Numerical analysis, testing and adaptive timestep considerations for filtered implicit methods

by

Stephen Michael McGovern

Bachelor of Science, Pennsylvania State University, 2011

Submitted to the Graduate Faculty of

the Dietrich School of Arts and Sciences in partial fulfillment

of the requirements for the degree of

Master of Science

University of Pittsburgh

2022

UNIVERSITY OF PITTSBURGH

DIETRICH SCHOOL OF ARTS AND SCIENCES

This thesis was presented

by

Stephen Michael McGovern

It was defended on

December 6, 2022

and approved by

Professor William Layton, Department of Mathematics

Professor Michael Neilan, Department of Mathematics

Professor Catalin Trenchea, Department of Mathematics

Numerical analysis, testing and adaptive timestep considerations for filtered implicit methods

Stephen Michael McGovern, M.S.

University of Pittsburgh, 2022

Pre- and post-filters can be added to many classic numerical methods to generate higher order methods with strong stability properties. In this paper we test and analyze two filtered Implicit Euler methods. The two methods are a natural embedded pair, so we also explore variable timestep using their difference as an error estimator at each step.

Table of Contents

List of Tables

List of Figures

1.0 Introduction

There are many classic numerical methods for differential equations which are well-studied, effective and generally easy to implement. Methods such as those from the BDF family and Runge-Kutta family fall into this category. They are used often in applications and legacy codebases across different fields.

Naturally, needs arise in these applications for improved methods, but stringent testing requirements must still be met within a reasonable time frame. Therefore a balance of accuracy, stability and testing concerns must be kept in mind. In addition, ease of implementation and cognitive complexity are also factors to be considered.

To address these needs for improved methods, it has been shown that new, higher order methods can be generated by adding inexpensive pre-filtering and/or post-filtering steps to many of the classic methods [5] [6] [4] [8]. Typically, these pre- and post-filters are only a few lines of code each. This keeps cognitive complexity and implementation times low. Moreover, the new methods have favorable accuracy and stability properties despite the relative simplicity of the code modifications.

The objective of this paper is to numerically test, analyze and review the derivation of two novel, Implicit Euler-based, filtered methods from [4]. There they were primarily tested on the Navier-Stokes Equations. Here we will test them on a few stiff and oscillating problems.

The first method is a second order, pre-filtered variant of Implicit Euler. We will follow the original naming convention and call it IE-Pre-2. The next method has the same pre-filter and solve step as IE-Pre-2, but adds an additional post-filtering step. This new method is third order. We will refer to it as IE-Pre-Post-3.

Finally, these two methods are a natural embedded pair because the second order method is performed at each step of the third order method. Denote the value computed from IE-Pre-2 as y_{n+1}^{2nd} and the the value computed from IE-Pre-Post-3 as y_{n+1}^{3rd}. Then we can use the

following

$$EST = |y_{n+1}^{3rd} - y_{n+1}^{2nd}|$$

as an estimator of the error at each step. We will discuss how to adapt the timestep using this estimator and discrete curvature ideas. An adaptive pre-filter and post-filter are presented, along with an adaptive implementation using halving and doubling. Code will be available at `https://github.com/stevemcgov` for the test problems and the filtered methods presented here.

2.0 The Filtered Methods

The filtered methods we will analyze and test are based on the Implicit Euler method. Pre- and post-filters are added to the method to induce higher order methods with strong stability properties. We will compare them to BDF methods of like orders.

The first method, IE-Pre-2, is a second order pre-filtered method from [4]. From the same source, IE-Pre-Post-3 has the same pre-filter as IE-Pre-2 but then adds a post-filter after the Implicit Euler solve. We will cover their derivations, stability properties and accuracy. Both methods were primarily tested on the Navier-Stokes equations in [4]. We will test the methods on some stiff and oscillating problems of the form $y' = f(t, y)$ in the following sections.

We first state the original method setup from [4]. For the Navier-Stokes equations with a suppressed spatial discretization, Implicit Euler reads:

$$\frac{u^{n+1} - u^n}{\Delta t} + u^{n+1} \cdot \nabla u^{n+1} - \nu \Delta u^{n+1} + \nabla p^{n+1} = 0 \,\text{and}\, \nabla \cdot u^{n+1} = 0$$

With the pre- and post-filters [4]:

Step 1) Pre-filter $\quad \tilde{u}^n = u^n - \frac{1}{2}(u^n - 2u^{n-1} + u^{n-2})$

Step 2) IE Solve
$$\begin{cases} \frac{1}{\Delta t} u^{n+1} + u^{n+1} \cdot \nabla u^{n+1} - \nu \Delta u^{n+1} + \nabla p^{n+1} = \frac{1}{\Delta t} \tilde{u}^n \\ \nabla \cdot u^{n+1} = 0 \end{cases}$$

Step 3) Post-filter $\quad u_{3rd}^{n+1} = u^{n+1} - \frac{5}{11}(u^{n+1} - 3u^n + 3u^{n-1} - u^{n-2})$

We will test the underlying idea on problems of the form $y' = f(t, y)$. Rewriting the above method:

$$\text{Step 1) Pre-filter} \qquad \tilde{y}_n = y_n - \frac{1}{2}(y_n - 2y_{n-1} + y_{n-2})$$

$$\iff \tilde{y}_n = \frac{1}{2}y_n + y_{n-1} - \frac{1}{2}y_{n-2}\,,$$

$$\text{Step 2) IE Solve} \qquad \frac{y_{n+1} - \tilde{y}_n}{k} = f(t_{n+1}y_{n+1})$$

$$\text{Step 3) Post-filter} \qquad y_{n+1}^{3rd} = y_{n+1} - \frac{5}{11}(y_{n+1} - 3y_n + 3y_{n-1} - y_{n-2})$$

$$\iff y_{n+1}^{3rd} = \frac{6}{11}y_{n+1} + \frac{15}{11}y_n - \frac{15}{11}y_{n-1} + \frac{5}{11}y_{n-2}$$

Note that we altered the notation slightly, swapping u for y and Δt for k. Here step 2 is using the standard Implicit Euler but with the pre-filtered value $\tilde{y}_n$:

$$\text{IE:} \quad y_{n+1} = \tilde{y}_n + kf(t_{n+1}, y_{n+1})$$

2.1 IE-Pre-2

According to [4], completing all three steps above yields the third order method IE-Pre-Post-3 (to be discussed in a subsequent section). Interestingly, completing just the first two steps above can be viewed as a standalone second order method. We use the original naming convention and refer to it as

$$\text{IE-Pre-2} \qquad : \text{Pre-filtered Implicit Euler, Second Order}$$

The method reads as follows:

$$\text{Step 1) Pre-filter} \qquad \tilde{y}_n = \frac{1}{2}y_n + y_{n-1} - \frac{1}{2}y_{n-2}\,,$$

$$\text{Step 2) IE Solve} \qquad \frac{y_{n+1} - \tilde{y}_n}{k} = f(t_{n+1}, y_{n+1})$$

It is second order accurate and L-stable.

2.1.1 Local Truncation Error for IE-Pre-2

We can show that IE-Pre-2 is second order with the usual Taylor expansion methodology. Starting with

$$\tilde{y}_n = \frac{1}{2}y_n + y_{n-1} - \frac{1}{2}y_{n-2}$$

$$y_{n+1} - \tilde{y}_n = kf(t_{n+1}, y_{n+1})$$

Add these equations and then set $f(t_{n+1}, y_{n+1})$ equal to $h(y')_{n-1}$. This yields:

$$y_{n+1} = \frac{1}{2}y_n + y_{n-1} - \frac{1}{2}y_{n-2} + k(y')_{n-1}$$

$$\iff 0 = y_{n+1} - \frac{1}{2}y_n - y_{n-1} + \frac{1}{2}y_{n-2} - k(y')_{n-1}$$

The Taylor expansions are:

$$\begin{aligned}
y_{n+1} &= y_{n+1} \\
-\frac{1}{2}y_n &= -\frac{1}{2}y_{n+1} && + \frac{1}{2}ky' && - \frac{1}{2}\cdot\frac{k^2}{2}y'' && + \frac{1}{2}\cdot\frac{k^3}{6}y''' && + \mathcal{O}(k^4) \\
-y_{n-1} &= -y_{n+1} && + 2ky' && - \frac{4k^2}{2}y'' && + \frac{8k^3}{6}y''' && + \mathcal{O}(k^4) \\
+\frac{1}{2}y_{n-2} &= \frac{1}{2}y_{n+1} && - \frac{1}{2}\cdot 3ky' && + \frac{1}{2}\cdot\frac{9k^2}{2}y'' && - \frac{1}{2}\cdot\frac{27k^3}{6}y''' && + \mathcal{O}(k^4) \\
-k(y')_{n-1} &= && - ky' \\
\hline
LTE &= \mathcal{O}(k^3)
\end{aligned}$$

We can see after adding the rows above that the first three columns on the right-hand side cancel out. The first nonzero term, and consequently the LTE, is $\mathcal{O}(k^3)$. After the accumulation of the LTE at each step, the global error is then $\mathcal{O}(k^2)$. Thus we have shown IE-Pre-2 is second order.

