Member-only story

A Simple Method for Numerical Integration in Python

Step-by-step coding example for approximating solutions to systems of ordinary differential equations

Zack Fizell · Following

Published in Towards Data Science · 6 min read · Jun 13, 2022

Photo by Sanjeevan SatheesKumar on Unsplash

Numerical integration is a technique that is utilized to approximate solutions for ordinary differential equations (ODEs). There are a range of methods for numerical integration; these vary in speed, accuracy, and complexity. To name a few, there is Euler's method, Runge-Kutta methods, and the trapezoidal rule. Thankfully, as a programmer or engineer, you do not need to know the exact details of how these methods work to get a good solution for an ODE or system of ODEs.

Before diving right in to the code, let's briefly cover what an ordinary differential equation is and how numerical integration works. These are differential equations that take the following form:

$$y^{(n)} = F(x, y, y', \cdots, y^{(n-1)})$$

where,

$$y = f(x)$$

Here, *y* is an unknown function of *x* (we are trying to approximate this function at certain *x* values) and *F* is a function of *x*, *y*, and the derivatives of *y* (what we have or were given). *n* represents the order, or highest derivative, of the differential equation. The function, *F*, is usually a well-known equation that was derived by a physicist, mathematician, scientist, etc. For example, you could be solving for the position of a particle given a function for its velocity. In this example, time would be equivalent to *x* in the equations above and the velocity would be equivalent to *F*.

Similarly, you could have a system of ODEs that you are looking to solve (or approximate). The equations would take the following form:

$$\begin{pmatrix} y_1^{(n)} \\ y_2^{(n)} \\ \cdots \\ y_m^{(n)} \end{pmatrix} = \begin{pmatrix} f_1(x, \vec{y}, \vec{y}\,', \cdots, \vec{y}^{\,(n-1)}) \\ f_2(x, \vec{y}, \vec{y}\,', \cdots, \vec{y}^{\,(n-1)}) \\ \cdots \\ f_m(x, \vec{y}, \vec{y}\,', \cdots, \vec{y}^{\,(n-1)}) \end{pmatrix}$$

where,

$$\vec{y} = [y_1(x), y_2(x), \cdots, y_m(x)]$$

Here, we have a set of *y* values (quantity of *m*) that we are interested in, each with their own derivatives and unknown solution.

One of the most common problems involving ODEs are initial value problems (IVPs). In these problems, you would be give an initial value for *y* and an ODE in the form given above. Typically, you would be working with a first order ODE. For a first order ODE, numerical methods take the ODE and treat it as a slope. Using this slope and a very small increment on *x*, the initial value of *y* can be incremented to approximate the next value of *y*. This process is repeated, until you have found all of the values of *y* that you are interested in. If the step size of *x* is chosen carefully, then the approximation can yield very accurate results. This holds true for a system of ODEs as well.

For this article, we will solve a set of three (*m*) first (*n*) order ordinary differential equations. These equations are arbitrary, so substitute your own as needed.

$$\begin{pmatrix} I' \\ L' \\ P' \end{pmatrix} = \begin{pmatrix} \frac{1}{2}P \\ 2(LP)^{1/3} \\ \frac{100}{I} \end{pmatrix}$$

Here, the tick marks indicate a single time derivative on our variables. We will be solving our system of ODEs as an initial value problem. This means we will pass initial values for *I*, *L*, and *P* into our numerical integrator. Using

these values, our numerical integrator will take small steps through time and use the derivatives to solve for our variables at each time step. I will go into more details about the numerical integrator and its capabilities in the coding section. Let's jump into the code!

. . .

Importing Libraries

Here, we are simply importing the necessary libraries and functions to run this code.

- `solve_ivp` from the *SciPy* library is used for solving initial value problems and numerical integration
- `pyplot` from *matplotlib* is used to plot the results from numerical integration (defined as `plt` for ease of calling)

```
# Importing Packages
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
```

Creating a User-Defined Function

The next step is defining our system of ODEs as a user-defined Python function called `model`. This will allow `solve_ivp` to utilize the function when numerically integrating. You will see in the code below that we first pull the *I*, *L*, and *P* variables from the *Y* vector that is passed into the function (`solve_ivp` passes *t* and *Y* behind the scenes). We then use these values to solve for the time derivatives we defined earlier in this article. We create a list for these derivatives and return them.

```
# Model for solve_ivp
def model(t, Y):
    I = Y[0]
    L = Y[1]
    P = Y[2]
    dIdt = 1/2*P
    dLdt = 2*(L*P)**(1/3)
    dPdt = 100/I
    dYdt = [dIdt, dLdt, dPdt]
    return dYdt
```

