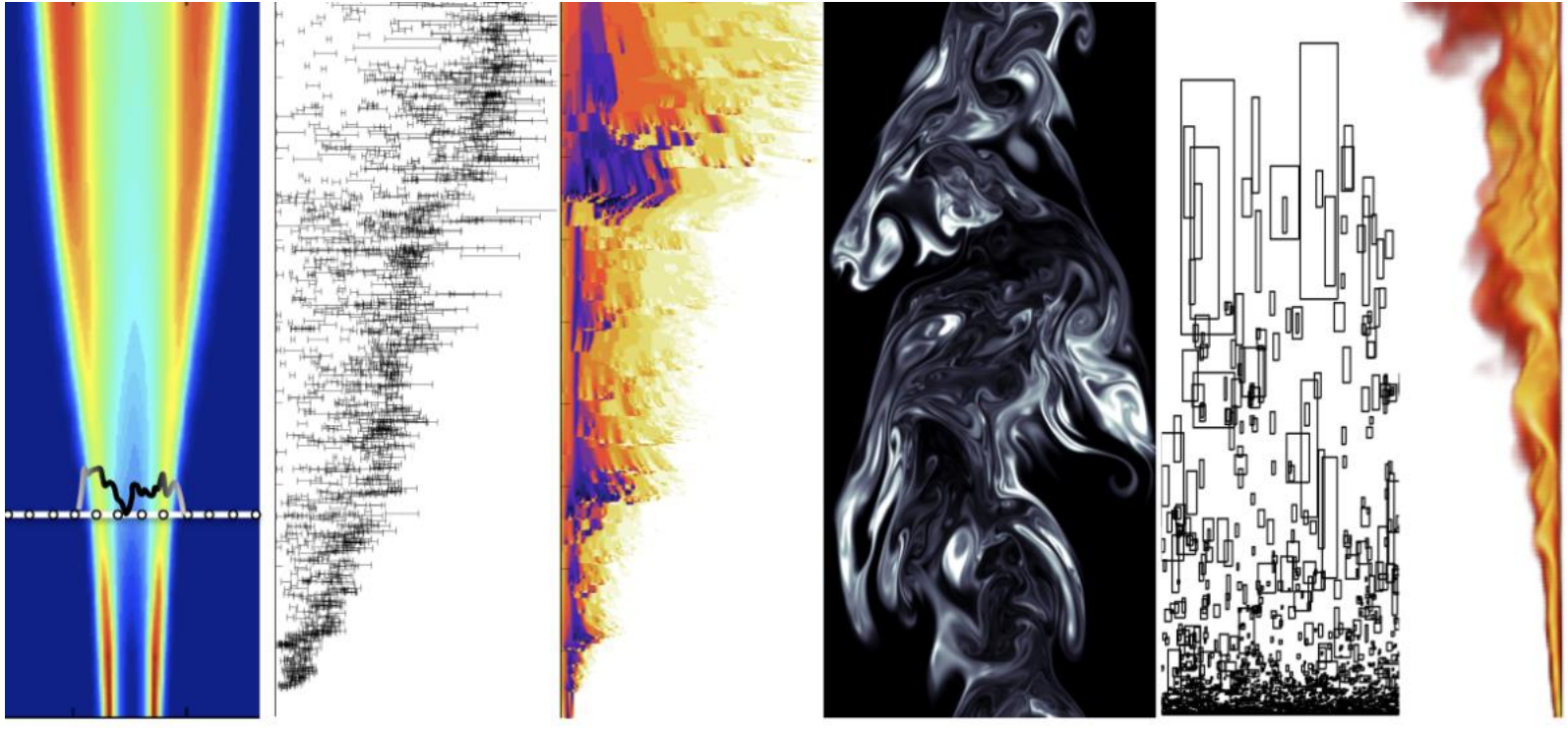
ChEn 541 Numerical Methods



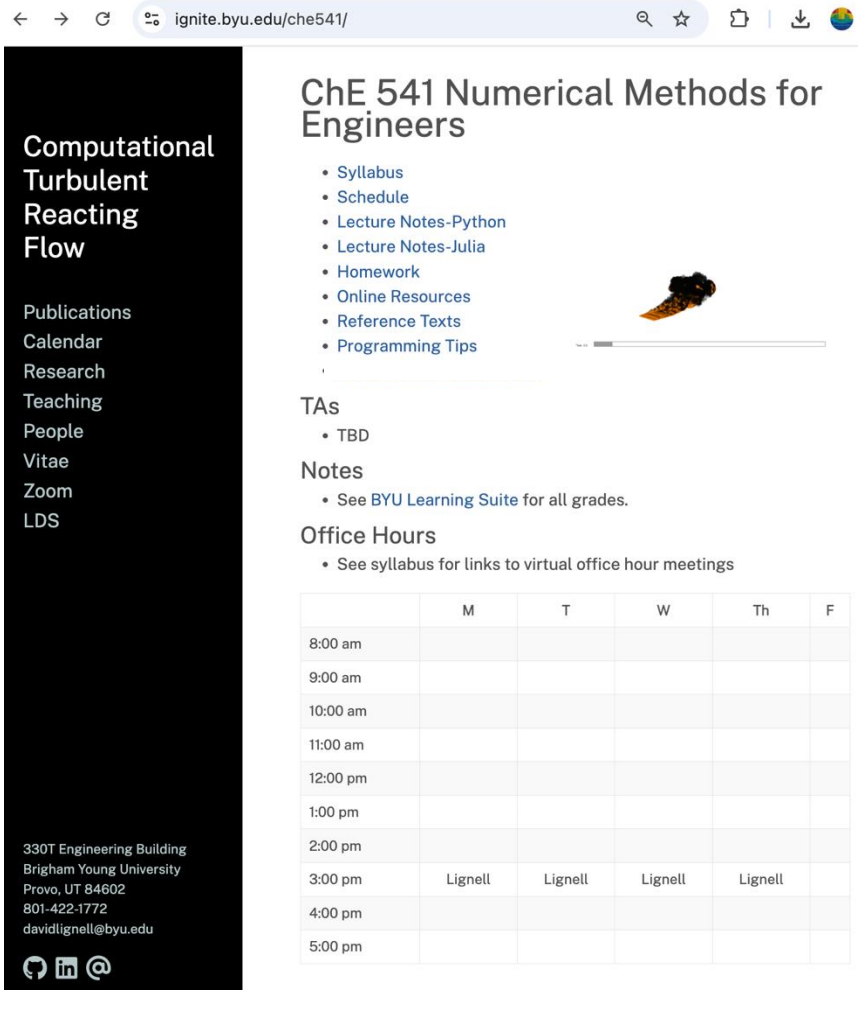
Office

- EB 330T—Chemical Engineering
- (801) 422-1772 (work)
- (801) 420-1468 (cell)
- davidlignell@byu.edu
- ignite.byu.edu