2.1.2 Derivation of IE-Pre-2

Here we cover how to choose the pre-filter and derive the method. It's noted in [4] that a 2-point pre-filter for Implicit Euler does not affect the order of the scheme, however, it can help to improve some of the error constants. A 3-point pre-filter on the other hand *can* affect the order of the scheme in a favorable manner. The Robert-Asselin filter [3][12] was used in the method development as a pre-filter. It is:

$$\tilde{y}_n = y_n - \frac{\alpha}{2}(y_n - 2y_{n-1} + y_{n-2})$$

$$\iff \tilde{y}_n = (1 - \frac{\alpha}{2})y_n + \alpha y_{n-1} - \frac{\alpha}{2}y_{n-2}$$

From here we can show how to derive IE-Pre-2 with a constant time-step. The goal is to determine which value for α will result in an $\mathcal{O}(k^3)$ local truncation error (LTE). A third order LTE will accumulate into a second order global error, i.e. provide us with a second order method. Start with

$$\tilde{y}_n = (1 - \frac{\alpha}{2})y_n + \alpha y_{n-1} - \frac{\alpha}{2}y_{n-2}$$

$$y_{n+1} - \tilde{y}_n = kf(t_{n+1}, y_{n+1})$$

and add. Then rewrite $f(t_{n+1}, y_{n+1})$ as $k(y')_{n+1}$ and we have:

$$y_{n+1} = (1 - \frac{\alpha}{2})y_n + \alpha y_{n-1} - \frac{\alpha}{2}y_{n-2} + k(y')_{n+1}$$

$$\iff 0 = y_{n+1} - (1 - \frac{\alpha}{2})y_n - \alpha y_{n-1} + \frac{\alpha}{2}y_{n-2} - k(y')_{n+1}$$

Taylor expanding each term yields:

$$
\begin{aligned}
y_{n+1} &= y_{n+1} \\
-(1-\frac{\alpha}{2})y_n &= -(1-\frac{\alpha}{2})y_{n+1} &&+(1-\frac{\alpha}{2})ky' &&-(1-\frac{\alpha}{2})\cdot\frac{k^2}{2}y'' &&+(1-\frac{\alpha}{2})\cdot\frac{k^3}{6}y''' + \mathcal{O}(k^4) \\
-\alpha y_{n-1} &= -\alpha y_{n+1} &&+\alpha\cdot 2ky' &&-\alpha\frac{4k^2}{2}y'' &&+\alpha\frac{8k^3}{6}y''' &&+\mathcal{O}(k^4) \\
+\frac{\alpha}{2}y_{n-2} &= \frac{\alpha}{2}y_{n+1} &&-\frac{\alpha}{2}\cdot 3ky' &&+\frac{\alpha}{2}\cdot\frac{9k^2}{2}y'' &&-\frac{\alpha}{2}\cdot\frac{27k^3}{6}y''' &&+\mathcal{O}(k^4) \\
-k(y')_{n+1} &= &&-ky'
\end{aligned}
$$

In order to get a consistent, second order method, we require the first three columns above to sum to zero.

In the first column above, we have

$$1-\left(1-\frac{\alpha}{2}\right)-\alpha+\frac{\alpha}{2}=0$$

which is zero regardless of the choice of α.

Adding the coefficients in the second column:

$$\left(1-\frac{\alpha}{2}\right)+2\alpha-\frac{3\alpha}{2}-1=0$$

which is trivially zero again.

So far, we have that the first two columns zero out. This means we have at least a consistent method. Next, we need the coefficients in the third column (corresponding to the $\mathcal{O}(k^2)$ terms) to also sum to zero. This will give second order accuracy. So we set the sum equal to zero and solve for α:

$$
\begin{aligned}
0 &= -(1-\frac{\alpha}{2})-4\alpha+\frac{9\alpha}{2} = -1-4\alpha+5\alpha \\
\iff \alpha &= 1
\end{aligned}
$$

Thus when $\alpha = 1$, the LTE is $\mathcal{O}(k^3)$. After the LTE accumulates, the global error is $\mathcal{O}(k^2)$. Then the pre-filtered method is second order consistent by construction.

2.1.3 Numerical Test for Order: IE-Pre-2

To establish a numerical baseline for the order of the method, we test IE-Pre-2 on the model problem. The successive error ratios are computed while doubling the number of steps (i.e. halving the timestep). Then the base two log of the error ratios gives an estimate of the order of the method. Note: The final column of the following tables is

$$\log_2 r_{n+1} \quad , \quad \text{where } r_{n+1} = \frac{e_n}{e_{n+1}}$$

The canonical model/test problem is

$$y' = \lambda y$$
$$y(0) = 1$$

For simplicity, set $\lambda = 1$ and the solution to the initial value problem is:

$$y = e^t$$

Solved on the interval $0 \leq t \leq 1$:

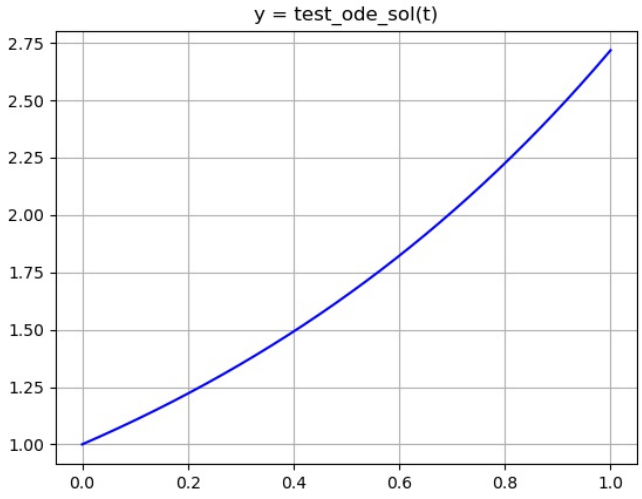

Figure 1: Test ODE Solution

The table of results for IE-Pre-2 show order 2. A table for BDF2 is given for comparison. These tests are consistent with the LTE analysis and derivation in the previous sections.

Table 1: IE-Pre-2 Numerical Test for Order

Number of Steps	Error	Ratio	Order
40	0.003478759798465	3.92804474303047	1.97381136299796
80	0.000885621225328	3.96192698470728	1.98620229270308
160	0.000223532949685	3.98041586855153	1.99291916950102
320	5.6158189764E-05	3.99006784696287	1.99641327818814
640	1.407449495E-05	3.99499857619393	1.99819498898096
1280	3.523028778E-06	3.99749065958741	1.99909466275310
2560	8.81310071E-07	3.99873773417948	1.99954466199163
5120	2.20397068E-07	3.99934523212429	1.99976382307753
10240	5.5108288E-08	4.00002037194924	2.00000734760883

Table 2: BDF2 Numerical Test for Order

Number of Steps	Error	Error Ratio	Order
40	0.000540909807066	3.91022121777502	1.96725022921012
80	0.000138332277623	3.95414112869318	1.98336436374093
160	3.4984152847E-05	3.97681954530808	1.99161509898100
320	8.797017931E-06	3.98834589130650	1.99579053361606
640	2.205680794E-06	3.99415683290372	1.99789098215851
1280	5.52226887E-07	3.99707404607948	1.99894429903361
2560	1.38157783E-07	3.99853906596000	1.99947298317840
5120	3.4552065E-08	3.99926485475793	1.99973482803300
10240	8.639604E-09	3.99974074728926	1.99990649131964

2.1.4 A-stable and Stability Region

We apply the Boundary Locus Method to find the stability region and determine A-stability for IE-Pre-2. According to the definitions presented in [7] [2], it is required for the stability region of the method to contain the entire left half-plane of the complex plane to be A-stable. We follow the method presented in [7] for finding the *boundary* of the stability region, as this tends to be easier than explicitly finding the region itself. We then use test points on either side of the boundary curve to determine the stability region.

