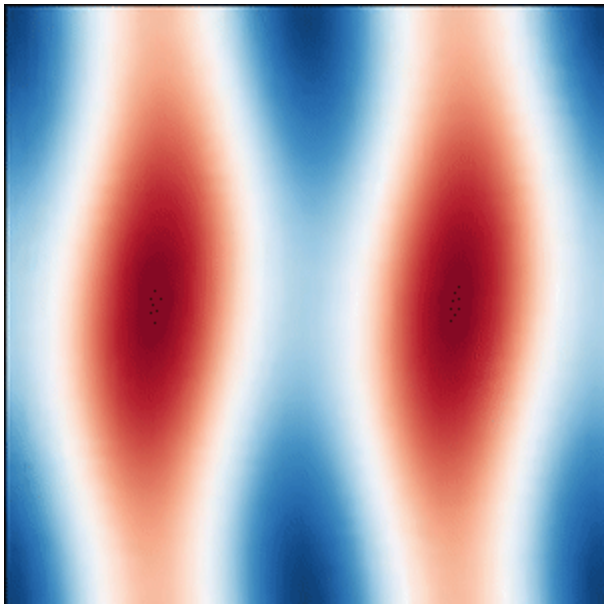
Create Your Own Navier-Stokes Spectral Method Fluid Simulation (With Python)

Philip Mocz Follow 7 min read · Aug 4, 2023

For today's recreational coding exercise, we solve the Navier-Stokes equations for an incompressible viscous fluid. To do so, we will implement a spectral method. The Navier-Stokes equations approximately describe the motions of a fluid such as water. The equations can capture the phenomenon of *turbulence*. The system is also of interest in mathematics as a general proof for the existence and smoothness of solutions in 3D is one of the open Millenium Prize Problems.

You may find the accompanying Python code on GitHub.

Before diving in, below is a gif of what running our simulation looks like:



. . .

The Navier-Stokes Equations

We will consider the evolution of the velocity field ***v(x)*** of an incompressible fluid with viscosity ν, governed by the Navier-Stokes equations:

$$\frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v} \cdot \nabla)\,\mathbf{v} = \nu \nabla^2 \mathbf{v} - \nabla P$$

$$\nabla \cdot \mathbf{v} = 0$$

where P is the fluid pressure. The pressure field in the case of *incompressible* fluids is special: it is not a separate variable that is evolved in time (unlike in the case of compressible fluids). Instead, the pressure is the solution that keeps the velocity field divergence-free (the second equation). The velocity field experiences *advection* — the non-linear term on the left-hand side of the first equation — which can cause compression. The fluid also experiences *diffusion* due to the viscosity coefficient ν. The larger the viscosity coefficient, the 'stickier' the fluid becomes, like honey.

Spectral Method

A *spectral method* represents the numerical solution to a partial differential equation as a sum of global basis functions with coefficients as the unknowns to be solved for. Here we will consider the Fourier basis, which is ideal for representing continuous solutions on a periodic domain. Such a method can then use the Fast Fourier Transform (FFT) algorithm to transform the solution on the physical domain $\boldsymbol{x}$ into Fourier space, where the solution becomes a sum of waves each described by a wavenumber $\boldsymbol{k}$. We've seen this approach in a previous tutorial on solving the time-dependent Schrodinger equation. We use the 'hat' notation to depict the Fourier coefficients of the velocity field:

$$\hat{\mathbf{v}}(\mathbf{k}) = \mathrm{FFT}\left(\mathbf{v}(\mathbf{x})\right)$$

What makes the spectral method nice? One special property is that the evaluations of gradients and related operators turn into multiplication by $\boldsymbol{k}$ in Fourier space. For example, applying the Fourier transform to the Navier-Stokes equations, where the velocity transforms as:

$$\mathbf{v}(\mathbf{x}) \rightarrow \hat{\mathbf{v}}(\mathbf{k})$$

the gradient ∇ transforms as:

$$\nabla \rightarrow i\mathbf{k}$$

and the Laplacian transforms as:

$$\nabla^2 \rightarrow -k^2$$

Thus, we can write functions to calculate operators like gradients, divergences, and curls of fields as follows:

