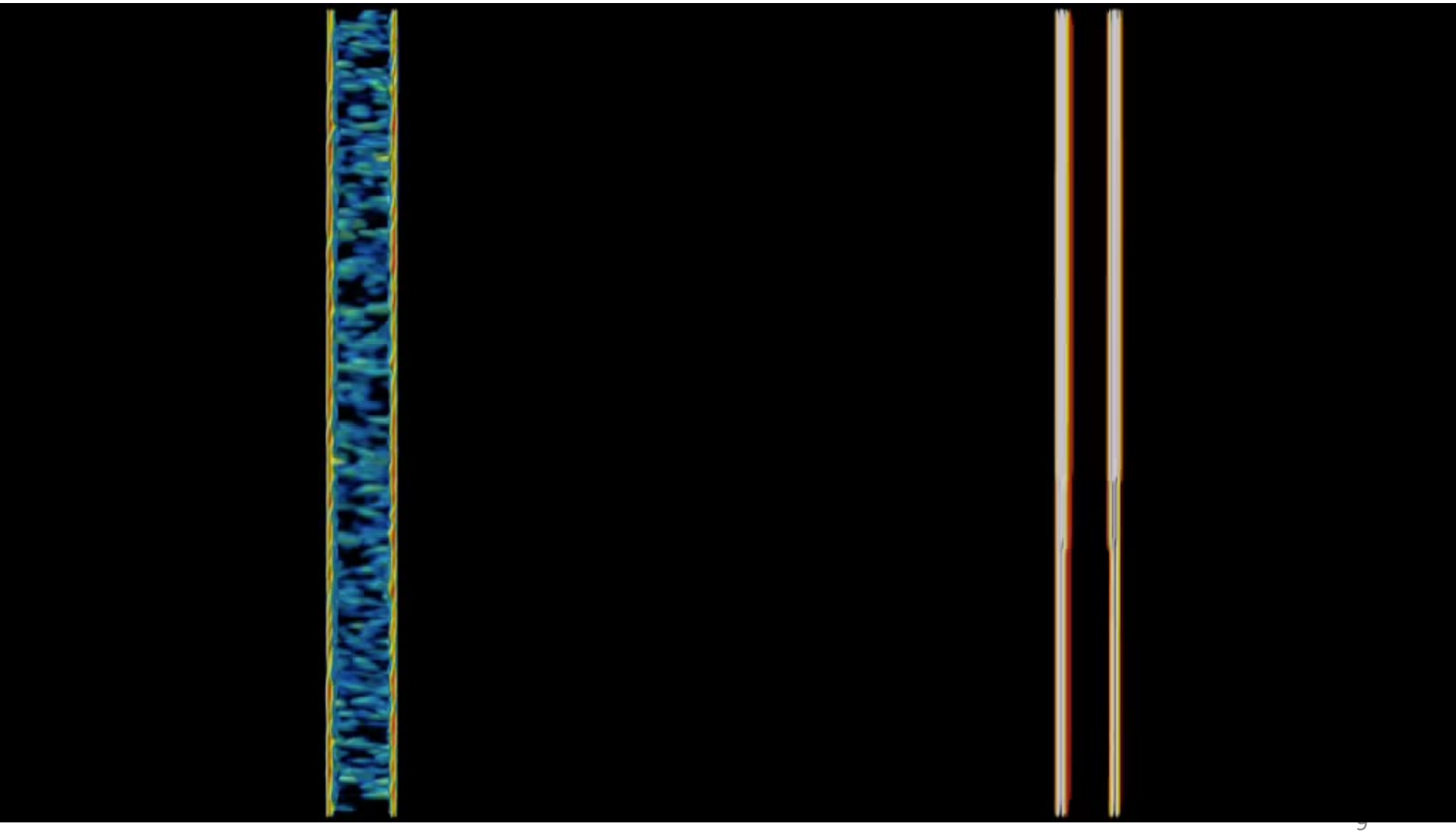








Simulation Results



Language Summary

Language	Platform	Student Cost	Professional Cost	Note
VBA	Windows	Free	N/A (available)	Limited numerical functionality: Excel
Mathcad	Windows	$25	$1,550	
Matlab	All	$99	$2150+	Free versions available. Toolboxes needed
Python	All	Free	Free	
Maple				
Mathematica				
etc. see http://en.wikipedia.org/wiki/Comparison_of_numerical_analysis_software				