To apply the Boundary Locus Method to IE-Pre-2, first construct the equivalent multistep method by substituting away $\tilde{y}_n$ as follows:

$$\begin{aligned} y_{n+1} - \tilde{y}_n &= kf(t^{n+1}, y_{n+1}) \iff \\ y_{n+1} - \left(\frac{1}{2}y_n + y_{n-1} - \frac{1}{2}y_{n-2}\right) &= kf(t_{n+1}, y_{n+1}) \iff \\ y_{n+1} - \frac{1}{2}y_n - y_{n-1} + \frac{1}{2}y_{n-2} &= kf(t_{n+1}, y_{n+1}) \iff \end{aligned}$$

Then apply this multistep method to the model problem $y' = \lambda y$. We have

$$y_{n+1} - \frac{1}{2}y_n - y_{n-1} + \frac{1}{2}y_{n-2} = k\lambda y_{n+1}$$

Denote $k\lambda$ as $\hat{k}$, analogous to $h\lambda$ and $\hat{h}$ in [7]. This is done because h/k and λ tend to appear together. Then the stability polynomial is:

$$r^3 - \frac{1}{2}r^2 - r + \frac{1}{2} = \hat{k}r^3 \iff$$

$$(1 - \hat{k})r^3 - \frac{1}{2}r^2 - r + \frac{1}{2} = 0$$

We will use the stability polynomial to verify which subregions satisfy the stability conditions. However, to plot the boundary curve, solve for $\hat{k}$ and use $r = e^{is}$. We have:

$$\frac{r^3 - \frac{1}{2}r^2 - r + \frac{1}{2}}{r^3} = \hat{k}$$

$$1 - \frac{1}{2}e^{-is} - e^{-2is} + \frac{1}{2}e^{-3is} = \hat{k}(s) \qquad 0 \le s \le 2\pi$$

Then using Euler's: $e^{ix} = \cos x + i \sin x$ and gathering terms, we have:

$$1 - \frac{1}{2}\cos(-s) - \cos(-2s) + \frac{1}{2}\cos(-3s) + i\left(-\frac{1}{2}\sin(-s) - \sin(-2s) + \frac{1}{2}\sin(-3s)\right) = \hat{k}(s)$$

$$1 - \frac{1}{2}\cos(s) - \cos(2s) + \frac{1}{2}\cos(3s) + i\left(\frac{1}{2}\sin(s) + \sin(2s) - \frac{1}{2}\sin(3s)\right) = \hat{k}(s)$$

We can plot the above parameterized curve in the complex plane.

Additionally, we can take advantage of Python's numpy and matplotlib.pyplot libraries to simplify the process. The code *boundary_locus.py* presented in the appendices only requires the original function for $\hat{k}$ in terms of r to generate the boundary curve below.

Function input:

$$\frac{r^3 - \frac{1}{2}r^2 - r + \frac{1}{2}}{r^3} = \hat{k}$$

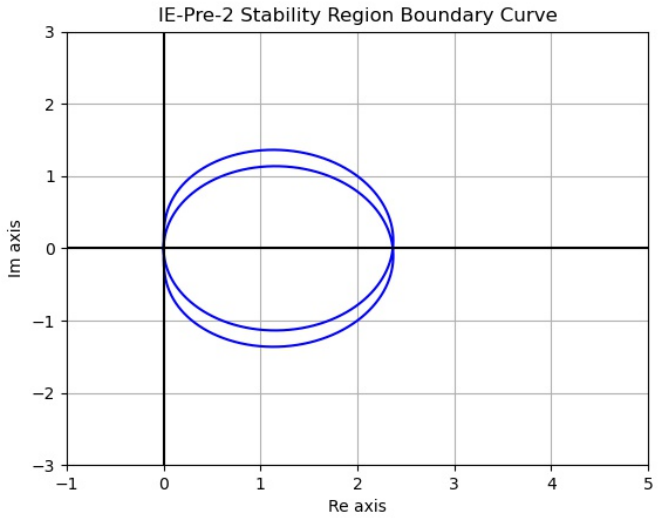

Figure 2: IE-Pre-2 Stability Region Boundary Curve

Now we test each subregion to determine the true stability region. As noted in [7] p. 87, the roots of the stability polynomial must satisfy the strict root condition $|r| < 1$ for each $\hat{k} \in \mathbb{C}$. To illustrate this, the values $\hat{k} = 1 + 1.25i\,,\ 1 - 1.25i\,,\ 1\,,\ 3$ are chosen from each region:

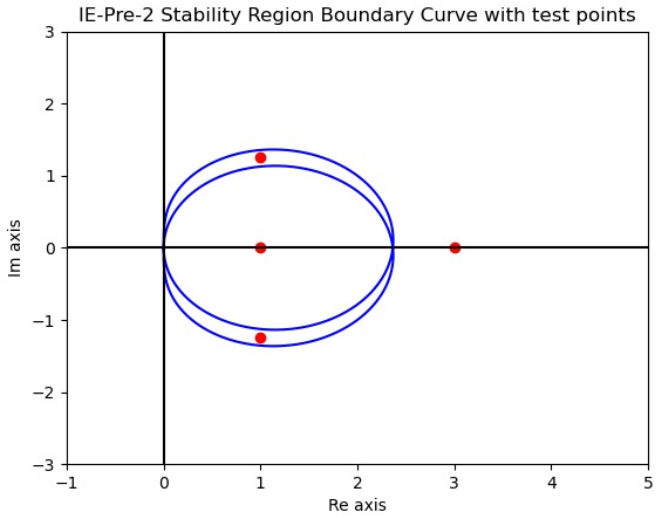

Figure 3: IE-Pre-2 Stability Region Boundary Curve with test points

The script *find_stability_roots.py* (presented in the appendices) uses the numpy library to programmatically find the roots of the following polynomials:

Table 3: Ie-Pre-2 Stability Polynomial with different test values

$\hat{k}$	Solve
$1 + 1.25i$	$(1 - (1 + 1.25i))r^3 - \frac{1}{2}r^2 - r + \frac{1}{2} = 0$
$1 - 1.25i$	$(1 - (1 - 1.25i))r^3 - \frac{1}{2}r^2 - r + \frac{1}{2} = 0$
$1 + 0i$	$(1 - (1))r^3 - \frac{1}{2}r^2 - r + \frac{1}{2} = 0$
$3 + 0i$	$(1 - (3))r^3 - \frac{1}{2}r^2 - r + \frac{1}{2} = 0$

The roots are subsequently tested against the strict root condition. After the roots are found, we see that the each of the inner subregions fails the strict root condition. Thus the stability region is the outermost region containing the test point $\hat{k} = 3$.

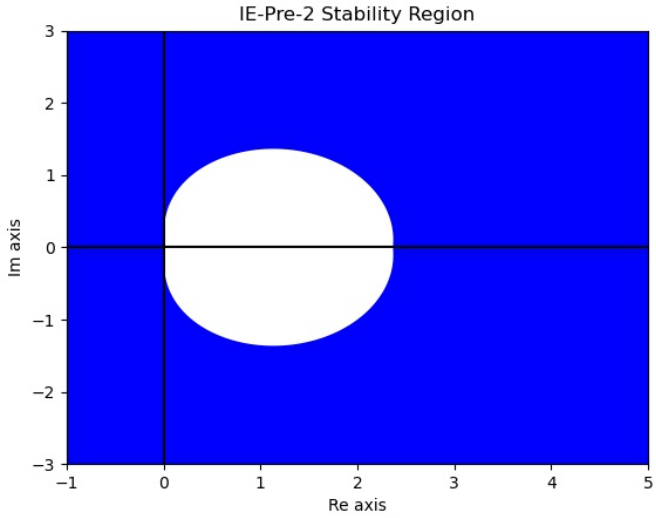

Figure 4: IE-Pre-2 Stability Region

Additionally, we compare to the BDF2 stability region. Here we can see that that IE-Pre-2 has a larger stability region than BDF2. A larger stability region suggests the method could be more dissipative [8].