Numerical Integrating and Obtaining Solution

As stated earlier in the article, we are going to be using `solve_ivp` to numerically integrate our system of ODEs. The `solve_ivp` function has a couple required arguments: the user-defined function, `model`, a time span tuple, `tspan`, and a set of initial conditions, `Y0`. The time span and initial conditions were chosen at random, so feel free to mess around with those and see what results you obtain. We can also pass in additional optional arguments that will help us define what numerical method and accuracy we would like our solution to have. We will choose the Runge-Kutta method of order 5(4) and a relative tolerance (relative accuracy) of $1/10^{10}$. There are plenty of other options for `solve_ivp`. Check out the documentation here.

```
# Initial Conditions
Y0 = [0.5, 0.2, 0.1]  # [I0, L0, P0]

# Time Span of Interest
tspan = (0, 5)  # (t0, tf)

# Solving ODE
sol = solve_ivp(model, tspan, Y0, method='RK45', rtol=1e-10)
I_sol, L_sol, P_sol = sol.y
time = sol.t
```

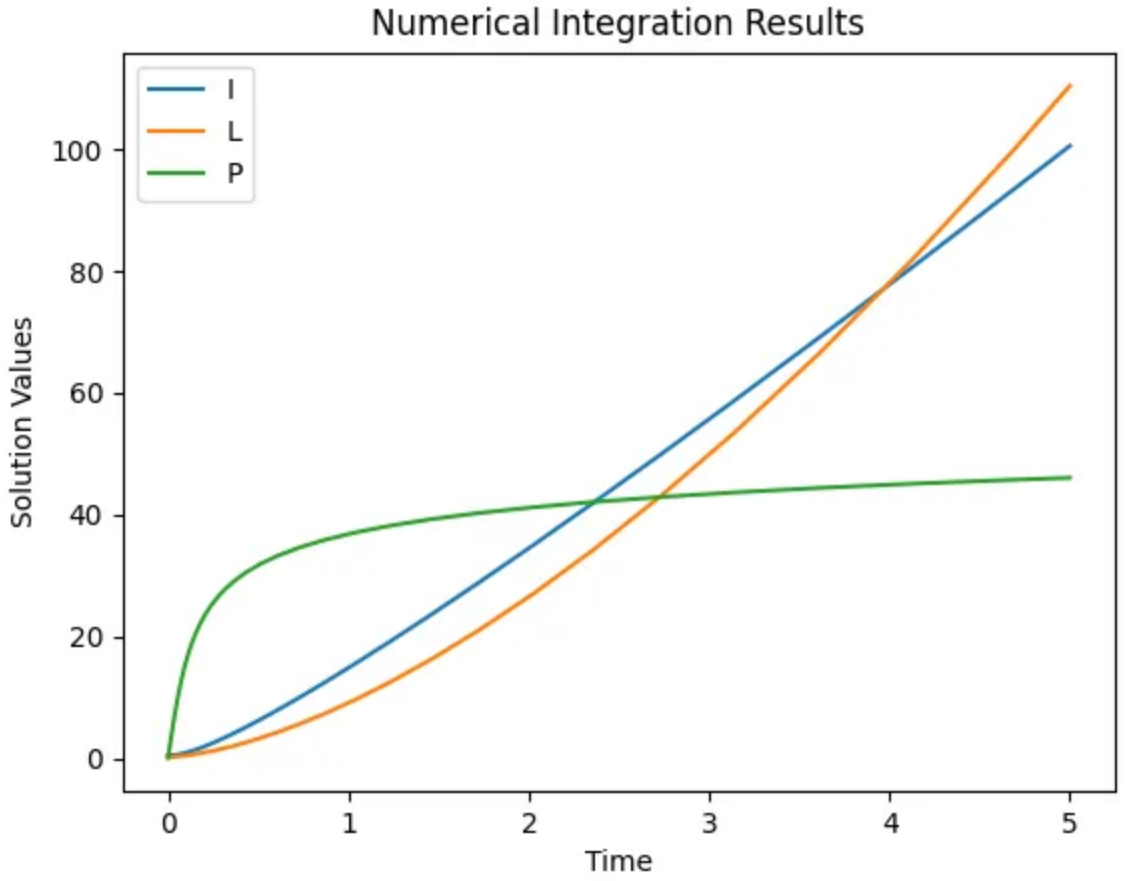
Once we run the numerical integrator, we will store the results in a variable, `sol`. We can then pull the time histories, or approximated solutions, of our variables *I*, *L*, and *P* using the dot operator. Additionally, we can pull the time steps that our variables were solved at.