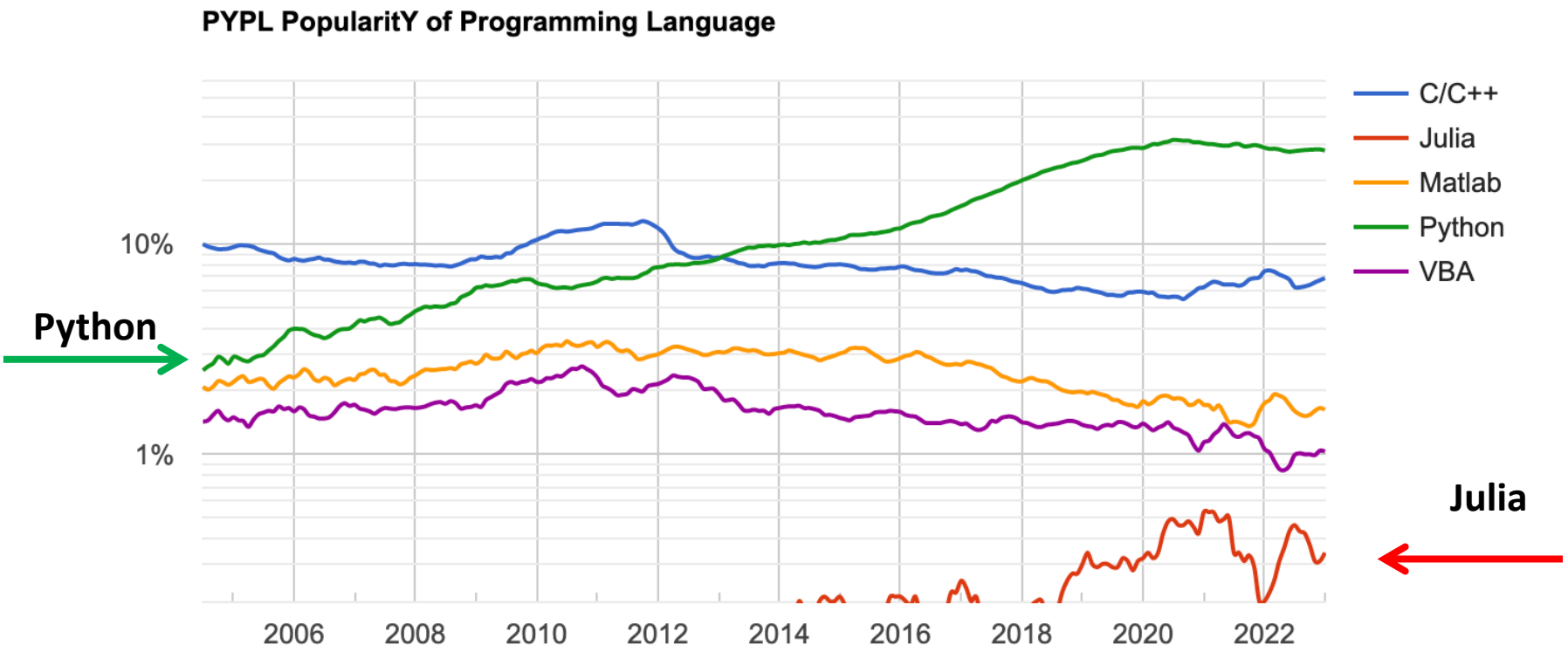
- Python, Julia, MATLAB for programing
  - MATLAB is most advanced for numerics; but slow
  - Python is most extensible, and broad: a “real” programing language; but slow
  - Julia is built for numerics; fast; new
- Full featured
  - Plotting, symbolic, built-in numerical tools: ODEs, functions, interpolation, plotting, etc.

Language Trends

Worldwide, Jan 2023 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Python	27.93 %	-0.9 %
2		Java	16.78 %	-1.3 %
3		JavaScript	9.63 %	+0.5 %
4	↑	C#	6.99 %	-0.3 %
5	↓	C/C++	6.9 %	-0.5 %
6		PHP	5.29 %	-0.8 %
7		R	4.03 %	-0.2 %
8	↑↑↑	TypeScript	2.79 %	+1.0 %
9		Swift	2.23 %	+0.3 %
10	↓↓	Objective-C	2.2 %	-0.1 %
11	↑↑	Go	1.94 %	+0.7 %
12	↑↑↑	Rust	1.9 %	+0.9 %
13	↓	Kotlin	1.81 %	+0.1 %
14	↓↓↓↓	Matlab	1.63 %	-0.1 %
15	↑	Ruby	1.13 %	+0.3 %
16	↓↓	VBA	1.03 %	-0.0 %
17		Ada	0.89 %	+0.2 %
18	↑↑↑	Dart	0.86 %	+0.5 %
19		Scala	0.62 %	+0.0 %
20	↓↓	Visual Basic	0.56 %	-0.1 %
21	↑↑↑	Lua	0.55 %	+0.2 %
22	↓↓	Abap	0.5 %	+0.1 %
23	↑↑	Haskell	0.35 %	+0.1 %
24	↑↑↑↑	Julia	0.34 %	+0.1 %
25	↓↓↓	Groovy	0.34 %	-0.0 %
26		Cobol	0.33 %	+0.1 %
27	↓↓↓↓	Perl	0.33 %	+0.0 %
28	↓	Delphi/Pascal	0.12 %	-0.1 %

Language Trends



The PYPL PopularitY of Programming Language Index is created by analyzing how often language tutorials are searched on Google.

Python—Anaconda

https://www.anaconda.com/products/distribution#Downloads

Use Python 3.9 (64 bit)

anaconda.com/products/distribution#Downloads

Anaconda Installers

Windows

Python 3.9

64-Bit Graphical Installer (621 MB)

MacOS

Python 3.9

64-Bit Graphical Installer (688 MB)

64-Bit Command Line Installer (681 MB)

64-Bit (M1) Graphical Installer (484 MB)

64-Bit (M1) Command Line Installer (472 MB)

Linux

Python 3.9

64-Bit (x86) Installer (737 MB)

64-Bit (Power8 and Power9) Installer (360 MB)

64-Bit (AWS Graviton2 / ARM64) Installer (534 MB)

64-bit (Linux on IBM Z & LinuxONE) Installer (282 MB)