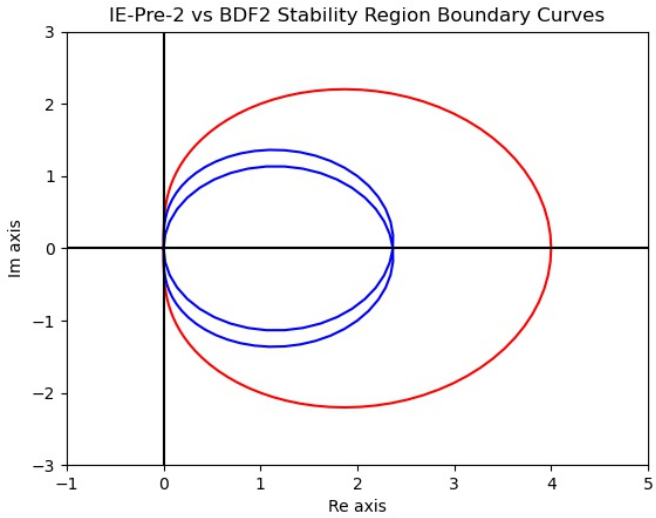

Figure 5: IE-Pre-2 vs BDF2 Stability Region Boundary Curves

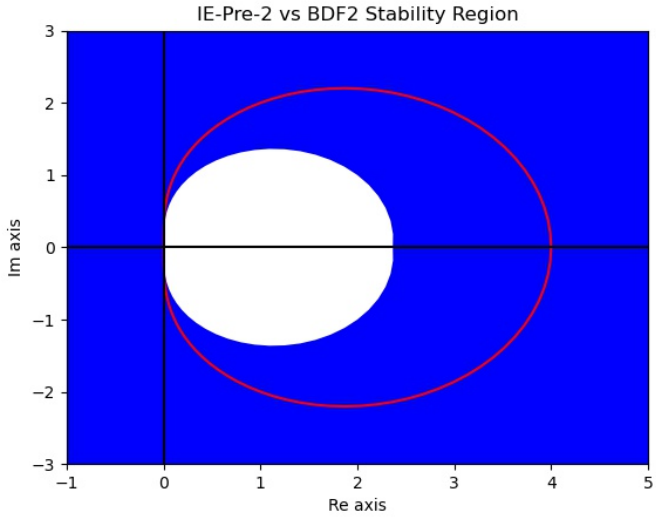

Figure 6: IE-Pre-2 vs BDF2 Stability Region

2.1.5 L-Stable

IE-Pre-2 is not only A-stable but also L-stable. This is a stronger version of A-stability. In [4], the method is shown to be L-stable by analyzing the eigenvalues of the incremental operator which advance solutions to the next time level. Here, we apply the method to $y' = \lambda y$, similar to $A-$stability. Then we show that y_n goes to zero as λ goes to $-\infty$. Compared to $A-$stability:

$$A-\text{stable}: \qquad y(t) \longrightarrow 0 \text{ as } t \longrightarrow \infty \text{ for fixed } \lambda, \text{ with Re } \lambda < 0$$

$$L-\text{stable}: \qquad y(t) \longrightarrow 0 \text{ as } \lambda \longrightarrow -\infty \text{ for fixed t and } \lambda \text{ is real and negative}$$

As mentioned above, first apply the method to $y' = \lambda y$, where λ is real and negative:

$$y_{n+1} - \frac{1}{2}y_n - y_{n-1} + \frac{1}{2}y_{n-2} = k\lambda y_{n+1}$$

with corresponding characteristic:

$$r^3 - \frac{1}{2}r^2 - r + \frac{1}{2} = k\lambda r^3$$

Suppose that $r_{1,2,3}$ are the roots of the above polynomial. Since the roots of a polynomial depend continuously on the coefficients [1] [9] [10], IE-Pre-2 is L-stable if the roots $r_i(\lambda) \longrightarrow 0$ as $\lambda \longrightarrow -\infty$ for t fixed.

$$r^3 - \frac{1}{2}r^2 - r + \frac{1}{2} = k\lambda r^3 \Longleftrightarrow$$
$$r^3 - \left(\frac{1/2}{1-k\lambda}\right)r^2 - \left(\frac{1}{1-k\lambda}\right)r + \left(\frac{1/2}{1-k\lambda}\right) = 0$$

Then as $1 - k\lambda \longrightarrow \infty$, we must have $r^3 = 0$ and thus

$$\lim_{\lambda \to -\infty} r_i(\lambda) = 0\,,\, 0\,,\, 0$$

Thus we have have that IE-Pre-2 is L-stable.

2.1.6 Code Implementation IE-Pre-2

```
def filtered_IE_pre_grant ( f_ode , x_range , y_init , num_steps ):
    import numpy as np
    from scipy.optimize import fsolve

    x = np.zeros ( num_steps + 1 )
    y = np.zeros ( ( num_steps + 1 , np.size ( y_init ) ) )
    h = ( x_range[1] - x_range[0] ) / num_steps
    x[0] = x_range[0]
    y[0,:] = y_init

    for k in range ( 1 , num_steps + 1 ):
        x[k] = x[k-1] + h
        if ( k < 3 ):
            y[k,:] = fsolve (bef, y[k , :] , args = ( f_ode, x[k-1], y[k-1,:], x[k]))
        else:
            temp_ykm1 = IE_pre_filter(y[k-1,:], y[k-2,:], y[k-3,:])
            y[k,:] = fsolve ( bef, y[k , :] , args = (f_ode, x[k-1], temp_ykm1, x[k]))

    return x , y

def IE_pre_filter( y_n , y_nm1 , y_nm2 ):
    # this the pre filter
    y_ntilde = (1.0/2.0) * y_n + y_nm1 - (1.0/2.0) * y_nm2
    return y_ntilde
```

Listing 2.1: Python IE-Pre-2

2.2 IE-Pre-Post-3

The next method we will analyze and test is the complete, pre- and post-filtered method. We refer to it as IE-Pre-Post-3:

IE-Pre-Post-3 : Pre- and Post-filtered Implicit Euler, Third Order

The method reads as follows:

Step 1) Pre-filter $$\tilde{y}_n = \frac{1}{2}y_n + y_{n-1} - \frac{1}{2}y_{n-2}\,,$$

Step 2) IE Solve $$\frac{y_{n+1} - \tilde{y}_n}{k} = f(t_{n+1}, y_{n+1})$$

Step 3) Post-filter $$y_{n+1}^{3rd} = \frac{6}{11}y_{n+1} + \frac{15}{11}y_n - \frac{15}{11}y_{n-1} + \frac{5}{11}y_{n-2}$$

As mentioned previously, this method is third order. The local truncation error is analyzed in the next section. According to Section 4.1.3 of [4], we lose L-stability and A-stability after the post-filter. However, it is still A(α) stable with $\alpha \approx 71^\circ$.