Calculating the gradient ∇v using FFTs:

```
def grad(v, kx, ky):
  """ return gradient of v """
  v_hat = np.fft.fftn(v)
  dvx = np.real(np.fft.ifftn( 1j*kx * v_hat))
  dvy = np.real(np.fft.ifftn( 1j*ky * v_hat))
  return dvx, dvy
```

grad.py hosted with ❤ by GitHub view raw

Calculating the divergence $\nabla . v$ using FFTs:

```
def div(vx, vy, kx, ky):
  """ return divergence of (vx,vy) """
  dvx_x = np.real(np.fft.ifftn( 1j*kx * np.fft.fftn(vx)))
  dvy_y = np.real(np.fft.ifftn( 1j*ky * np.fft.fftn(vy)))
  return dvx_x + dvy_y
```

div.py hosted with ❤ by GitHub view raw

Calculating the curl $\nabla \times \boldsymbol{v}$ using FFTs:

```
def curl(vx, vy, kx, ky):
  """ return curl of (vx,vy) """
  dvx_y = np.real(np.fft.ifftn( 1j*ky * np.fft.fftn(vx)))
  dvy_x = np.real(np.fft.ifftn( 1j*kx * np.fft.fftn(vy)))
  return dvy_x - dvx_y

```

curl.py hosted with ❤ by GitHub view raw

The above are quite useful functions to have on hand to compute derivatives of any periodic function. They make use of some Fourier space variables we have defined, including a 2D array of all the wave numbers k:

```python
N = 400   # spatial resolution
L = 1     # box size

# Fourier Space Variables
klin = 2.0 * np.pi / L * np.arange(-N/2, N/2)
kmax = np.max(klin)
kx, ky = np.meshgrid(klin, klin)
kx = np.fft.ifftshift(kx)
ky = np.fft.ifftshift(ky)
kSq = kx**2 + ky**2
kSq_inv = 1.0 / kSq
kSq_inv[kSq==0] = 1

# dealias filter (2/3 rule)
dealias = (np.abs(kx) < (2./3.)*kmax) & (np.abs(ky) < (2./3.)*kmax)
```

fourier_space.py hosted with ❤️ by GitHub view raw

Let's now outline our main spectral method. It will consist of:

1. An **advection step** in physical space using a forward Euler step, including the application of a *de-aliasing* filter
2. A **Poisson solve** in Fourier space to get the pressure
3. A **correction step** to apply the pressure gradient to keep the fluid velocity divergence-free
4. A **diffusion solve** in Fourier space (implicit backward Euler)

That is, the main loop of the algorithm looks like the following:

```python
#Main Loop
for i in range(Nt):

  # Advection: rhs = -(v.grad)v
  dvx_x, dvx_y = grad(vx, kx, ky)
  dvy_x, dvy_y = grad(vy, kx, ky)

  rhs_x = -(vx * dvx_x + vy * dvx_y)
  rhs_y = -(vx * dvy_x + vy * dvy_y)

  rhs_x = apply_dealias(rhs_x, dealias)
  rhs_y = apply_dealias(rhs_y, dealias)

  vx += dt * rhs_x
  vy += dt * rhs_y

  # Poisson solve for pressure
  div_rhs = div(rhs_x, rhs_y, kx, ky)
  P = poisson_solve( div_rhs, kSq_inv )
  dPx, dPy = grad(P, kx, ky)

  # Correction (to eliminate divergence component of velocity)
  vx += - dt * dPx
  vy += - dt * dPy

  # Diffusion solve (implicit)
  vx = diffusion_solve( vx, dt, nu, kSq )
  vy = diffusion_solve( vy, dt, nu, kSq )

  # vorticity (for plotting)
  wz = curl(vx, vy, kx, ky)
```

navier-stokes.py hosted with ❤️ by GitHub view raw

Let's go through the algorithm step-by-step. First, is the calculation of the effect of the non-linear advection term:

$$\frac{\partial \mathbf{v}}{\partial t} = -(\mathbf{v}\cdot\nabla)\mathbf{v} \equiv \mathbf{r.h.s.}$$

