CL6L214 High Performance Computing in Earth System Science
Project Report

AGILITY OF COMMON ODE SOLVERS FOR NON-LINEAR PARTIAL DIFFERENTIAL EQUATIONS: A CASE STUDY

May 23, 2020

Pritom Sarma
18GG05007
M.Sc Geology
School of Earth, Ocean and Climate Sciences
Indian Institute of Technology, Bhubaneswar
ps12@iitbbs.ac.in

Contents

1 Introduction

Partial differential equations(PDEs) are a key mathematical tool used to describe physical phenomenon which are dynamical in nature. But not all PDEs have a well defined explicit analytical solutions, hence numerical solutions to such PDEs are of utmost importance in deciphering the underlying physical phenomenon governed by the said PDEs. Though there are several numerical methods employed in solving PDEs, in this work I focus on the 'Method of Lines' scheme of one dimensional discretization. It most often refers to the numerical construction or analysis of partial differential equations that proceeds by first discretizing the spatial derivatives only and leaving the time variable continuous. This leads to a system of ordinary differential equations to which a numerical method for initial value ordinary differential equations can be applied. This system then can easily be fed into any of the common ordinary differential equation(ODE) solvers available across languages and platforms.

In this project I look into various ODE solvers across two programming languages FORTRAN and Python to test their agility in solving large problems involving non-linear partial differential equations.

1.1 Physical problem and governing equations

I choose the problem of diffusion of a fluid across a medium whose permeability increases as a function of the pore-pressure. Such increase in permeability has been seen within crustal rocks in laboratory experiments(Yilmaz et al., 1994) and such increase in permeability is observed to be exponential with respect to the effective pressure and thus the pore pressure itself, as shown in Figure 1.

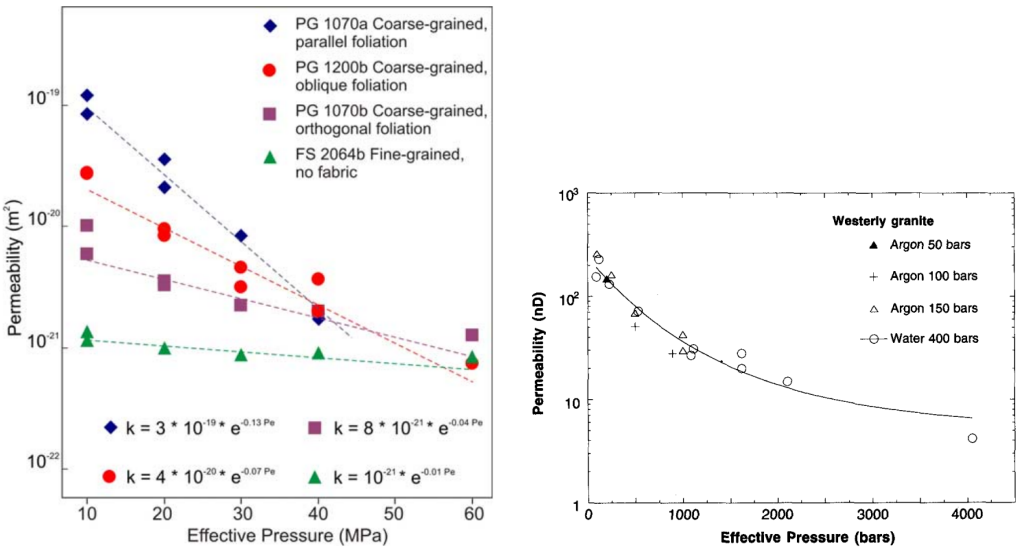

Figure 1: Yilmaz et al 1994 and De Paola et al 2009, observations on variation of permeability with effective pressure.

Thus the equation for one dimensional pressure evolution in a porous medium can be express from Darcy's law as:

$$\frac{\mu}{S_s}\frac{\partial p(x,t)}{\partial t} = \frac{\partial}{\partial x}(k\frac{\partial p(x,t)}{\partial x}) \tag{1}$$

Where, $p(x,t)$ is the pore pressure profile, k is the permeability which changes as a function of the pressure, μ is the dynamic viscosity of the fluid and S_s is the storage coefficient for the sake of simplicity we consider the storage coefficient to be constant.