Jupyter Lab

File Edit View Run Kernel Tabs Settings Help

Files | Running | Commands | Cell Tools | Tabs

> notebooks

Name	Last Modified
Data.ipynb	an hour ago
Fasta.ipynb	a day ago
Julia.ipynb	a day ago
Lorenz.ipynb	seconds ago
R.ipynb	a day ago
iris.csv	a day ago
lightning.json	9 days ago
lorenz.py	3 minutes ago

Lorenz.ipynb | Terminal 1 | Console 1 | Data.ipynb | README.md

Code | Python 3

In this Notebook we explore the Lorenz system of differential equations:

$$\dot{x} = \sigma(y - x)$$
$$\dot{y} = \rho x - y - xz$$
$$\dot{z} = -\beta z + xy$$

Let's call the function once to view the solutions. For this set of parameters, we see the trajectories swirling around two points, called attractors.

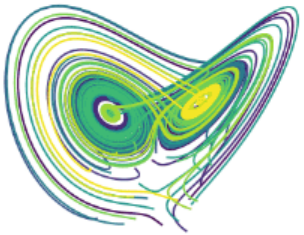
In [4]:
```
from lorenz import solve_lorenz
t, x_t = solve_lorenz(N=10)
```

Output View

sigma 10.00

beta 2.67

rho 28.00



lorenz.py

```
def solve_lorenz(N=10, max_time=4.0, sigma=10.0, beta=8./3, rho=28.0):
    """Plot a solution to the Lorenz differential equations."""
    fig = plt.figure()
    ax = fig.add_axes([0, 0, 1, 1], projection='3d')
    ax.axis('off')

    # prepare the axes limits
    ax.set_xlim((-25, 25))
    ax.set_ylim((-35, 35))
    ax.set_zlim((5, 55))

    def lorenz_deriv(x_y_z, t0, sigma=sigma, beta=beta, rho=rho):
        """Compute the time-derivative of a Lorenz system."""
        x, y, z = x_y_z
        return [sigma * (y - x), x * (rho - z) - y, x * y - beta * z]

    # Choose random starting points, uniformly distributed from -15 to 15
    np.random.seed(1)
    x0 = -15 + 30 * np.random.random((N, 3))

```

Julia

juliadocs.github.io/Julia-Cheat-Sheet/

English

The Fast Track to julia 1.0

A quick and dirty overview of

This page's source is located here. Pull requests are welcome!

What is...?

Julia is an open-source, multi-platform, high-level, high-performance programming language for technical computing.

Julia has an LLVM-based JIT compiler that allows it to match the performance of languages such as C and FORTRAN without the hassle of low-level code. Because the code is compiled on the fly you can run (bits of) code in a shell or REPL, which is part of the recommended workflow.

Julia is dynamically typed, provides multiple dispatch, and is designed for parallelism and distributed computation.

Julia has a built-in package manager.

Julia has many built-in mathematical functions, including special functions (e.g. Gamma), and supports complex numbers right out of the box.

Julia allows you to generate code automagically thanks to Lisp-inspired macros.

Julia was born in 2012.

Basics

Assignment	`answer = 42`
	`x, y, z = 1, [1:10; ], "A string"`
	`x, y = y, x # swap x and y`

Constant declaration	`const DATE_OF_BIRTH = 2012`
End-of-line comment	`i = 1 # This is a comment`
Delimited comment	`#= This is another comment =#`

Chaining	`x = y = z = 1  # right-to-left`
	`0 < x < 3  # true`
	`5 < x != y < 5 # false`

Function definition	`function add_one(i)`
	`  return i + 1`
	`end`

Insert LaTeX symbols	`\delta` + [Tab]

Operators

Basic arithmetic	`+,-,*,/`
Exponentiation	`2^3 == 8`
Division	`3/12 == 0.25`
Inverse division	`7\3 == 3/7`
Remainder	`x % y` or `rem(x,y)`
Negation	`!true == false`
Equality	`a == b`
Inequality	`a != b` or `a ≠ b`
Less and larger than	`<` and `>`
Less than or equal to	`<=` or `≤`
Greater than or equal to	`>=` or `≥`
Element-wise operation	`[1, 2, 3] .+ [1, 2, 3] == [2, 4, 6]` `[1, 2, 3] .* [1, 2, 3] == [1, 4, 9]`
Not a number	`isnan(NaN)` not(!) `NaN == NaN`
Ternary operator	`a == b ? "Equal" : "Not equal"`
Short-circuited AND and OR	`a && b` and `a || b`
Object equivalence	`a === b`

Collection functions

Apply f to all elements of collection coll	`map(f, coll)` or
	`map(coll) do elem`
	`  # do stuff with elem`
	`  # must contain return`
	`end`

Filter coll for true values of f	`filter(f, coll)`
List comprehension	`arr = [f(elem) for elem in coll]`

Types

Julia has no classes and thus no class-specific methods.

Types are like classes without methods.

Abstract types can be subtyped but not instantiated.

Concrete types cannot be subtyped.

By default, `struct` s are immutable.

Immutable types enhance performance and are thread safe, as they can be shared among threads without the need for synchronization.

Objects that may be one of a set of types are called `Union` types.