Gradients can be calculated in Fourier space using our helper functions, but the product has to calculated in physical space. The non-linear term can actually cause a numerical issue in some cases; namely, the product of two fields can have wavenumbers that are higher than the maximum wavenumber we can represent in our Fourier basis. To avoid issues, we can apply a *de-aliasing filter* to zero-out high-frequency modes. We do this using the well-known **2/3-rule**, which says that for products of fields represented in the Fourier basis, we can only trust the wavenumbers below 2/3 the maximum wavenumber in the basis (*aside:* if we were to multiply more than just 2 fields together, our filter would need to be stricter). Thus we implement the filter:

```
def apply_dealias(f, dealias):
  """ apply 2/3 rule dealias to field f """
  f_hat = dealias * np.fft.fftn(f)
  return np.real(np.fft.ifftn( f_hat ))
```

dealias.py hosted with ❤️ by GitHub view raw

Once we have the de-aliased righthand-side, we can evolve the velocity field from time step *n* to time step *n+1* with the forward Euler method:

$$\mathbf{v}^{(n+1)} = \mathbf{v}^{(n)} + \Delta t \cdot \mathbf{r.h.s.}^{(n)}$$

This advection step can introduce divergence in the velocity field, and it is the role of the pressure in the Navier-Stokes equations to remove it. According to the Helmholtz decomposition theorem, any vector field can be decomposed as a sum of the curl of a vector *A* plus the gradient of the scalar *P*. That is, the decomposition looks like:

$$\mathbf{r.h.s.} = \nabla \times \mathbf{A} + \nabla P$$

If we take the divergence of both sides, the curl of *A* vanishes and we are left with:

$$\nabla \cdot (\mathbf{r.h.s.}) = \nabla^2 P$$

This is a Poisson equation and we can solve for *P*. It turns out *P* is actually the fluid pressure here! (We have chosen our notation carefully.) To solve for *P*, we can transform into Fourier space, where the Laplacian operator becomes $-k^2$. We write a quick Poisson solver:

```python
def poisson_solve( rhs, kSq_inv ):
  """ solve the Poisson equation, given source field rho """
  P_hat = -(np.fft.fftn( rhs )) * kSq_inv
  P = np.real(np.fft.ifftn(P_hat))
  return P
```

poisson_solve.py hosted with ❤️ by GitHub view raw

Knowing *P,* we then find its gradient (by multiplying by *ik* in Fourier space) and update the velocity field:

$$\mathbf{v}^{(n+1)} \leftarrow \mathbf{v}^{(n+1)} - \Delta t \nabla P$$

Finally, we will apply the diffusion step:

$$\frac{\partial \mathbf{v}}{\partial t} = \nu \nabla^2 \mathbf{v}$$

In general, it is best to solve diffusion problems implicitly because explicit methods have strict limitations on stable timestep sizes. Thus we will apply a backward Euler step in Fourier space:

$$\frac{\hat{\mathbf{v}}^{(n+1)} - \hat{\mathbf{v}}^{(n)}}{\Delta t} = -\nu k^2 \hat{\mathbf{v}}^{(n+1)}$$

implemented as follows:

```python
def diffusion_solve( v, dt, nu, kSq ):
  """ solve the diffusion equation over a timestep dt, given viscosity nu """
  v_hat = (np.fft.fftn( v )) / (1.0+dt*nu*kSq)
  v = np.real(np.fft.ifftn(v_hat))
  return v
```

diffusion_solve.py hosted with ❤️ by GitHub view raw

And there we have it! That's it for the numerical method. Just a handful of lines at its core. Time to try it out!