2.2.1 Local Truncation Error for IE-Pre-Post-3

To analyze the local truncation error, we first rewrite the method above as a multistep method. Starting at Step 3, solve for the pre-filtered y_{n+1}. Denote it by y^* and denote y_{n+1}^{3rd} now by y_{n+1}:

$$\begin{aligned} y_{n+1} &= \frac{6}{11}y^* + \frac{15}{11}y_n - \frac{15}{11}y_{n-1} + \frac{5}{11}y_{n-2} \iff \\ y^* &= \frac{11}{6}y_{n+1} - \frac{15}{6}y_n + \frac{15}{6}y_{n-1} - \frac{5}{6}y_{n-2} \end{aligned}$$

Now substitute this back into the solve step and we have the method:

$$\frac{11}{6}y_{n+1} - \frac{15}{6}y_n + \frac{15}{6}y_{n-1} - \frac{5}{6}y_{n-2} - \tilde{y}_n = kf\left(t_{n+1}, \frac{11}{6}y_{n+1} - \frac{15}{6}y_n + \frac{15}{6}y_{n-1} - \frac{5}{6}y_{n-2}\right)$$

Then use Step 1 to substitute out $\tilde{y}_n$ and combine terms:

$$\begin{aligned} \frac{11}{6}y_{n+1} - \frac{15}{6}y_n + \frac{15}{6}y_{n-1} - \frac{5}{6}y_{n-2} - \left(\frac{1}{2}y_n + y_{n-1} - \frac{1}{2}y_{n-2}\right) & \\ = kf\left(t_{n+1}, \frac{11}{6}y_{n+1} - \frac{15}{6}y_n + \frac{15}{6}y_{n-1} - \frac{5}{6}y_{n-2}\right) & \\ \frac{11}{6}y_{n+1} - \frac{18}{6}y_n + \frac{9}{6}y_{n-1} - \frac{2}{6}y_{n-2} = kf\left(t_{n+1}, \frac{11}{6}y_{n+1} - \frac{15}{6}y_n + \frac{15}{6}y_{n-1} - \frac{5}{6}y_{n-2}\right) & \end{aligned}$$

We apply the method to $y' = \lambda y = f(t, y)$ to analyze the local truncation error. Since $f(t, y) = \lambda y$, we have:

$$\begin{aligned} 11y_{n+1} - 18y_n + 9y_{n-1} - 2y_{n-2} &= k\lambda(11y_{n+1} - 15y_n + 15y_{n-1} - 5y_{n-2}) \\ &= 11k\lambda y_{n+1} - 15k\lambda y_n + 15k\lambda y_{n-1} - 5k\lambda y_{n-2} \end{aligned}$$

For the LTE, $\lambda y_{n+1} = (y')_{n+1}$, $\lambda y_n = (y')_n$, $\lambda y_{n-1} = (y')_{n-1}$, $\lambda y_{n-2} = (y')_{n-2}$:

$$11y_{n+1} - 18y_n + 9y_{n-1} - 2y_{n-2} = 11k(y')_{n+1} - 15k(y')_n + 15k(y')_{n-1} - 5k(y')_{n-2}$$

$$\iff 11y_{n+1} - 18y_n + 9y_{n-1} - 2y_{n-2} - 11k(y')_{n+1} + 15k(y')_n - 15k(y')_{n-1} + 5k(y')_{n-2} = 0$$

Following the same process for the LTE of IE-Pre-2, we Taylor expand around each of the terms:

$$
\begin{array}{rlllll}
11y_{n+1} = & 11y_{n+1} & & & & \\
-18y_n = & -18y_{n+1} & + 18ky' & - 18 \cdot \dfrac{k^2}{2}y'' & + 18 \cdot \dfrac{k^3}{6}y''' & - 18 \cdot \dfrac{k^4}{24}y^{(4)} \\
& & & & & + \mathcal{O}(k^5) \\
9y_{n-1} = & 9y_{n+1} & - 9 \cdot 2ky' & + 9 \cdot \dfrac{4k^2}{2}y'' & - 9 \cdot \dfrac{8k^3}{6}y''' & + 9 \cdot \dfrac{16k^4}{24}y^{(4)} \\
& & & & & + \mathcal{O}(k^5) \\
-2y_{n-2} = & -2y_{n+1} & + 2 \cdot 3ky' & - 2 \cdot \dfrac{9k^2}{2}y'' & + 2 \cdot \dfrac{27k^3}{6}y''' & - 2 \cdot \dfrac{81k^4}{24}y^{(4)} \\
& & & & & + \mathcal{O}(k^5) \\
-11k(y')_{n+1} = & & - 11ky' & & & \\
15k(y')_n = & & + 15ky' & - 15k \cdot ky'' & + 15h \cdot \dfrac{k^2}{2}y''' & - 15k \cdot \dfrac{k^3}{6}y^{(4)} \\
& & & & & + \mathcal{O}(k^5) \\
-15k(y')_{n-1} = & & - 15ky' & + 15k \cdot 2ky'' & - 15h \cdot \dfrac{4k^2}{2}y''' & + 15k \cdot \dfrac{k^3}{6}y^{(4)} \\
& & & & & + \mathcal{O}(k^5) \\
5k(y')_{n-2} = & & + 5ky' & - 5k \cdot 3ky'' & + 5h \cdot \dfrac{9k^2}{2}y''' & - 5k \cdot \dfrac{27k^3}{6}y^{(4)} \\
& & & & & + \mathcal{O}(k^5) \\
\hline
LTE = \mathcal{O}(k^4) & & & & &
\end{array}
$$

The first four columns above sum to zero and so the local truncation error is $\mathcal{O}(k^4)$. Thus the global error is then $\mathcal{O}(k^3)$ for IE-Pre-Post-3.

2.2.2 Derivation of IE-Pre-Post-3

Supposing the post-filter is unknown, we have our method as:

$$\text{Step 1) Pre-filter} \qquad \tilde{y}_n = y_n - \frac{1}{2}(y_n - 2y_{n-1} + y_{n-2}),$$

$$\text{Step 2) IE Solve} \qquad \frac{y_{n+1} - \tilde{y}_n}{k} = f(t_{n+1}, y_{n+1})$$

$$\text{Step 3) Post-filter} \qquad y_{n+1}^{3rd} = y_{n+1} - \beta(y_{n+1} - 3y_n + 3y_{n-1} - y_{n-2})$$

Here in the pre-filter step, $y_n - 2y_{n-1} + y_{n-2}$ is the discrete curvature at the last step, denoted κ_{n-1}. In the post-filter, the quantity $y_{n+1} - 3y_n + 3_{n-1} - y_{n-2}$ is the difference of the curvature from the last step and the current step:

$$\kappa_n - \kappa_{n-1} = y_{n+1} - 2y_n + y_{n-1} - (y_n - 2y_{n-1} + y_{n-2})$$

$$= y_{n+1} - 3y_n + 3y_{n-1} - y_{n-2}$$

In order to find the post-filter β, we proceed similar to the method used to find the LTE. Denote the pre-filtered y_{n+1} as y^*. Denote y_{n+1}^{3rd} as y_{n+1}. Solve for y^* as follows:

$$y_{n+1} = (1-\beta)y^* + 3\beta y_n - 3\beta y_{n-1} + \beta y_{n-2}$$

$$y^* = \frac{1}{1-\beta}y_{n+1} + \frac{-3\beta}{1-\beta}y_n + \frac{3\beta}{1-\beta}y_{n-1} + \frac{-\beta}{1-\beta}y_{n-2}$$

Now we substitute this and the pre-filter into the solve step:

$$\frac{1}{1-\beta}y_{n+1} + \frac{-3\beta}{1-\beta}y_n + \frac{3\beta}{1-\beta}y_{n-1} + \frac{-\beta}{1-\beta}y_{n-2} - \left(\frac{1}{2}y_n + y_{n-1} - \frac{1}{2}y_{n-2}\right)$$

$$= kf\left(t_{n+1}, \frac{1}{1-\beta}y_{n+1} + \frac{-3\beta}{1-\beta}y_n + \frac{3\beta}{1-\beta}y_{n-1} + \frac{-\beta}{1-\beta}y_{n-2}\right)$$

Combine terms and we have:

$$\frac{1}{1-\beta}y_{n+1} + \frac{-1-5\beta}{2(1-\beta)}y_n + \frac{-1+4\beta}{1-\beta}y_{n-1} + \frac{1-3\beta}{2(1-\beta)}y_{n-2}$$

$$= kf\left(t_{n+1}, \frac{1}{1-\beta}y_{n+1} + \frac{-3\beta}{1-\beta}y_n + \frac{3\beta}{1-\beta}y_{n-1} + \frac{-\beta}{1-\beta}y_{n-2}\right)$$

As with the LTE, we apply the method to $y' = \lambda y$:

$$\frac{1}{1-\beta}y_{n+1} + \frac{-1-5\beta}{2(1-\beta)}y_n + \frac{-1+4\beta}{1-\beta}y_{n-1} + \frac{1-3\beta}{2(1-\beta)}y_{n-2}$$

$$= k\lambda\left(\frac{1}{1-\beta}y_{n+1} + \frac{-3\beta}{1-\beta}y_n + \frac{3\beta}{1-\beta}y_{n-1} + \frac{-\beta}{1-\beta}y_{n-2}\right)$$