Type annotation	`var::TypeName`

Type declaration	`struct Programmer`
	`  name::String`
	`  birth_year::UInt16`
	`  fave_language::AbstractString`
	`end`

Mutable type declaration	replace struct with mutable struct
Type alias	`const Nerd = Programmer`
Type constructors	`methods(TypeName)`
Type instantiation	`me = Programmer("Ian", 1984, "Julia")` `me = Nerd("Ian", 1984, "Julia")`

Subtype declaration	`abstract type Bird end`
	`struct Duck <: Bird`
	`  pond::String`
	`end`

Parametric type	`struct Point{T <: Real}`
	`  x::T`
	`  y::T`
	`end`
	
	`p =Point{Float64}(1,2)`

Union types	`Union{Int, String}`
Traverse type hierarchy	`supertype(TypeName)` and `subtypes(TypeName)`
Default supertype	`Any`
All fields	`fieldnames(TypeName)`
All field types	`TypeName.types`

When a type is defined with an *inner* constructor, the default *outer* constructors are not available and have to be defined manually if need be. An inner constructor is best used to check whether the parameters conform to certain (invariance) conditions. Obviously, these invariants can be violated by accessing and modifying the fields directly, unless the type is defined as immutable. The `new` keyword may be used to create an object of the same type.

Type parameters are invariant, which means that

Learn Julia in Y Minutes

learnxinyminutes.com/docs/julia/

Learn X in Y minutes

Share this page

Select theme: light dark

Where X=Julia

Get the code: learnjulia.jl

Julia is a new homoiconic functional language focused on technical computing. While having the full power of homoiconic macros, first-class functions, and low-level control, Julia is as easy to learn and use as Python.

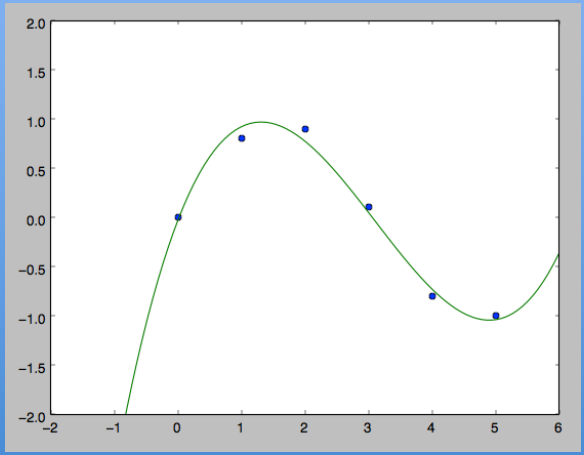
This is based on Julia 1.0.0

```
# Single line comments start with a hash (pound) symbol.
#= Multiline comments can be written
   by putting '#=' before the text  and '=#'
   after the text. They can also be nested.
=#

#################################################
## 1. Primitive Datatypes and Operators
#################################################

# Everything in Julia is an expression.

# There are several basic types of numbers.
typeof(3)       # => Int64
typeof(3.2)     # => Float64
typeof(2 + 1im) # => Complex{Int64}
typeof(2 // 3)  # => Rational{Int64}

# All of the normal infix operators are available.
1 + 1      # => 2
8 - 1      # => 7
10 * 2     # => 20
35 / 5     # => 7.0
10 / 2     # => 5.0  # dividing integers always results in a Float64
div(5, 2)  # => 2    # for a truncated result, use div
5 \ 35     # => 7.0
2^2        # => 4    # power, not bitwise xor
12 % 10    # => 2

# Enforce precedence with parentheses
(1 + 3) * 2  # => 8

# Bitwise Operators
~2         # => -3 # bitwise not
3 & 5      # => 1  # bitwise and
```

Code: fit polynomial to data

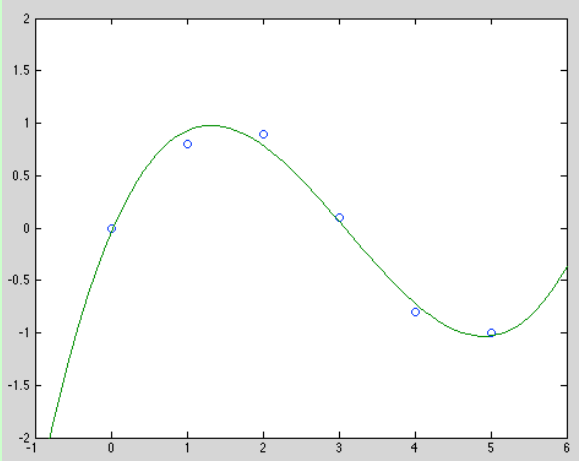
Python

```python
from numpy              import *
from scipy.interpolate  import *
from matplotlib.pyplot  import *

x  = array([0, 1, 2, 3, 4, 5])
y  = array([0, 0.8, 0.9, 0.1, -0.8, -1])
xp = linspace(-2,6,100)

p3 = polyfit(x,y,3)

plot(x,y,'o', xp,polyval(p3,xp),'-')
ylim(-2,2)
ion(); show()
```