Plotting Results

Finally, we can visualize what our output from `solve_ivp` looks like using *Pyplot*. Using the time and time history data that we gathered, we can plot each of the variables on the same plot. No plot is complete without a title, axis labels, or legend, so we will include those as well.

```
# Plotting Results
plt.plot(time, I_sol)
plt.plot(time, L_sol)
plt.plot(time, P_sol)
plt.title('Numerical Integration Results')
plt.xlabel('Time')
plt.ylabel('Solution Values')
plt.legend(['I', 'L', 'P'])
plt.show()
```

The output of this portion of code shows the approximated solutions for I*, L*, and *P* using the numerical integrator.



Numerical Integration Results [Created by Author]

With that, we have successfully solved our original set of ordinary differential equations using `solve_ivp`. As you can see, the process is relatively simple; all that you need is to set up the arguments for the numerical integrator. Once you do that, it is smooth sailing.

. . .

That concludes this article on a simple method for numerical integration of ODEs in Python. Hopefully, you learned something and can apply this to your projects. Leave a comment if you have any questions! Please leave a clap and follow if you haven't already! Thank you!

Python Science Technology Data Science Physics

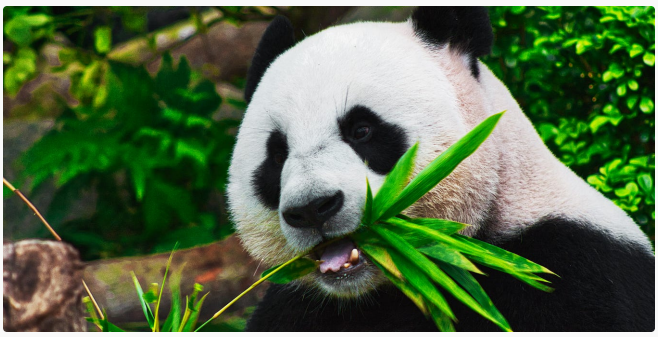
Written by Zack Fizell

Following

1.4K Followers · Writer for Towards Data Science

M.S. in Aeronautics and Astronautics — Articles on Orbital Mechanics| Machine Learning| Coding — https://medium.com/@zackfizell10/membership

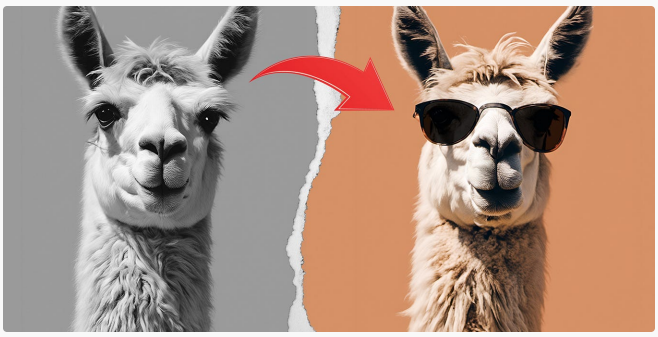
More from Zack Fizell and Towards Data Science



Zack Fizell in Towards Data Science

Easiest Way to Plot on a World Map with Pandas and GeoPandas

A comprehensive guide to read, clean, and plot geospatial data using Pandas and...



Maxime Labonne in Towards Data Science

A Beginner's Guide to LLM Fine-Tuning

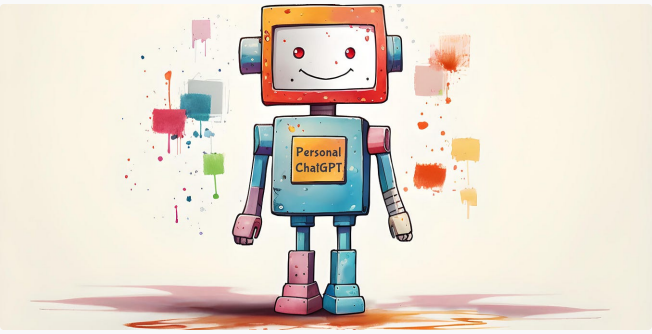
How to fine-tune Llama and other LLMs with one tool

· 6 min read · Mar 10, 2022

246 3

· 8 min read · Aug 30

411 5



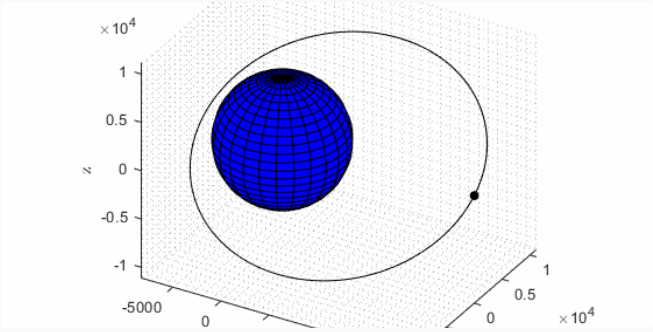
Robert A. Gonsalves in Towards Data Science

Your Own Personal ChatGPT

How you can fine-tune OpenAI's GPT-3.5 Turbo model to perform new tasks using you...