The exponential scaling of permeability with pore pressure given by an empirical model (Yilmaz et al 1994, De Paola et al 2009) is of the form:

$$k = k_o e^{\gamma p} \implies \alpha = \alpha_o e^{\gamma p} \tag{2}$$

Where, k_o is the intial permeability of the medium, α_o is the initial permeability of the medium and γ is the pressure sensitivity coefficient. So Eq(1) becomes:

$$\implies \frac{\partial p(x,t)}{\partial t} = \frac{\partial}{\partial x}(\alpha(p)\frac{\partial p(x,t)}{\partial x}) \tag{3}$$

Putting α from Eq(2) in Eq(3):

$$\implies \frac{\partial p(x,t)}{\partial t} = \frac{\partial}{\partial x}(\alpha_0 e^{\gamma p}\frac{\partial p(x,t)}{\partial x})$$

$$\implies \frac{\partial p(x,t)}{\partial t} = \alpha_0 e^{\gamma p}\gamma(\frac{\partial p}{\partial x})^2 + \alpha_0 e^{\gamma p}\frac{\partial^2 p}{\partial x^2} \tag{4}$$

1.2 Solver implementations

We use a total of seven initial value problem ODE solvers, four of them were a part of the Python 3.7's SciPy package and three of them were separate solver packages of FORTRAN.

1.2.1 FORTRAN implementations

I use the following initial value solver packages in the FORTRAN implementations:

1. **Livermore Solver for Ordinary Differential Equations(LSODE)**:The LSODE solver uses adaptive numerical methods to advance a solution to a system of ordinary differential equations one time-step, given values for the variables. It solves the initial-value problem for stiff or nonstiff systems of first-order ODE's, $\frac{dy}{dt} = f(t,y)$, or, in component form, $\frac{dy}{dt} = f(i) = f(i,t,y(1),y(2),...,y(N)), i = 1,...,N$.(Radhakrishnan & Hindmarsh, 1993)

2. **Dormand-Prince of 5th Order(DOPRI5)**:The method is a member of the Runge-Kutta family of ODE solvers. More specifically, it uses six function evaluations to calculate fourth- and fifth-order accurate solutions. The difference between these solutions is then taken to be the error of the (fourth-order) solution. This error estimate is very convenient for adaptive stepsize integration algorithms. This code computes the numerical solution of a system of first order ordinary differential equations $\frac{dy}{dt} = f(x,y)$. It uses an explicit Runge-Kutta method of order (4)5 due to Dormand & Prince with step size control and dense output((Dormand & Prince, 1980).)(Hairer et al., 2000).

3. **Variable-coefficient Ordinary Differential Equation Solver(VODE)**:VODE is a new initial value ODE solver for stiff and nonstiff systems. It uses variable-coefficient Adams-Moulton and Backward Differentiation Formula (BDF) methods in Nordsieck form, as taken from the older solvers EPISODE and EPISODEB, treating the Jacobian as full or banded. Unlike the older codes, VODE has a highly flexible user interface that is nearly identical to that of the ODEPACK solver LSODE. Like its predecessors, VODE demonstrates that multistep methods with fully variable stepsizes and coefficients can outperform fixed-step-interpolatory methods on problems with widely different active time scales. In one comparison test, on a one-dimensional diurnal kinetics-transport problem with a banded internal Jacobian, the run time for VODE was 36 percent lower than that of LSODE without the J-saving algorithm and 49 percent lower with it. The fixed-leading-coefficient version ran slightly faster, by another 12 percent without J-saving and 5 percent with it. All of the runs achieved about the same accuracy.(Brown et al., 1989)

1.2.2 Python 3.7(SciPy) implementations

For the Python 3.7 implementation I use the SciPy, which is a collection of open source software for scientific computing in Python. Within the SciPy library all integrators are present inside the *scipy.integrate* and the intrinsic of *solve_ivp* is being used to solve the intial value problem ODEs. *solve_ivp* provides an option for stepsize control, integration in the complex domain for both stiff and non-stiff problems, with an option to choose from several integration methods. Following are the ones being used :

1. **Range-Kutta Method of orders 4(5) and 2(3)(RK45 & RK23)**: The Runge-Kutta methods are based on forming a one-step nonlinear approximation to the differential equation, instead of a linear methods. These schemes are based upon linear combinations of the slopes of the function. The Runge-Kutta method is popular because of its simplicity and efficiency. It is one of the most powerful predictor-corrector methods, following the form of a single predictor step and one or more corrector steps. The Range-Kutta method of 4th and 2nd order has been used here.