Initial Conditions

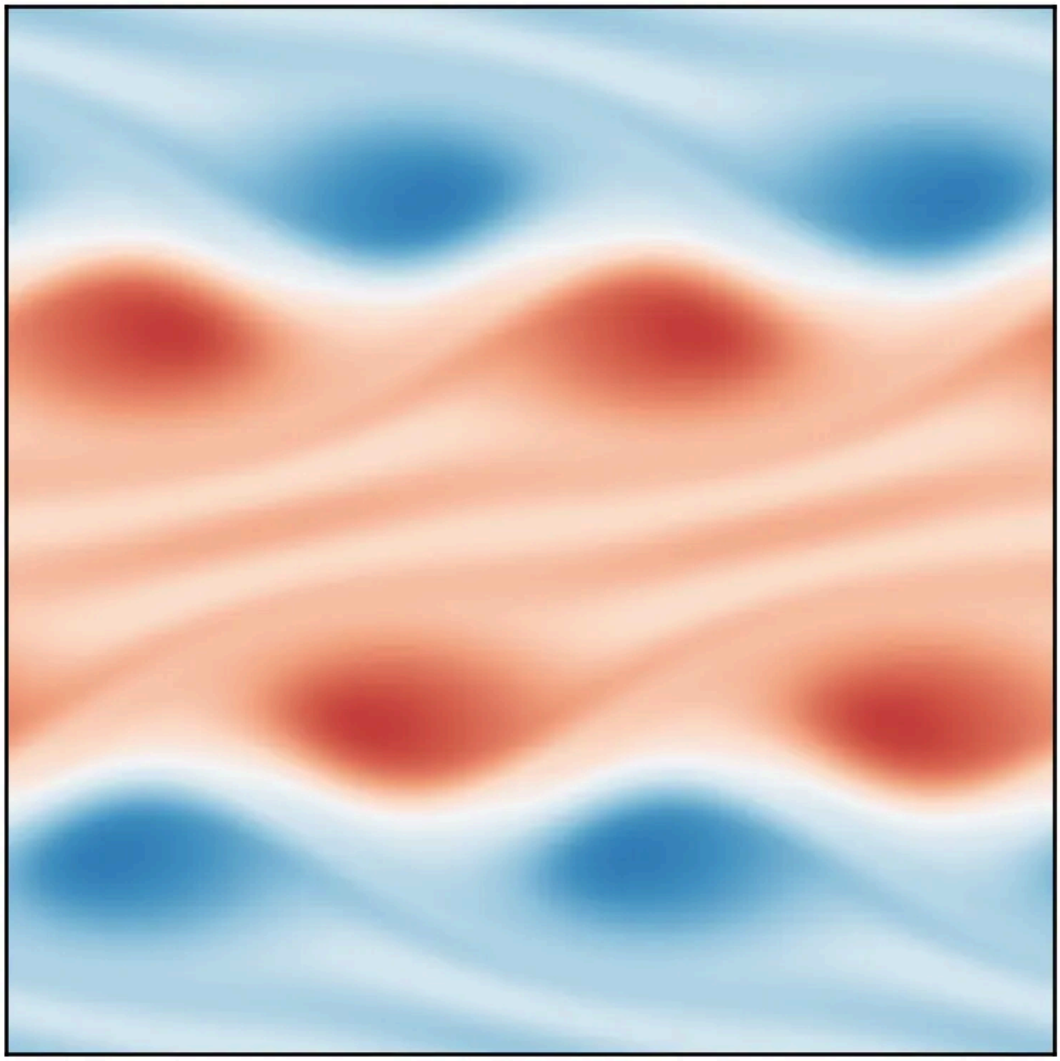
We implement a simple test problem in a 2D periodic domain $[0,1]^2$. The fluid field is initialized as a low wavenumber vortex. The evolution leads to shear and the formation of eddies, which eventually dissipate due to viscosity.