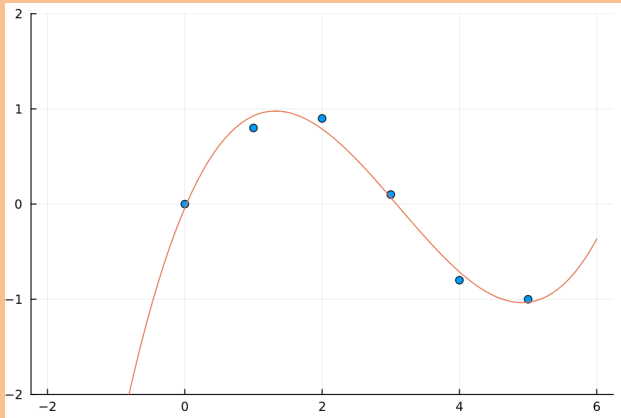
Matlab

```modern_latex_math_table_of_contents_with_page_numbers
x  = [0, 1, 2, 3, 4, 5];
y  = [0, 0.8, 0.9, 0.1, -0.8, -1];
xp = linspace(-2,6,100);

p3  = polyfit(x,y,3);

plot(x,y,'o', xp,polyval(p3,xp),'-');
ylim([-2,2]);
```



Julia

```julia
using Plots
using Polynomials
using CurveFit

x = [0, 1, 2, 3, 4, 5]
y = [0, 0.8, 0.9, 0.1, -0.8, -1]
xp = collect(range(-2,stop=6,length=100))

p3  = poly_fit(x,y,3)
fp3 = Polynomial(p3)

scatter(x,y)
plot!(xp,fp3.(xp), legend=false, ylimits=(-2,2))
```



Code: integrate function

Python

```python
from numpy              import *
from scipy.integrate    import *
from matplotlib.pyplot  import *

def F(x) :
    return 2*x**2 + 1

I = quad(F,0,1)

print I
```

Matlab

```
I = quad(@integrationF, 0,1)
```

```
function F = integrationF(x)

    F = 2*x.^2 + 1;

end
```

2 files (optional)

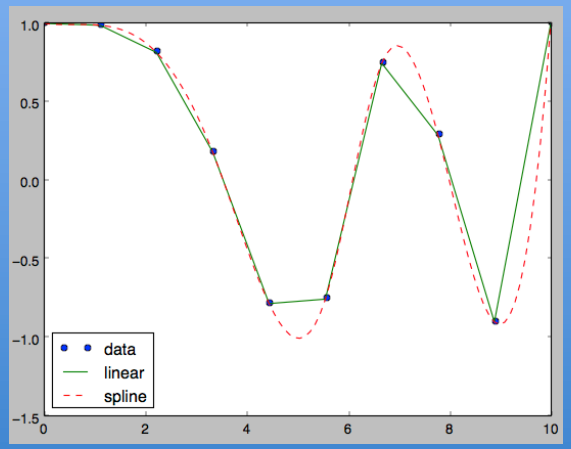
Julia

```julia
using QuadGK

function f(x)
    return 2.0*x^2 + 1
end

I,err = quadgk(f,0,1)
println(I)
```

Code: interpolation

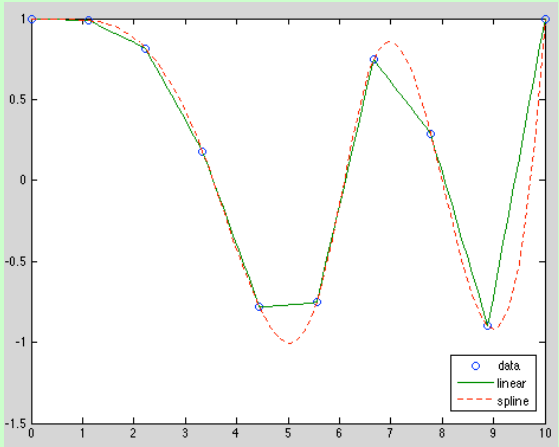
Python

```python
from numpy             import *
from scipy.interpolate import *
from matplotlib.pyplot import *

x1        = linspace(0,10,10)
y1        = cos(x1**2/8)
x2        = linspace(0,10,100)

f_linear  = interp1d(x1,y1)
f_spline  = interp1d(x1,y1, kind='cubic')
y2_linear = f_linear(x2)
y2_spline = f_spline(x2)

plot(x1,y1,'o', x2,y2_linear,'-', x2,y2_spline,'--')
legend(['data', 'linear', 'spline'], loc='best')
ion(); show()
```



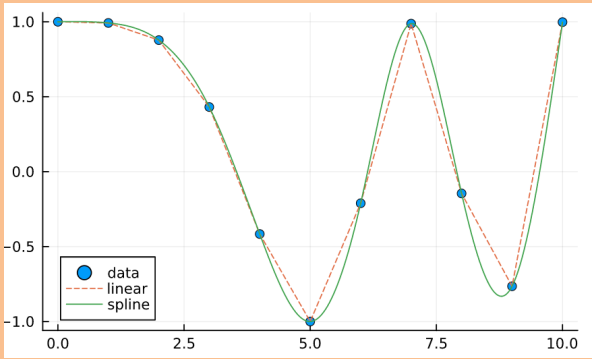
Matlab

```MATLAB
x1        = linspace(0,10,10);
y1        = cos(-x1.^2/8);
x2        = linspace(0,10,100);

y2_linear = interp1(x1,y1,x2,'linear','extrap');
y2_spline = interp1(x1,y1,x2,'spline','extrap');

plot(x1,y1,'o', x2,y2_linear,'-', x2,y2_spline,'--')
legend('data', 'linear', 'spline', 'Location','Best')
```