The righthand side is

$$= \frac{1}{1-\beta}k\lambda y_{n+1} + \frac{-3\beta}{1-\beta}k\lambda y_n + \frac{3\beta}{1-\beta}k\lambda y_{n-1} + \frac{-\beta}{1-\beta}k\lambda y_{n-2}$$

$$= \frac{1}{1-\beta}(y')_{n+1} + \frac{-3\beta}{1-\beta}(y')_n + \frac{3\beta}{1-\beta}(y')_{n-1} + \frac{-\beta}{1-\beta}(y')_{n-2}$$

Together with the lefthand side:

$$\frac{1}{1-\beta}y_{n+1} + \frac{-1-5\beta}{2(1-\beta)}y_n + \frac{-1+4\beta}{1-\beta}y_{n-1} + \frac{1-3\beta}{2(1-\beta)}y_{n-2}$$

$$+\frac{-1}{1-\beta}(y')_{n+1} + \frac{3\beta}{1-\beta}(y')_n + \frac{-3\beta}{1-\beta}(y')_{n-1} + \frac{\beta}{1-\beta}(y')_{n-2} = 0$$

Assuming $\beta \neq 1$, then

$$2y_{n+1} + (-1-5\beta)y_n + (-2+8\beta)y_{n-1} + (2-6\beta)y_{n-2}$$

$$-2(y')_{n+1} + 6\beta(y')_n - 6\beta(y')_{n-1} + 2\beta(y')_{n-2} = 0$$

Now we Taylor expand and add:

$$2y_{n+1} = 2y_{n+1}$$

$$(-1-5\beta)y_n = (-1-5\beta)y_{n+1} + (-1-5\beta)(-ky') + (-1-5\beta)\left(\frac{k^2}{2}y''\right) + (-1-5\beta)\left(-\frac{k^3}{6}y'''\right) + \mathcal{O}(k^4)$$

$$(-2+8\beta)y_{n-1} = (-2+8\beta)y_{n+1} + (-2+8\beta)(-2ky') + (-2+8\beta)\left(\frac{4k^2}{2}y''\right) + (-2+8\beta)\left(\frac{-8k^3}{6}y'''\right) + \mathcal{O}(k^4)$$

$$(1-3\beta)y_{n-2} = (1-3\beta)y_{n+1} + (1-3\beta)(-3ky') + (1-3\beta)\left(\frac{9k^2}{2}\right)y'' + (1-3\beta)\left(-\frac{27k^3}{6}y'''\right) + \mathcal{O}(k^4)$$

$$-2k(y')_{n+1} = -2k(y')_{n+1}$$

$$6\beta k(y')_n = 6\beta ky' + 6\beta k(-ky'') + 6\beta k\left(\frac{k^2}{2}y'''\right) + \mathcal{O}(k^4)$$

$$-6\beta k(y')_{n-1} = -6\beta ky' + (-6\beta k)(-2ky'') + (-6\beta k)\left(\frac{4k^2}{2}y'''\right) + \mathcal{O}(k^4)$$

$$2\beta k(y')_{n-2} = 2\beta ky' + 2\beta k(-3ky'') + 2\beta k\left(\frac{9k^2}{2}y'''\right) + \mathcal{O}(k^4)$$

For the coefficients corresponding to the y_{n+1} terms on the righthand side, we have

$$2 - 1 - 5\beta - 2 + 8\beta 1 - 3\beta = 0$$

Similarly, for the coefficients corresponding to the $\mathcal{O}(k)$ terms, we have

$$1 + 5\beta + 4 - 16\beta - 3 + 9\beta - 2 + 6 - 6 + 2\beta = 5 - 5 + 16\beta - 16\beta = 0$$

Since these sums are both zero, we have at least a first order consistent method.

Now for the $\mathcal{O}(k^2)$ terms, we have:

$$-1-5\beta-8+32\beta+9-27\beta-2\cdot 6\beta+2\cdot 12\beta-2\cdot 6\beta$$

$$=-9+9+32\beta-32\beta+24\beta-24\beta=0$$

This shows that the method is at least second order, regardless of the choice of β. Finally, we require the $\mathcal{O}(k^3)$ terms to vanish for third order. We have

$$1+5\beta+16-64\beta-27+81\beta+18\beta-18\beta\cdot 4+2\beta\cdot 9$$

$$=-10+22\beta$$

Set this to zero

$$0=-10+22\beta \iff \beta=\frac{5}{11}$$

Thus when $\beta=\frac{5}{11}$, the method has an $\mathcal{O}(k^4)$ LTE. This shows the method is third order.

2.2.3 Numerical Test for Order: IE-Pre-Post-3

Now we test IE-Pre-Post-3 on the model problem. The setup is the same as the numerical test for order for IE-Pre-2. The table of results for IE-Pre-Post-3 show order 3 and a table for BDF3 is given for comparison.

Table 4: IE-Pre-Post-3 Numerical Test for Order

Number of Steps	Error	Error Ratio	Order
40	4.1521257617E-05	7.59568706336048	2.92518046875255
80	5.466425522E-06	7.79817609627155	2.96313673364670
160	7.00987699E-07	7.89920286100073	2.98170707271935
320	8.8741575E-08	7.94973854111580	2.99090741239215
640	1.1162829E-08	7.97693060044225	2.99583372647718
1280	1.399389E-09	8.02117840515409	3.00381420106736
2560	1.74462E-10	8.37086360827598	3.06537647093630
5120	2.0842E-11	50.46344086021510	5.65716667501698
10240	4.13E-13	0.06764128300240	-3.88595216510007

Table 5: BDF3 Numerical Test for Order

Number of Steps	Error	Error Ratio	Order
40	9.553137484E-06	7.58275489160416	2.92272208934436
80	1.259850492E-06	7.79277593061643	2.96213733444276
160	1.61669026E-07	7.89673276042446	2.98125586780609
320	2.0472901E-08	7.94847999839999	2.99067899770101
640	2.5757E-09	7.97395660763608	2.99529575455539
1280	3.23014E-10	7.98221085785148	2.99678838954308
2560	4.0467E-11	7.97924693520140	2.99625259430264
5120	5.071E-12	7.70060687795010	2.94497214779557
10240	6.59E-13	12.05691056910570	3.59178837711539

2.2.4 Stability Region

Recall from the previous section that IE-Pre-Post-3 can be written as

$$\frac{11}{6}y_{n+1} - \frac{18}{6}y_n + \frac{9}{6}y_{n-1} - \frac{2}{6}y_{n-2} = kf\left(t_{n+1}, \frac{11}{6}y_{n+1} - \frac{15}{6}y_n + \frac{15}{6}y_{n-1} - \frac{5}{6}y_{n-2}\right)$$

We follow the same methodology we used to find the Stability Region for IE-Pre-2. First, apply this method to $y' = \lambda y$.

$$11y_{n+1} - 18y_n + 9y_{n-1} - 2y_{n-2} = 11k\lambda y_{n+1} - 15k\lambda y_n + 15k\lambda y_{n-1} - 5k\lambda y_{n-2}$$

Again, use $k\lambda = \hat{k}$ and then the stability polynomial is:

$$\begin{aligned} 11r^3 - 18r^2 + 9r - 2 = 11\hat{k}r^3 - 15\hat{k}r^2 + 15\hat{k}r - 5\hat{k} \iff \\ 0 = (11 - 11\hat{k})r^3 - (18 - 15\hat{k})r^2 + (9 - 15\hat{k})r - (2 - 5\hat{k}) \end{aligned}$$

This will be used for testing values in the subregions. Recall we solve for $\hat{k}$ to plot the boundary.