Exercise: Improving Code Efficiency

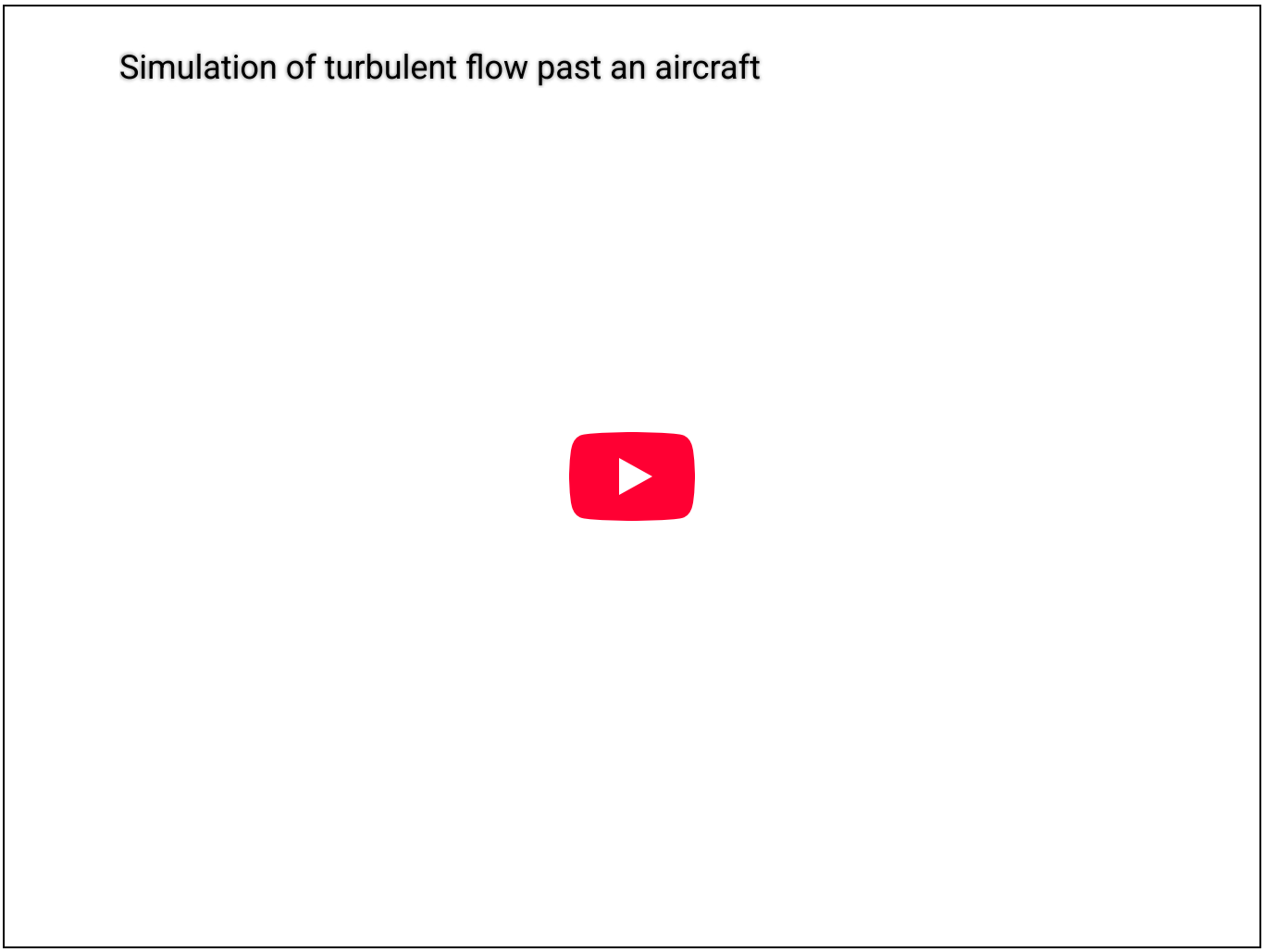
The algorithm presented here is designed to be a pedagogical demonstration of solving the different parts of the Navier-Stokes equations (advection, correction, diffusion). The code however does make redundant uses of Fourier transforms (e.g. translating velocity between physical and Fourier space each time a gradient is calculated, which then might be translated back again). It is left as an exercise to the reader to improve code efficiency by minimizing the number of FFT calls.

. . .