2. **Radau Method of 5th order**: Implicit Runge-Kutta method of the Radau IIA family of order 5. The error is controlled with a third-order accurate embedded formula. A cubic polynomial which satisfies the collocation conditions is used for the dense output(Hairer et al., 2015).

3. **Backward Difference Formula(BDF)**:Implicit multi-step variable-order (1 to 5) method based on a backward differentiation formula for the derivative approximation. The implementation follows the one described in. A quasi-constant step scheme is used and accuracy is enhanced using the NDF modification. Can be applied in the complex domain.(Curtiss & Hirschfelder, 1952).

2 Methodology

2.1 Initial and Boundary Conditions

We have Eq(4) as:

$$\frac{\partial p(x,t)}{\partial t} = \alpha_0 e^{\gamma p}\gamma(\frac{\partial p}{\partial x})^2 + \alpha_0 e^{\gamma p}\frac{\partial^2 p}{\partial x^2}$$

The initial conditions are:

$$p(x, t = 0) = 100,\ 0 < x < L$$

And boundary conditions are:

$$p(x = 0, t) = 100,\ t > 0$$
$$p(x = L, t) = 1,\ t > 0$$

Where, L is length of the domain.

2.2 Transformations and Simplifications

Eq(4) in itself might prove as a complicated and costly problem to solve numerical, so we further simplify it using the Cole-Hopf transformation(Hopf, 1950 and Cole, 1951), assuming there exists some η, such that :

$$p = \frac{1}{\gamma}\ln(\gamma\eta) \implies \eta = \frac{1}{\gamma}\exp(\gamma p) \tag{5}$$

Plugging this transformation into Eq(4) simplifies it into the following form:

$$\frac{\partial \eta}{\partial t} = \alpha_o \gamma \eta \frac{\partial^2 \eta}{\partial x^2} \tag{6}$$

So, the initial and boundary conditions gets transformed into:

$$\eta(x, t = 0) = \frac{1}{\gamma}\exp(100\gamma),\ 0 < x < L$$

$$\eta(x = 0, t) = \frac{1}{\gamma}\exp(100\gamma),\ t > 0$$

$$\eta(x = L, t) = \frac{1}{\gamma}\exp(1\gamma),\ t > 0$$

We then use a Method of Lines(MOL) based discretization keeping the time varying term on the LHS intact and discretizing the space varying term on RHS, using a second order central difference scheme:

$$(\frac{\partial \eta}{\partial t})_i = \alpha_o \gamma \eta(i) \frac{\eta(i+1) - 2\eta(i) + \eta(i-1)}{\Delta x^2} \tag{7}$$

2.3 Parameter space

The base parameter space for both FORTRAN and Python implementations which are used to solve the model are the same. They are defined below:

Parameter	**Value**
Initial diffusivity, α_o	0.001
Pressure sensitivity, γ	0.05
Length of domain, L	100
Time-stepping, dt	0.1
Grid spacing, dx	0.01

Table 1: Base Parameters

The FORTRAN implementations are solved for 10000 timesteps and 10000 spatial nodes. The Python implementations are found to be inherently slower, so they are scaled down temporally and spatially assuming geometrical and kinematic similarity(Hubbert, 1937) and their computation time is then scaled up using curve-fitting techniques.