Employing *boundary_locus.py* to graph the boundary curve with

$$\hat{k}(r) = \frac{11r^3 - 18r^2 + 9r - 2}{11r^3 - 15r^2 + 15r - 5}$$

gives the following:

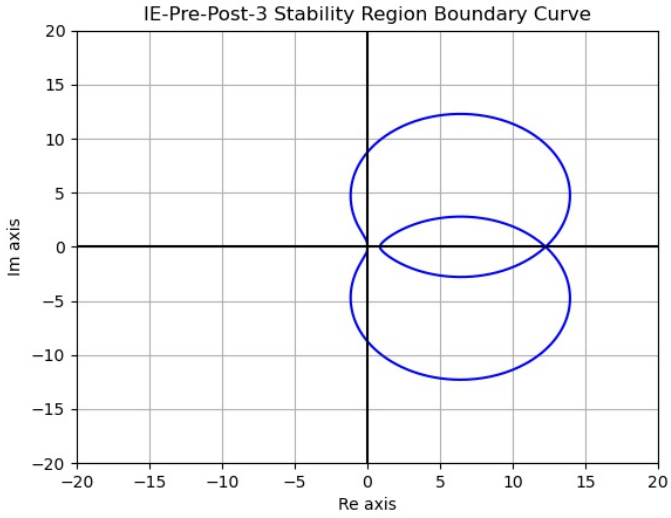

Figure 7: IE-Pre-Post-3 Stability Region Boundary Curve

Now we find roots of the stability polynomial with $\hat{k}$ test values to determine which subregions are absolutely stable. We choose $\hat{k} = 5\,,\ -5\,,\ 5 + 5i$ and use *find_roots.py* (see appendices). This shows that the only subregion which satisfies the strict root condition $|r| < 1$ for each $\hat{k}$ within it is the region corresponding to $\hat{k} = -5$.

Table 6: Ie-Pre-Post-3 Stability Polynomial with different test values

$\hat{k}$	Solve
$5 + 0i$	$(11 - 11(5))r^3 - (18 - 15(5))r^2 + (9 - 15(5))r - (2 - 5(5)) = 0$
$-5 + 0i$	$(11 - 11(-5))r^3 - (18 - 15(-5))r^2 + (9 - 15(-5))r - (2 - 5(-5)) = 0$
$5 + 5i$	$(11 - 11(5 + 5i))r^3 - (18 - 15(5 + 5i))r^2 + (9 - 15(5 + 5i))r - (2 - 5(5 + 5i)) = 0$

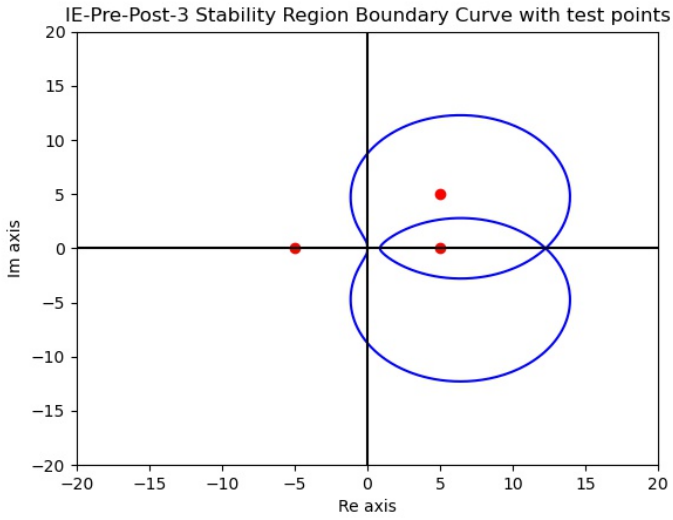

Figure 8: IE-Pre-Post-3 Stability Region

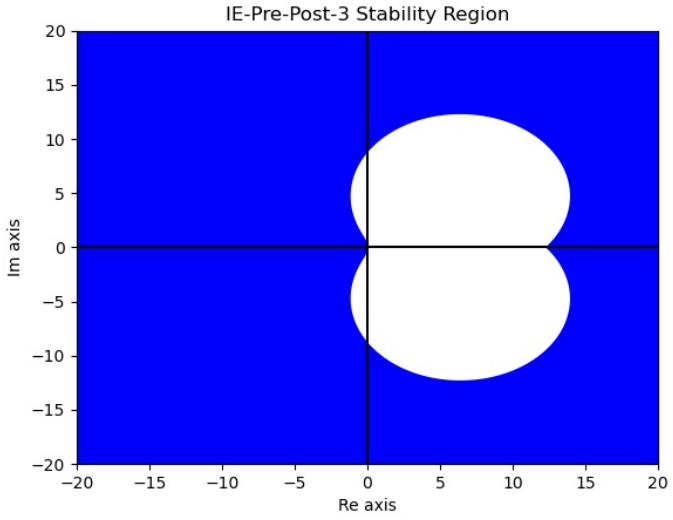

Figure 9: IE-Pre-Post-3 Stability Region

2.2.5 Code Implementation IE-Pre-Post-3

This version starts with the initial condition and then uses RK3 until there are enough values to begin filtering. RK3 was chosen because it is the same order. In the code repository, an alternative version is presented which uses the initial condition, then Implicit Euler, then BDF2 and then starts the filtering process. Both versions seem to perform similarly.

```
def filtered_IE_pre_post_3 ( f_ode , x_range , y_init , num_steps ):
    import numpy as np
    from scipy.optimize import fsolve
    from stiff_ode import stiff_solution
    from rk3 import rk3


    y_exact = stiff_solution ( x_range[1] )
    x = np.zeros ( num_steps + 1 )
    y = np.zeros ( ( num_steps + 1 , np.size ( y_init ) ) )
    h = ( x_range[1] - x_range[0] ) / num_steps
    x[0] = x_range[0]
    y[0,:] = y_init

    for k in range ( 1 , num_steps + 1 ):
        temp_ykpost = 0.0
        x[k] = x[k-1] + h
        dx = h
        if ( k < 3 ):
            #rk3
            # two intermediate points
            xa = x[k-1] + dx / 2
            ya = y[k-1] + dx / 2 * f_ode( x[k-1] , y[k-1] )

            xb = x[k-1] + dx
            yb = y[k-1] + dx * ( 2 * f_ode( xa , ya ) - f_ode( x[k-1] , y[k-1] ) )

            # estimate solution / true step
            x[k] = x[k-1] + dx
            y[k,:] = y[k-1,:] + \
            dx * (f_ode(x[k-1] , y[k-1]) + 4.0 * f_ode(xa , ya) + f_ode(xb , yb)) / 6.0

        else:
            temp_ykm1 = IE_pre_filter(y[k-1,:] , y[k-2,:] , y[k-3,:])
            y[k,:] = fsolve ( bef , y[k, :] , args = (f_ode, x[k-1], temp_ykm1 , x[k]))
            y[k,:] = IE_post_filter(y[k,:] , y[k-1,:] , y[k-2,:] , y[k-3,:])

    return x , y

def IE_pre_filter( y_n , y_nm1 , y_nm2 ):
    # this is the pre filter
    y_ntilde = (1.0/2.0) * y_n + y_nm1 - (1.0/2.0) * y_nm2
    return y_ntilde

def IE_post_filter( y_np1,  y_n , y_nm1 , y_nm2 ):
    # this is the post filter
    ynp1_third = (6.0/11.0) * y_np1 + (15.0/11.0) * y_n - (15.0/11.0) * y_nm1 + (5.0/11.0) *
     y_nm2
    return ynp1_third
```

Listing 2.2: Python IE-Pre-Post-3

3.0 The Test Problems for Constant Time Step

Here we test IE-Pre-2 and IE-Pre-Post-3 with a constant timestep on two stiff problems and the harmonic oscillator. We present tables of errors, error ratios and convergence rates for reference. True solution graphs are also given.

3.1 Stiff ODE 1

3.1.1 Problem Statement

As remarked on page 50 of [2], "the concept of stiffness is best understood in qualitative, rather than quantitative, terms." Loosely speaking, a differential equation is stiff if the step size required for stability is much smaller than the step size needed for desired accuracy. Additionally, if different time scales are required on different parts of the differential equation, or it is highly oscillatory, then it can be considered stiff. The first stiff problem we will consider is

$$\begin{cases} y' = \lambda(-y + \sin t) \\ \\ y(0) = 0 \end{cases}$$

The general solution is:

$$\begin{aligned} y(t) &= Ce^{-\lambda t} + \frac{\lambda^2 \sin t - \lambda \cos t}{1 + \lambda^2} \\ \\ &= Ce^{-\lambda t} + \frac{\lambda^2 \sin t}{1 + \lambda^2} - \frac{\lambda \cos t}{1 + \lambda^2} \end{aligned}$$

With the initial condition $y(0) = 0$, the constant C becomes

$$C = \frac{\lambda}{1 + \lambda^2}$$

Here, as λ becomes more negative, the problem becomes more stiff. First are the plots of the true solutions for $\lambda = 4.0, -1.0, -2.0$, followed by the numerical results , $0 \leq t \leq 5$.