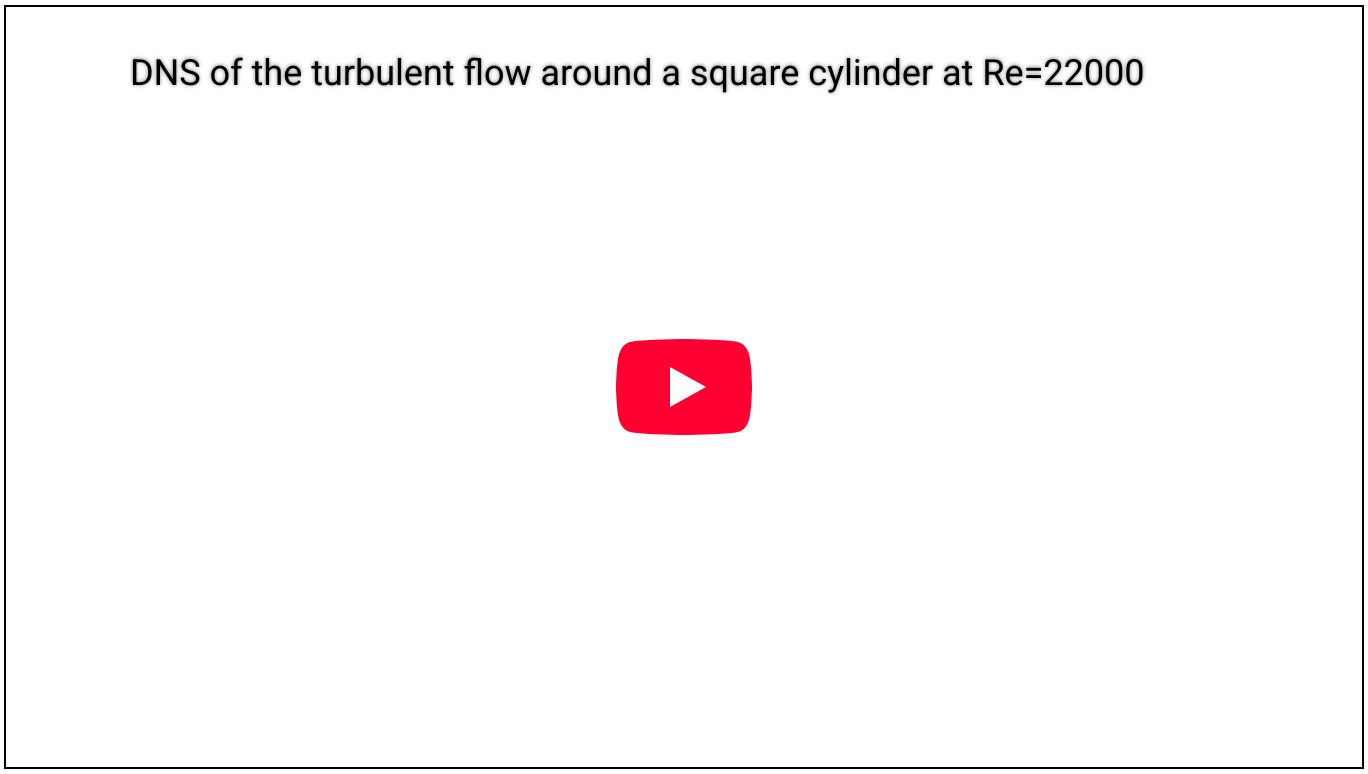
Running the code allows you to visualize the simulation in real time and will yield the figure for the fluid's vorticity (curl of the velocity):



The incompressible Navier-Stokes equations can be solved using a variety of numerical techniques, including finite element, finite different, spectral methods, and lattice Boltzmann methods. Simulating the equations has important applications in engineering. For example, aircraft design is benefitted from the technology of simulating turbulent flow across the full body of the plane, and such computations can reduce the need for physical wind tunnel experiments.



The Navier-Stokes equations describe an idealized, continuous fluid but fall short of a perfect representation of reality. For example, the equations can predict infinite velocity at sharp corners (the so-called ‘corner singularity’) once boundary conditions are introduced! The question of whether the equations can blow up in pathological cases on a periodic domain remains an open problem in mathematics. Nevertheless, the Navier-Stokes equations remarkably encapsulate the intricate phenomenon of turbulence — a complex and chaotic behavior that emerges in fluid flows characterized by high Reynolds numbers.



. . .

Download the Python code on GitHub for our Navier-Stokes solver to visualize the simulation in real-time and play around with the initial conditions. Enjoy!