Julia

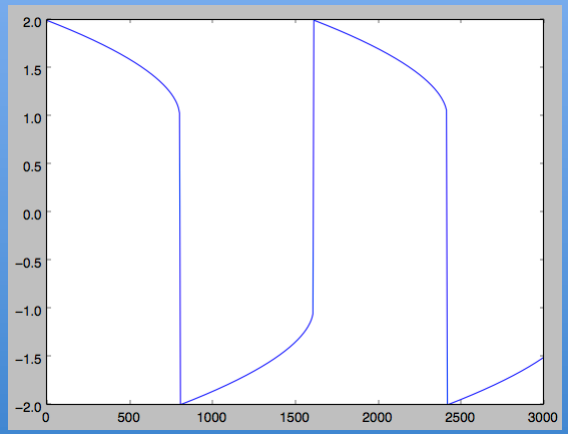
```julia
using Plots
using Interpolations

x1 = 0:10
y1 = cos.(x1.^2.0./8.0)
x2 = 0:0.01:10

f_linear = LinearInterpolation(x1,y1)
f_spline = CubicSplineInterpolation(x1, y1)

p=scatter(x1, y1,        label="data", size=(500,300))
p=plot!(x2,f_linear(x2), label="linear", ls=:dash)
p=plot!(x2,f_spline(x2), label="spline", ls=:solid)
```



Code: Stiff ODE system

Python

```python
from numpy              import *
from scipy.integrate    import *
from matplotlib.pyplot  import *

def odeF(y,t) :
    dydt = zeros(2)
    dydt[0] = y[1]
    dydt[1] = 1000*(1-y[0]**2)*y[1]-y[0]
    return dydt

t = linspace(0,3000,500)
y0 = array([2, 0])
y = odeint(odeF, y0, t, mxstep=1000)

plot(t,y[:,0],'-')
ion(); show()
```

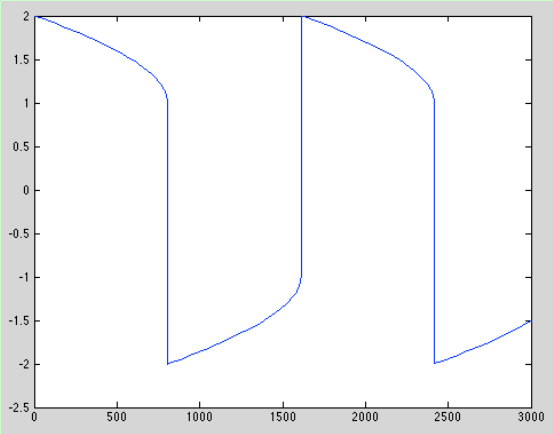


Matlab

```modelica
function ydot = odeF(t,y)

    ydot(1,1) = y(2);
    ydot(2,1) = 1000*(1-y(1)^2)*y(2)-y(1);

end
```

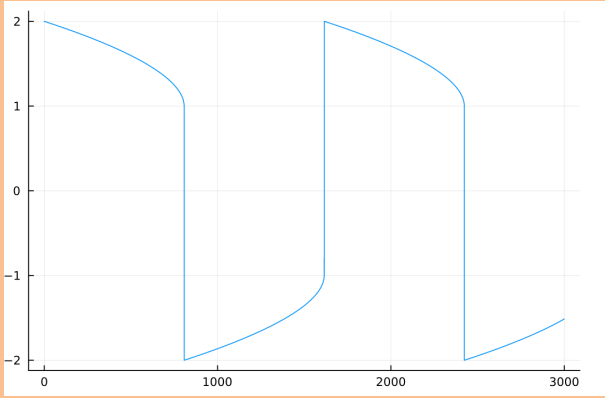
```modelica
tend = 3000;
y0   = [2 0];
[t y] = ode15s(@odeF, [0 tend], y0);

plot(t,y(:,1),'-');
```

2 files



Julia

```julia
using DifferentialEquations
using Plots

function rhsf!(dy, y, p, t)
    dy[1] = y[2]
    dy[2] = 1000*(1-y[1]^2)*y[2]-y[1]
end

tspan = (0.0, 3000)
y0 = [2 0]

prob = ODEProblem(rhsf!, y0, tspan)
sol = solve(prob)

y = vcat(sol.u...)
plot(sol.t, y[:,1], legend=false)
```