TA

- Fletcher Smith
- fletchs9@byu.edu

ignite.byu.edu/che541



- Syllabus
- Schedule
- Lectures
- Homework
- Books/Materials
- Languages

Topics

Numerics

Precision
Roundoff
Convergence
Stability

Tools

Algebraic Systems

Linear
-Direct
-Iterative
-Special
Nonlinear
-Single
-Systems

ODEs

IVP
BVP
1st order
nth order
Boundaries
Implicit
Explicit
Discretize

PDEs

Parabolic
Elliptic
Hyperbolic
Finite Diff
Finite Vol
Spectral

Integration

Interpolation

Curve Fitting

Special

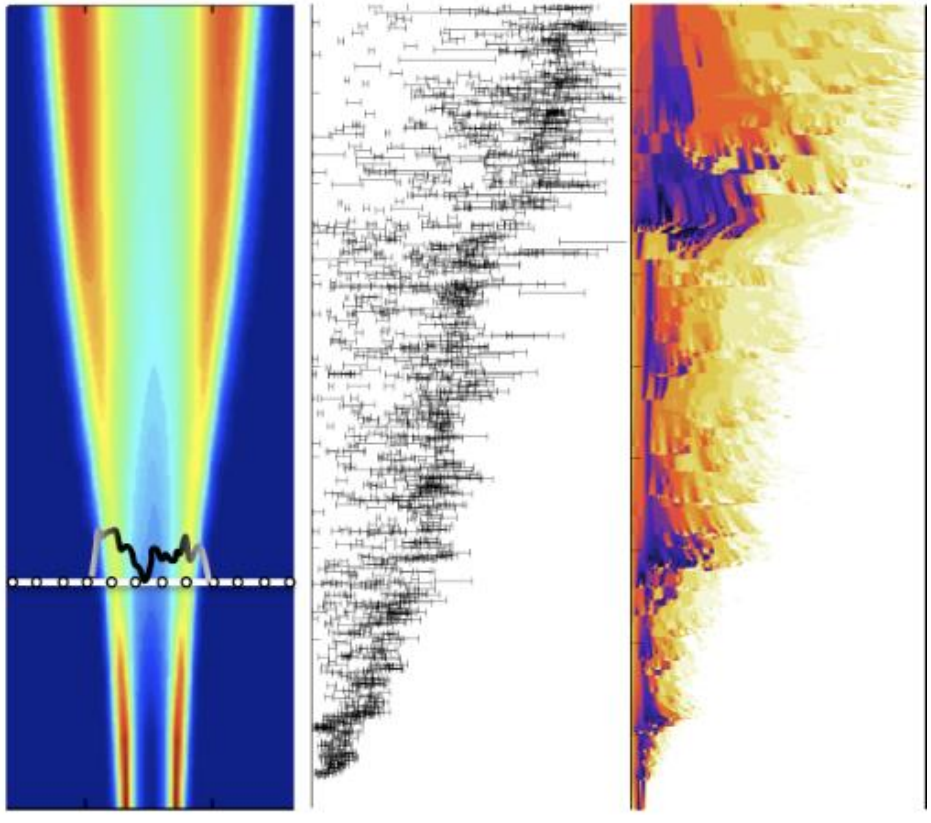
Applications

- **With a partner:**
  - **List engineering problems requiring numerical methods for solution**
- **What type of methods?**