About the Author

Philip Mocz is a computational physicist at Lawrence Livermore National Laboratory. He received a Ph.D. (’17) and A.B. (’12) from Harvard University, where he designed high-performance computing algorithms and simulations of astrophysical systems for his dissertation. If you’ve enjoyed this post, 👏clap for the story and 👉 follow me for more Python tutorials on a range of scientific computing topics!

Python Simulation Physics Fluid Mechanics Coding

Published in Level Up Coding

277K followers · Last published 8 hours ago

Follow

Coding tutorials and news. The developer homepage gitconnected.com && skilled.dev && levelup.dev

Written by Philip Mocz

2K followers · 63 following

Follow

Computational Physicist. Sharing intro tutorials on creating your own computer simulations! Harvard ’12 (A.B), ’17 (PhD). Connect with me @PMocz

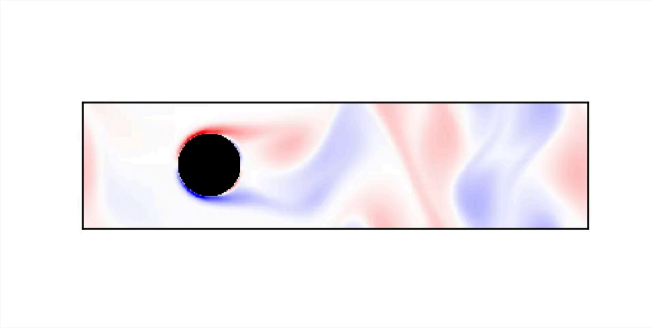
Responses (8)

Mr Mrmysterious

What are your thoughts?

See all responses

More from Philip Mocz and Level Up Coding



In The Startup by Philip Mocz

Create Your Own Lattice Boltzmann Simulation (With Python)

For today's recreational coding exercise, we simulate fluid flow past a cylinder using the...

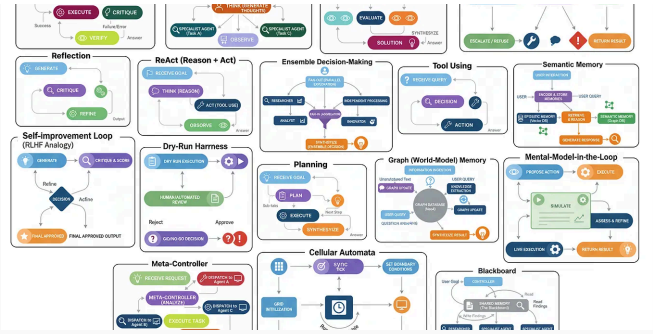
Dec 21, 2020 585 6

In Level Up Coding by Jeffrey Bakker

I'm a middle-aged developer, and the way I shine has changed

And what matters to me has also changed

Oct 6 8.4K 161



In Level Up Coding by Fareed Khan

Building 17 Agentic AI Patterns and Their Role in Large-Scale AI...

Ensembling, Meta-Control, ToT, Reflexive, PEV and more

Sep 25 2.1K 45

In Level Up Coding by Philip Mocz

Create Your Own Finite Volume Fluid Simulation (With Python)

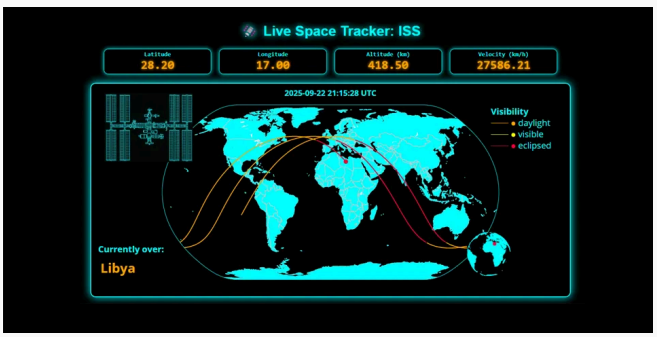
For today's recreational coding exercise, we will simulate the Kelvin-Helmholtz Instability...