3 Results

3.1 Numerical Solution

In this section the numerical solution of the said non-linear PDE, solved by the solvers used has been discussed. The spatial distribution of the pore-pressure profile is substantially different from the linear case which has the complementary error-function solution. The spatial distribution of pore pressure at different time steps have been shown below:

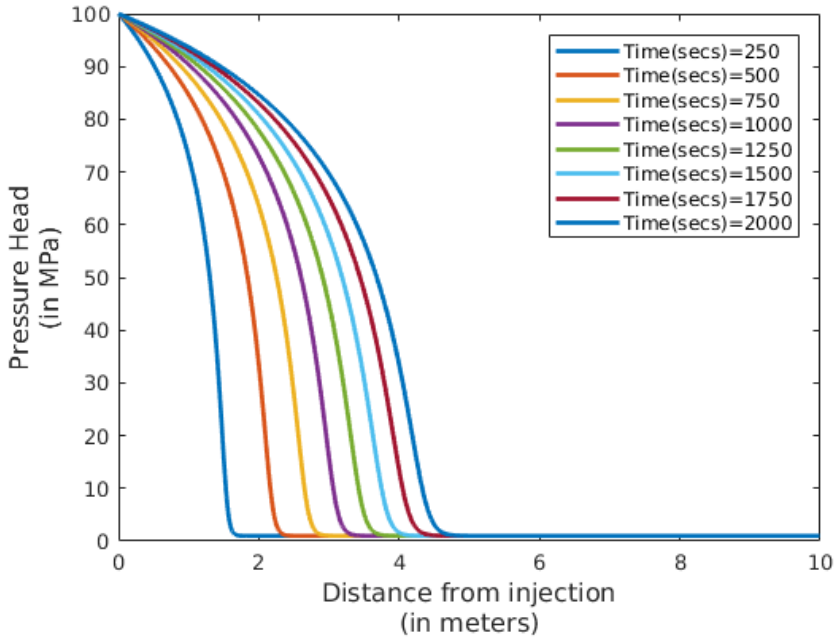

Figure 2: Spatial distribution of pore pressure at different time steps for the base parameters: Numerical solution.

3.2 Scaling and Extrapolation

Since the Python 3.7 implementations are substantially slower that the FORTRAN implementation, we scale them down assuming geometric and kinematic similarity, then extrapolate them using a general power model, where we fit the computational time data to the following model, $f(x) = ax^b + c$, to scale up the computational time to the standard spatial nodes and time steps for the faster FORTRAN implementations. Here $f(x)$, is the computational time, x, is the scaling factor and a, b&c are fitting coefficients. Results of the fitting and extrapolation are as follows:

Solver	a	b	c	x	f(x)
RK45	2.11E+05	2.043	0.8003	1	210600.8003
RK23	1.44E+05	1.923	-6.492	1	144093.508
Radau	8.43E+04	2.495	0.5442	1	84320.5442
BDF	1.65E+04	1.821	-3.202	1	16446.798

Table 2: Fitting Data

3.3 Computational Time

In this section, the computation time for the solvers used are being discussed for the same base parameter space(Table 1) and 10000 timesteps and 10000 spatial nodes, to which the Python 3.7 computation times are being scaled up, $f(x)$ from Table 2. Figure 3 below shows the computation time for the solvers.