- **Names of methods?**

- Adiabatic flame temperature
  - Given an enthalpy, find temperature
- Chemical equilibrium
- Steady distillation column
  - Unsteady?
- PSR: steady, unsteady
- PFR: steady, unsteady
- Conductive HT (1D, 2D, SS, US)

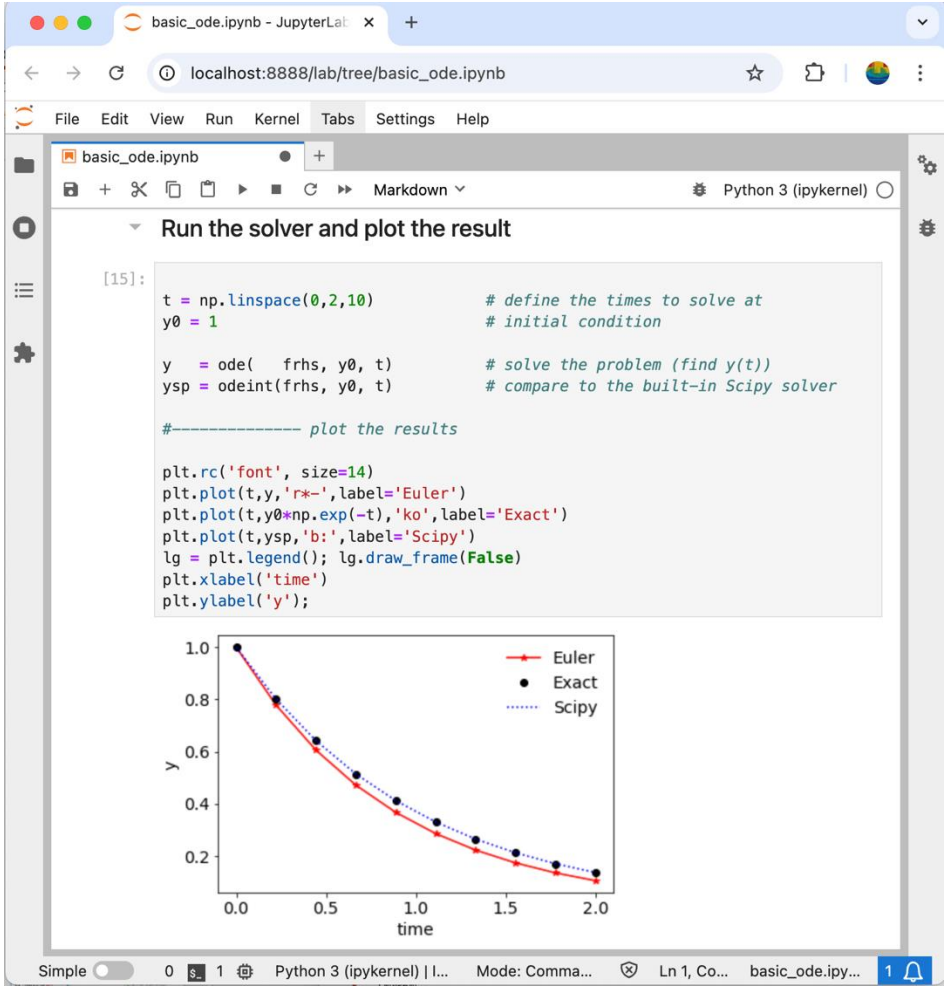
Research Example

- One-Dimensional Turbulence
- Parabolic PDEs
  - u, v, w, energy, yi
- Complex chemistry
- Multicomponent mass transfer
- Radiative heat transfer
- Stochastic
- Methods:
  - Finite volume discretization
  - Method of lines
  - Operator splitting
  - Stiff ODE integration
    - Nonlinear → linear algebraic systems
  - Interpolation
  - Mesh adaption
  - Others...