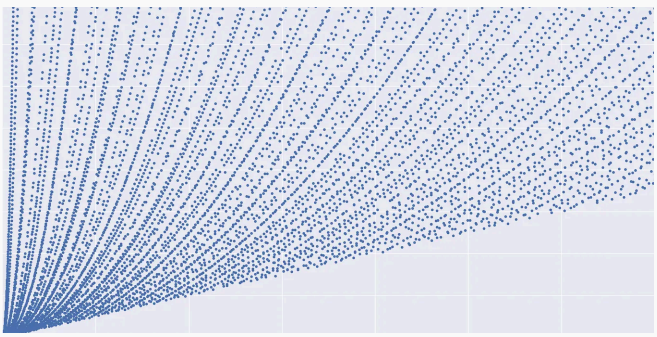
Oct 1, 2020 432 4

See all from Philip Mocz

See all from Level Up Coding

Recommended from Medium





In Data Science Collective by Lee Vaughan

Build a Sleek Sci-Fi Dashboard with Python and Dash

From movie Inspiration to ISS Tracker

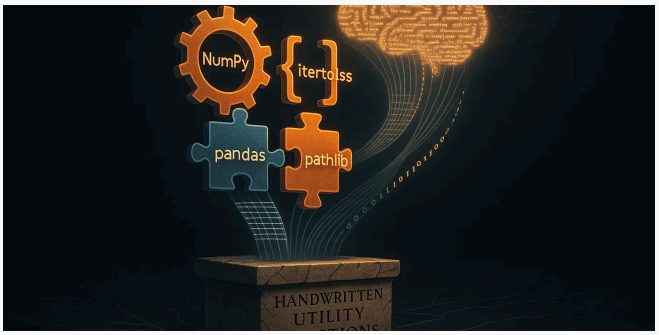
6d ago 599 7

In Python in Plain English by Aurel Nicolae

Generate Primitive Pythagorean Triples in Python

I was inspired by Polar Pi's video "Formula for Generating ALL Pythagorean Triples" to writ...

Jul 31 50



In Top Python Libraries by Abdur Rahman

9 Python Libraries So Smart, I Stopped Writing Utility Functions

Why spend hours writing boilerplate when these do it for you?

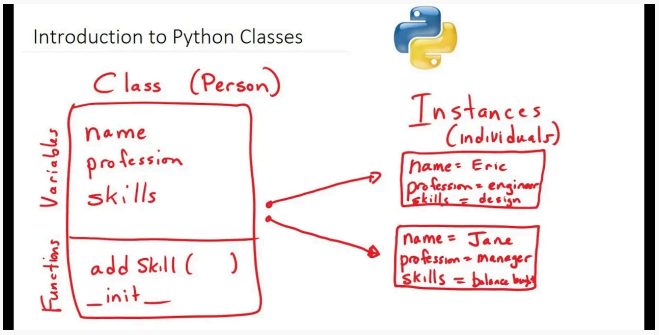
Aug 2 1K 11

In AlgoMart by Yash Jain

Anaconda vs. uv: Which Python Environment Tool Should You Use?

For developers working with Python in serious projects—whether it's scientific...

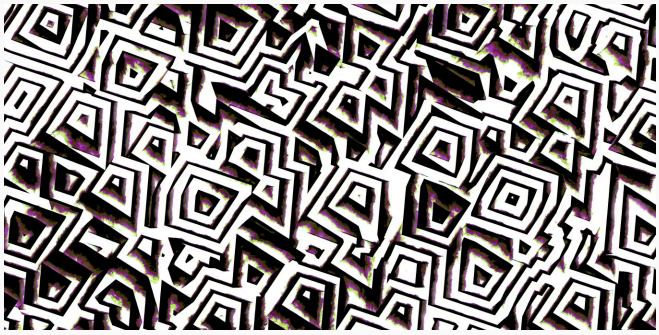
Jun 8 93



In PyZilla | Python by Azeem Teli

Classes in Python? Nah Bro, I Found a Simpler Way 🧘

Ditching OOP drama for cleaner, simpler code—one dictionary at a time.



In Science Spectrum by Cole Frederick

Gaps Between Prime Numbers

Is there a pattern?