· 15 min read · Sep 7

564 7



Zack Fizell in Towards Data Science

How to Use MATLAB to Create Two-Body Orbits

Step by step walkthrough on using MATLAB to determine how a spacecraft moves under...

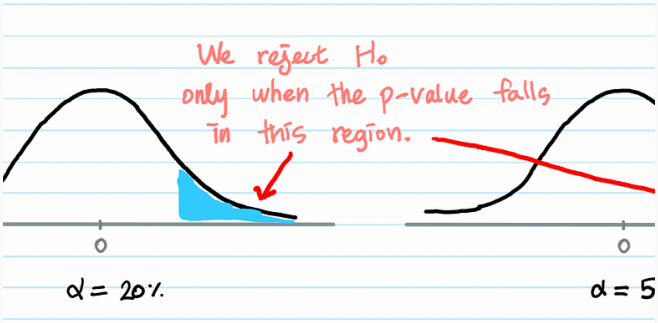
· 6 min read · Jul 21, 2022

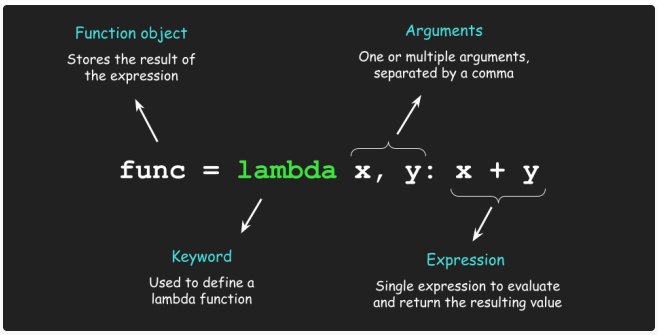
254

See all from Zack Fizell

See all from Towards Data Science

Recommended from Medium





Ms Aerin in IntuitionMath

Chi Square Test—Intuition, Examples, and Step-by-Step...

The best way to see if two variables are related.

✦ · 15 min read · Feb 12

377 3

Ernest Asena

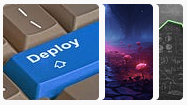
Lambda Functions in Python: Unleashing the Magic of Concise...

Welcome to a magical journey through the world of Python lambda functions, where...

4 min read · Aug 23

8

Lists

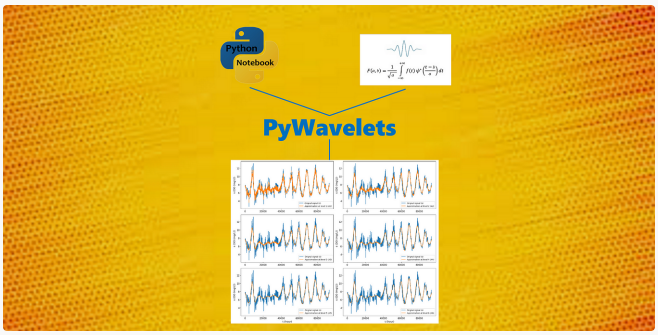


Predictive Modeling w/ Python
20 stories · 397 saves

ChatGPT
21 stories · 156 saves

Coding & Development
11 stories · 181 saves

ChatGPT prompts
24 stories · 396 saves




Dr. Shouke Wei

Multilevel Discrete Wavelet Transform and Noise Reduction o...

How to make multilevel Discrete Wavelet Transform and noise reduction of 1D Time...

· 8 min read · Mar 21

110 3


Cris Velasquez, MSc.

Riding the Waves of Stock Prices with Wavelet Transform Signals in...

Towards Unlocking Market Signals for Clearer Trading Insights

· 9 min read · Sep 11

240 3


Rohit Saroj

How to Turn Your Python Script into an Executable File

.py to .exe quickly!

4 min read · Jul 5

116

TechClaw

Python Regex: Unleashing the Power of Regular Expressions

In the ever-evolving world of programming, efficiency and precision are paramount....

3 min read · 6 days ago

66 2

See more recommendations