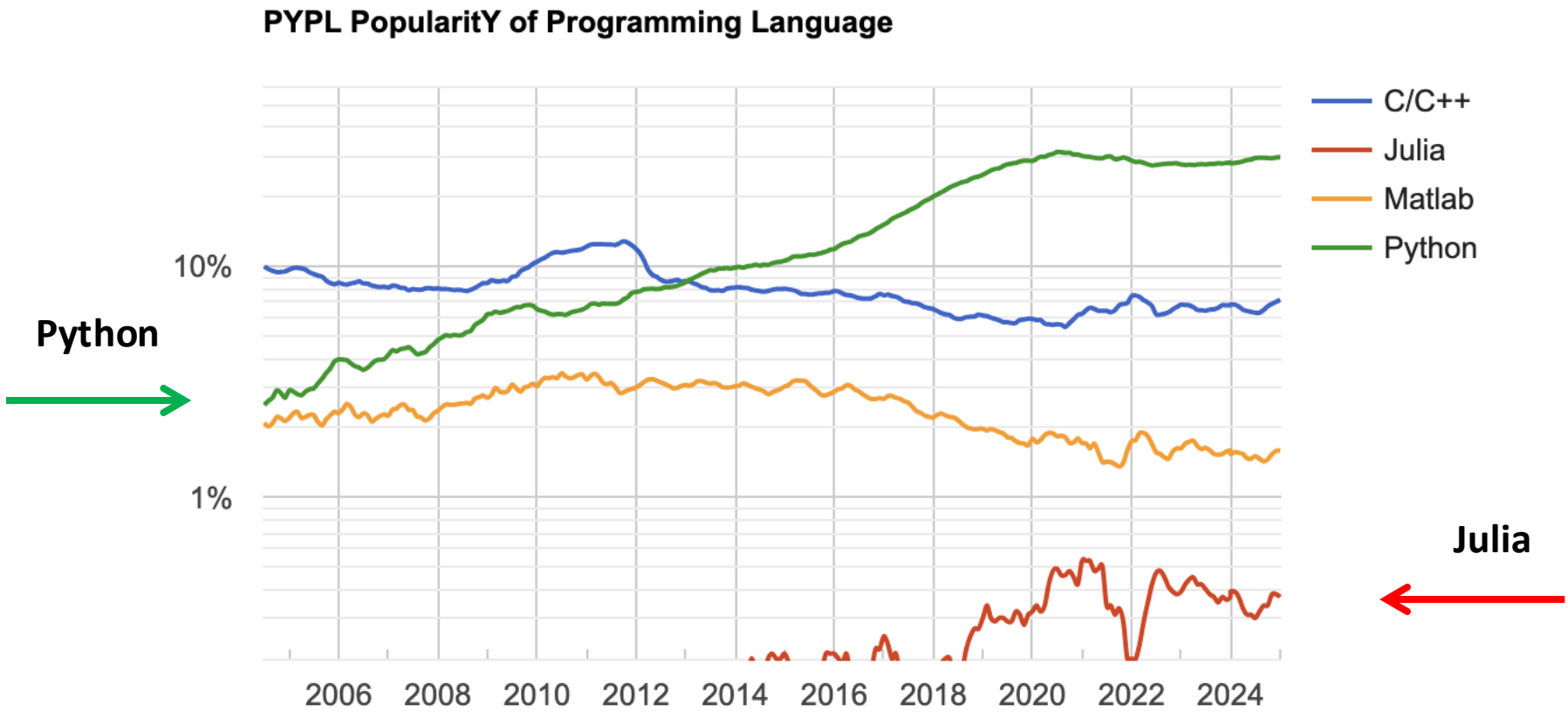


Software

- Languages
  - Python (numpy)
  - Julia
  - (Matlab)
- Development environment
  - Prefer Jupyter or JupyterLab
  - Combine documentation/equations, code, plots
- Anaconda
- Google Colab
- Conventions
  - HW as single html file
  - No spaces in file names
  - Prettify your code (alignment, whitespace, variable names)



Languages


PYPL PopularitY of Programming Language
C/C++
Julia
Matlab
Python
10%
1%
2006
2008
2010
2012
2014
2016
2018
2020
2022
2024
Python
Julia

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
```

Compare Languages: 2D Unsteady Heat Eqn

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))
```