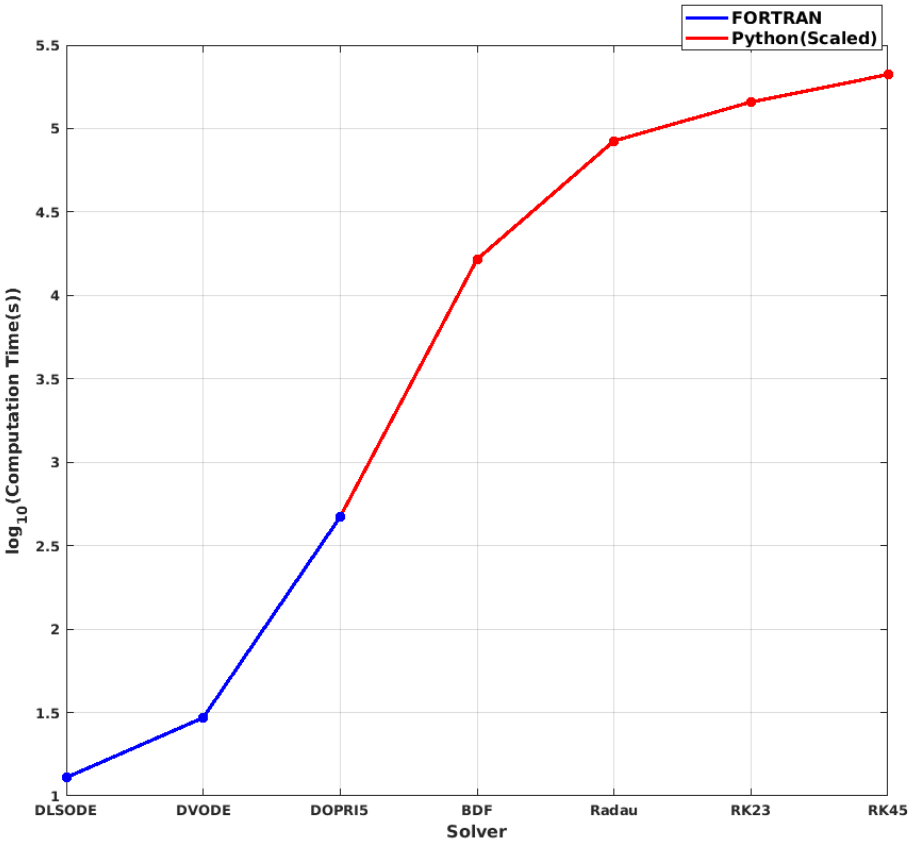

Figure 3: log_{10}(Computation Time) for all the solvers, blue curve are for the FORTRAN implementations and red curve is Python implementations. FORTRAN implementations are seen to be faster than the Python implementations.

3.4 CPU Usage

In this section, the CPU Usage by each solver is being discussed, the system in which the codes were run was a Intel i7 powered 4 node system with 4.45 GHz of maximum clock speed. The CPU processes were monitored using an UNIX shell add-on called `-mpstat` which dumps statistics of all CPU process as function of time in a specified dumpfile. The results of the CPU processes during runtime of each solver has been shown below.

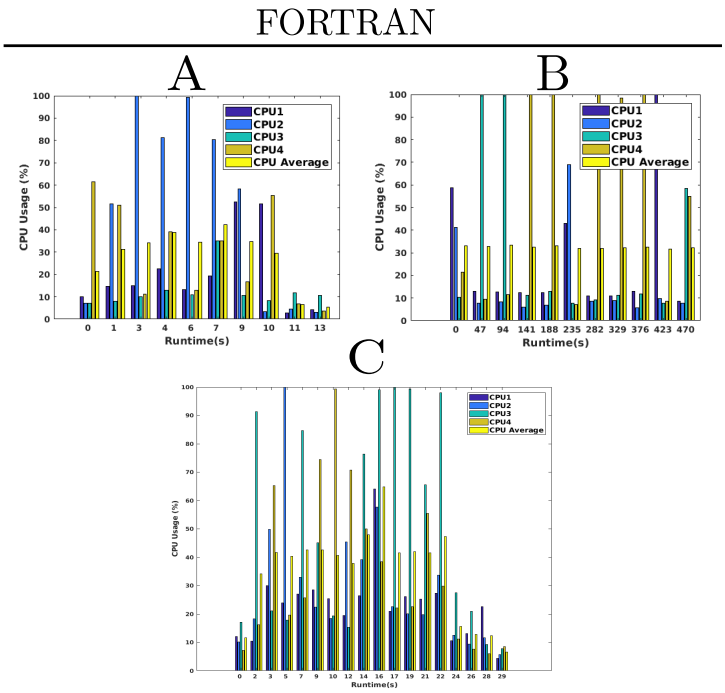

Figure 4: CPU Usage for the solvers in the FORTRAN implementations: A) DSLODE, B) DOPRI5, C) DVODE.