Code: system of nonlinear equations

Python

```python
from scipy.optimize import *
from numpy          import *

#-------------------------------------

def solverF(x) :

    f1 = x[0];
    f2 = x[1];
    f3 = x[2];
    Q1 = x[3];
    Q2 = x[4];
    Q3 = x[5];

    Qt = 0.01333;        # Given total volumetric flow rate
    e1 = 0.00024;        # pipe roughness (m)
    e2 = 0.00012;
    e3 = 0.0002;
    L1 = 100;            # pipe length (m)
    L2 = 150;
    L3 = 80;
    D1 = 0.05;           # pipe diameter (m)
    D2 = 0.045;
    D3 = 0.04;
    mu = 1.002E-3;       # viscosity (kg/m*s)
    rho = 998;           # density (kg/m3)

    F = zeros((6));

    F[0] = f1*L1/D1*rho/2*(4*Q1/pi/D1**2)**2 - \
           f2*L2/D2*rho/2*(4*Q2/pi/D2**2)**2;     # DP_1 - DP_2 = 0
    F[1] = f1*L1/D1*rho/2*(4*Q1/pi/D1**2)**2 - \
           f3*L3/D3*rho/2*(4*Q3/pi/D3**2)**2;     # DP_1 - DP_3 = 0
    F[2] = 1/sqrt(f1)+2*log10(e1/D1/3.7 + \
           2.51/(rho*D1/mu*4*Q1/pi/D1**2/sqrt(f1)));  # Colbrook 1
    F[3] = 1/sqrt(f2)+2*log10(e2/D2/3.7 + \
           2.51/(rho*D2/mu*4*Q2/pi/D2**2/sqrt(f2)));  # Colbrook 2
    F[4] = 1/sqrt(f3)+2*log10(e3/D3/3.7 + \
           2.51/(rho*D3/mu*4*Q3/pi/D3**2/sqrt(f3)));  # Colbrook 3
    F[5] = Q1+Q2+Q3-Qt;                               # total flow

    return F

#-------------------------------------

Qt     = 0.01333;        # Given total volumetric flow rate
xGuess = array([0.01, 0.01, 0.01, 0.004, 0.004, Qt-0.004-0.004]);
                # f1, f2, f3, Q1, Q2, Q3

x      = fsolve(solverF, xGuess)

print "[f1, f2, f3, Q1, Q2, Q3] = ", x
```

Matlab

```MATLAB
function F=solverF(x)

    f1 = x(1);          % recover variables so eqautions easier to read
    f2 = x(2);
    f3 = x(3);
    Q1 = x(4);
    Q2 = x(5);
    Q3 = x(6);

    Qt = 0.01333;       % Given total volumetric flow rate
    e1 = 0.00024;       % pipe roughness (m)
    e2 = 0.00012;
    e3 = 0.0002;
    L1 = 100;           % pipe length (m)
    L2 = 150;
    L3 = 80;
    D1 = 0.05;          % pipe diameter (m)
    D2 = 0.045;
    D3 = 0.04;
    mu = 1.002E-3;      % viscosity (kg/m*s)
    rho = 998;          % density (kg/m3)

    F(1) = f1*L1/D1*rho/2*(4*Q1/pi/D1^2)^2 - ...
           f2*L2/D2*rho/2*(4*Q2/pi/D2^2)^2;     % DP_1 - DP_2 = 0
    F(2) = f1*L1/D1*rho/2*(4*Q1/pi/D1^2)^2 - ...
           f3*L3/D3*rho/2*(4*Q3/pi/D3^2)^2;     % DP_1 - DP_3 = 0
    F(3) = 1/sqrt(f1)+2*log10(e1/D1/3.7 + ...
           2.51/(rho*D1/mu*4*Q1/pi/D1^2/sqrt(f1))); % Colbrook 1
    F(4) = 1/sqrt(f2)+2*log10(e2/D2/3.7 + ...
           2.51/(rho*D2/mu*4*Q2/pi/D2^2/sqrt(f2))); % Colbrook 2
    F(5) = 1/sqrt(f3)+2*log10(e3/D3/3.7 + ...
           2.51/(rho*D3/mu*4*Q3/pi/D3^2/sqrt(f3))); % Colbrook 3
    F(6) = Q1+Q2+Q3-Qt;                              % total flow

end
```

2 files

```MATLAB
Qt = 0.01333;       % Given total volumetric flow rate

xGuess = ([0.01 0.01 0.01 0.004 0.004 Qt-0.004-0.004]);
          % f1, f2, f3, Q1, Q2, Q3

x = fsolve(@solverF, xGuess)
```

```julia
using NLsolve

function solverFunc!(F, x)
    f1, f2, f3 = x[1:3]
    Q1, Q2, Q3 = x[4:6]

    e1, e2, e3 = 0.00024, 0.00012, 0.0002   # pipe roughness (m)
    L1, L2, L3 = 100, 150, 80               # pipe length (m)
    D1, D2, D3 = 0.05, 0.045, 0.04          # pipe diameter (m)
    μ = 1.002E-3                            # viscosity (kg/m*s)
    ρ = 998                                 # density (kg/m3)

    F[1] = f1*L1/D1*ρ/2*(4*Q1/π/D1^2)^2 -    # ΔP1 - ΔP2 = 0
           f2*L2/D2*ρ/2*(4*Q2/π/D2^2)^2
    F[2] = f1*L1/D1*ρ/2*(4*Q1/π/D1^2)^2 -    # ΔP1 - ΔP3 = 0
           f3*L3/D3*ρ/2*(4*Q3/π/D3^2)^2
    F[3] = 1/√f1 + 2*log10(e1/D1/3.7 +
           2.51/(ρ*D1/μ*4*Q1/π/D1^2/√f1)) # Colbrook 1
    F[4] = 1/√f2 + 2*log10(e2/D2/3.7 +
           2.51/(ρ*D2/μ*4*Q2/π/D2^2/√f2)) # Colbrook 1
    F[5] = 1/√f3 + 2*log10(e3/D3/3.7 +
           2.51/(ρ*D3/μ*4*Q3/π/D3^2/√f3)) # Colbrook 1
    F[6] = Q1+Q2+Q3-Qt
end

Qt = 0.01333                    # total volumetric flow rate
xGuess = [0.01, 0.01, 0.01, 0.004, 0.004, Qt-0.004-0.004]

x = nlsolve(solverFunc!, xGuess).zero
println("f1, f2, f3, Q1, Q2, Q3 = \n", x)
```