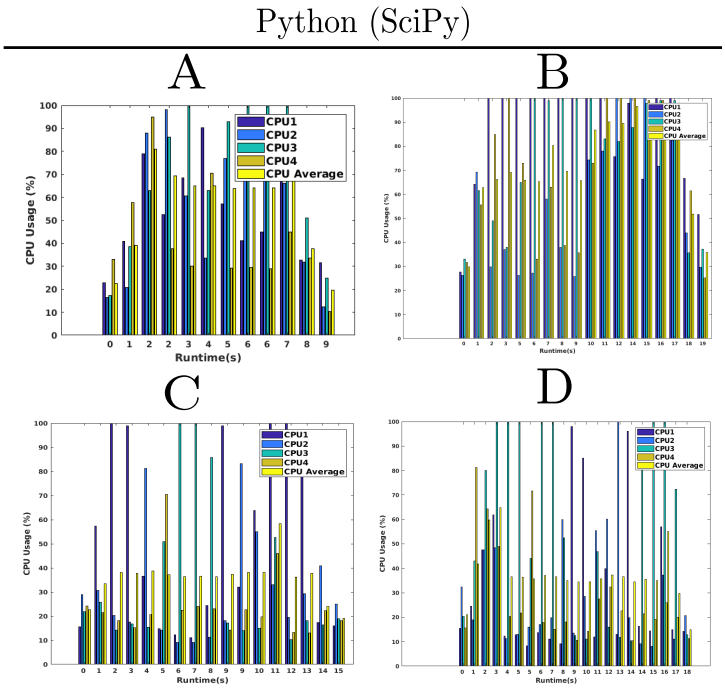

Figure 5: CPU Usage for the solvers in the Python implementations: A) BDF, B) RADAU, C) RK23, D) RK45.

4 Conclusions and Discussions

Results show a clear precedence of FORTRAN as a language for solving problems requiring high performance and those which are computationally costly, this is mainly due to the fact that it is compiler based and statically typed, whereas Python being a interpreter based language and being dynamically typed is slower in nature. This is the basic reason why FORTRAN despite being a low-level language is widely used in high performance computation, across scientific fields. Only advantage Python might have over FORTRAN is one has to write lot less code and can be used to solve toy problems, but is seriously ineffective in solving large numerical problems.

As far as solvers are concerned those which use Jacobian based methods (Family of Newton methods) are faster as seen in Figure 3, DLSODE & DVODE solvers which use Jacobian are atleast an order of magnitude faster than the DOPRI5 solver, same is true for BDF & Radau solvers in the Python implementations which are faster than the Range-Kutta solvers, this can be attributed to the fact that the use of a Newton method(or allied) increases the rate of convergence of the solution, closer the guess is to the original solution.

Monitoring of the CPU usage show that none of the common ODE solvers are paralellized, work load is switched from one node to other during the computation time. Though OpenMP based paralellization can be done, but for lower end systems with lower memories and less number of cores, this generally leads to overclocking.

References

Brown, P. N., Byrne, G. D., & Hindmarsh, A. C. (1989). Vode: A variable-coefficient ode solver. *SIAM journal on scientific and statistical computing*, *10*(5), 1038–1051.

Cole, J. D. (1951). On a quasi-linear parabolic equation occurring in aerodynamics. *Quarterly of applied mathematics*, *9*(3), 225–236.

Curtiss, C. F., & Hirschfelder, J. O. (1952). Integration of stiff equations. *Proceedings of the National Academy of Sciences of the United States of America*, *38*(3), 235.

De Paola, N., Faulkner, D., & Collettini, C. (2009). Brittle versus ductile deformation as the main control on the transport properties of low-porosity anhydrite rocks. *Journal of Geophysical Research: Solid Earth*, *114*(B6).

Dormand, J. R., & Prince, P. J. (1980). A family of embedded runge-kutta formulae. *Journal of computational and applied mathematics*, *6*(1), 19–26.

Hairer, E., Nørsett, S., & Wanner, G. (2000). *Solving ordinary differential equations i–nonstiff problems, 2nd printing of the 2nd edition.* Springer, New York etc.

Hopf, E. (1950). The partial differential equation ut+ uux= μxx. *Communications on Pure and Applied mathematics*, *3*(3), 201–230.

Hubbert, M. K. (1937). Theory of scale models as applied to the study of geologic structures. *Bulletin of the Geological Society of America*, *48*(10), 1459–1520.

Radhakrishnan, K., & Hindmarsh, A. C. (1993). Description and use of lsode, the livermore solver for ordinary differential equations.

Wanner, G., & Hairer, E. (1996). *Solving ordinary differential equations ii.* Springer Berlin Heidelberg.

Yilmaz, Ö., Nolen-Hoeksema, R. C., & Nur, A. (1994). Pore pressure profiles in fractured and compliant rocks 1. *Geophysical prospecting*, *42*(6), 693–714.