Flow through 3 parallel pipes given total flow, pipe props

Code: 2D unsteady heat equation

Python

```python
# 2-D unsteady heat equation
# df/dt = alpha*d2f/dx2 + d2f/dy2 + S
# Forward Euler, central difference.
# Finite difference
# Points on boundaries, solve interior points.
# BC = 0; IC = 0

from numpy                 import *
from matplotlib.pyplot     import *
from mpl_toolkits.mplot3d  import *
from math                  import *

Ld       = 1.0                  # domain length
nTauRun  = 0.5                  # # of diffusion timescales to run
nxy      = 22                   # # of uniform grid points in x, y
alpha    = 1                    # thermal diffusivity
cfl      = 0.5                  # time step factor

tau  = Ld**2/alpha              # domain diffusion timescale
tend = nTauRun*tau              # run time
dxy  = Ld/(nxy-1)               # grid spacing
dt   = dxy**2/alpha/4*cfl       # time step size
nt   = ceil(tend/dt)            # number of time steps
dt   = tend/nt                  # clean it up
np   = ceil(1/cfl)*10           # how often to plot?

f = zeros((nt,nxy,nxy))         # initialize the solution
S = ones((nt,nxy,nxy))          # set the source term

X,Y = meshgrid(linspace(0,Ld,nxy),linspace(0,Ld,nxy))  # for plotting
i = arange(1,nxy-1)
j = i

for it in arange(1,nt) :

    f[it][ix_(i,j)] = f[it-1][ix_(i,j)] \
                    + (alpha*dt/dxy**2)*(f[it-1][ix_(i-1,j)]-2*f[it-1][ix_(i,j)]+f[it-1][ix_(i+1,j)]) \
                    + (alpha*dt/dxy**2)*(f[it-1][ix_(i,j-1)]-2*f[it-1][ix_(i,j)]+f[it-1][ix_(i,j+1)]) \
                    + S[it-1][ix_(i,j)]

    if(it%np==0) :          # plot
        clf()
        contourf(X,Y,f[it,:,:],20)
        ion(); show()
        aa = raw_input()
```

Matlab

```matlab
% 2-D unsteady heat equation
% df/dt = alpha*d2f/dx2 + d2f/dy2 + S
% Forward Euler, central difference.
% Finite difference
% Points on boundaries, solve interior points.
% BC = 0; IC = 0

Ld       = 1.0;                 % domain length
nTauRun  = 0.5;                 % # of diffusion timescales to run
nxy      = 22;                  % # of uniform grid points in x, y
alpha    = 1;                   % thermal diffusivity
cfl      = 0.5;                 % time step factor

tau  = Ld^2/alpha;              % domain diffusion timescale
tend = nTauRun*tau;             % run time
dxy  = Ld/(nxy-1);              % grid spacing
dt   = dxy^2/alpha/4*cfl;       % time step size
nt   = ceil(tend/dt);           % number of time steps
dt   = tend/nt;                 % clean it up
np   = ceil(1/cfl)*10;          % how often to plot?

f = zeros(nt,nxy,nxy);          % initialize the solution
S = ones(nt,nxy,nxy);           % set the source term

[X,Y] = meshgrid(linspace(0,Ld,nxy),linspace(0,Ld,nxy)); % for plotting
i = 2:nxy-1;
j = i;

for it=2:nt

    f(it,i,j) = f(it-1,i,j) ...
                + (alpha*dt/dxy^2)*(f(it-1,i-1,j)-2*f(it-1,i,j)+f(it-1,i+1,j)) ...
                + (alpha*dt/dxy^2)*(f(it-1,i,j-1)-2*f(it-1,i,j)+f(it-1,i,j+1)) ...
                + S(it-1,i,j);

    if(mod(it,np)==0)           % plot
        Z = reshape(f(it,:,:),nxy,nxy);
        contourf(X,Y,Z,20);
        pause(0.1);
    end

end
```

Finite difference, Euler integration

Julia

```julia
using Plots

Ld      = 1.0
nTauRun = 0.5
nxy     = 22
alpha   = 1
cfl     = 0.5

tau  = Ld^2/alpha
tend = nTauRun*tau
dxy  = Ld/(nxy-1)
dt   = dxy^2/alpha/4*cfl
nt   = ceil(Int,tend/dt)
dt   = tend/nt
np   = ceil(Int,1/cfl)*10

F = zeros(Float64, nt,nxy,nxy)
S = ones(Float64, nt,nxy,nxy)

x = range(0,stop=Ld, length=nxy)
y = range(0,stop=Ld, length=nxy)
i = 2:nxy-1
j=i

a = Animation()

for it in 2:nt
    @. F[it,i,j] = F[it-1,i,j] +
                   (alpha*dt/dxy^2)*(F[it-1,i-1,j]-2*F[it-1,i,j]+F[it-1,i+1,j])+
                   (alpha*dt/dxy^2)*(F[it-1,i,j-1]-2*F[it-1,i,j]+F[it-1,i,j+1])+
                   S[it-1,i,j]
    if it%np==0
        p = contour(x,y,F[it,:,:], fill=true,
                    c=:viridis, aspect_ratio=1.0, xlimits=(0,1))
        frame(a,p)
    end
end

p = contour!(x,y,F[end,:,:], fill=true, c=:viridis, aspect_ratio=1.0, xlimits=(0,1))
```