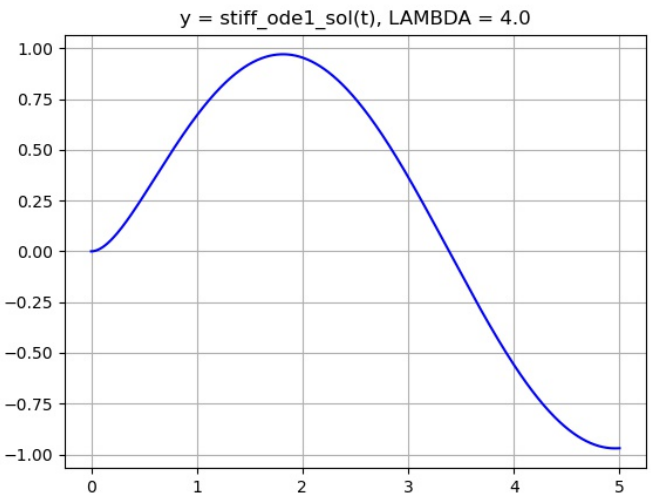

Figure 10: True Solution for Stiff ODE 1 when Lambda = 4.0

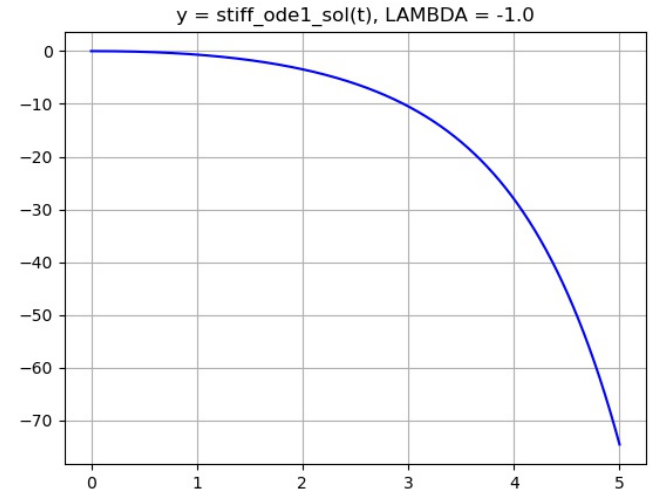

Figure 11: True Solution for Stiff ODE 1 when Lambda = -1.0

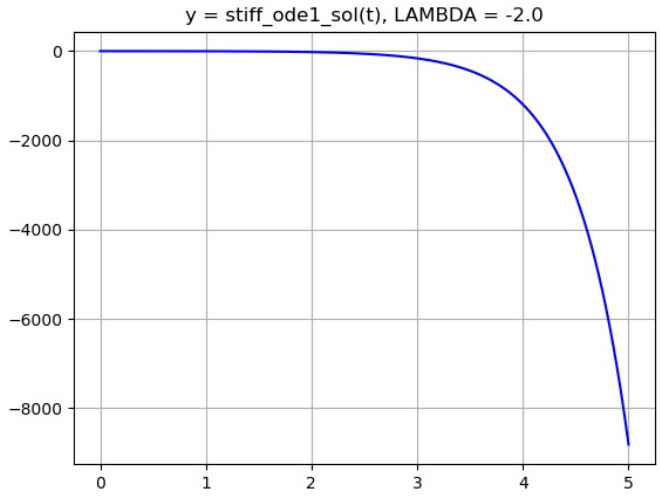

Figure 12: True Solution for Stiff ODE 1 when Lambda = -2.0

3.1.2 Numerical Results

Table 7: Stiff ODE 1, Lambda = 4.0 Error Table

Number of Steps	IE-Pre-2	BDF2	IE-Pre-Post-3	BDF3
40	1.0724E-03	3.5843E-04	6.8730E-04	1.0765E-04
80	2.1230E-04	7.5629E-05	6.9624E-05	1.3806E-05
160	4.5999E-05	1.7151E-05	7.6460E-06	1.7436E-06
320	1.0612E-05	4.0680E-06	8.8870E-07	2.1893E-07
640	2.5421E-06	9.8952E-07	1.0687E-07	2.7423E-08
1280	6.2163E-07	2.4395E-07	1.3094E-08	3.4313E-09
2560	1.5367E-07	6.0557E-08	1.6202E-09	4.2912E-10
5120	3.8201E-08	1.5086E-08	2.0148E-10	5.3651E-11
10240	9.5232E-09	3.7647E-09	2.5079E-11	6.7060E-12

Table 8: Stiff ODE 1, Lambda = -1.0 Error Table

Number of Steps	IE-Pre-2	BDF2	IE-Pre-Post-3	BDF3
40	7.0771E+00	1.7464E+00	5.8170E-01	1.1738E-01
80	1.8149E+00	4.5555E-01	8.2118E-02	1.6293E-02
160	4.6581E-01	1.1697E-01	1.0947E-02	2.1479E-03
320	1.1844E-01	2.9680E-02	1.4139E-03	2.7575E-04
640	2.9892E-02	7.4783E-03	1.7966E-04	3.4932E-05
1280	7.5103E-03	1.8771E-03	2.2644E-05	4.3957E-06
2560	1.8824E-03	4.7022E-04	2.8421E-06	5.5130E-07
5120	4.7121E-04	1.1768E-04	3.5597E-07	6.9028E-08
10240	1.1788E-04	2.9434E-05	4.4485E-08	8.6338E-09

Table 9: Stiff ODE 1, Lambda = -2.0 Error Table

Number of Steps	IE-Pre-2	BDF2	IE-Pre-Post-3	BDF3
40	5.9401E+03	1.7444E+03	1.0897E+03	2.4136E+02
80	1.3044E+03	4.3508E+02	1.5444E+02	3.5488E+01
160	3.2706E+02	1.1146E+02	2.1187E+01	4.8518E+00
320	8.3396E+01	2.8407E+01	2.7879E+00	6.3513E-01
640	2.1161E+01	7.1843E+00	3.5780E-01	8.1263E-02
1280	5.3369E+00	1.8074E+00	4.5325E-02	1.0277E-02
2560	1.3405E+00	4.5333E-01	5.7035E-03	1.2922E-03
5120	3.3596E-01	1.1352E-01	7.1531E-04	1.6199E-04
10240	8.4094E-02	2.8404E-02	8.9557E-05	2.0279E-05

3.2 Stiff ODE 2

3.2.1 Problem Statement

The second stiff problem to be tested is

$$\begin{cases} y' = \lambda(y - \sin t) + \cos t \\ \\ y(0) = 1 \end{cases}$$

where the solution is

$$y(t) = e^{\lambda t} + \sin t$$

Here, the problem becomes more stiff as λ increases. We can see from the solution graphs that the problem requires a smaller relative stepsize at first due to the intial steepness of the curve for small λ. After a sharp change in direction, not as small of a stepsize is required. The varying timescales and sharp turn both contribute to the problem's "stiffness." Accuracy for all methods starts to deteriorate at the $\lambda = 3$ test presented below.

3.2.2 Numerical Results

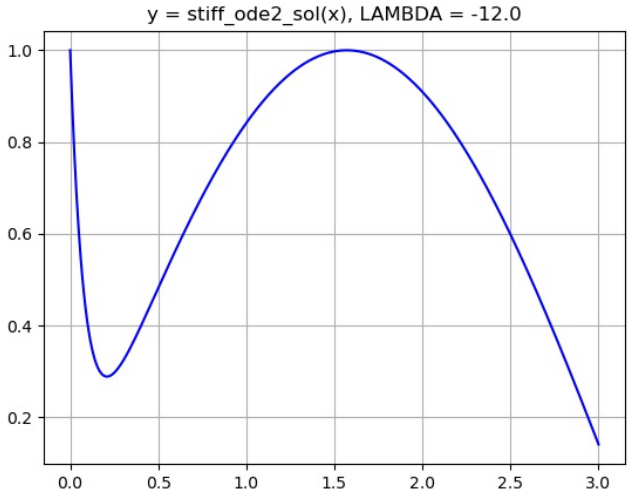

Figure 13: True Solution for Stiff ODE 2 when Lambda = -12.0

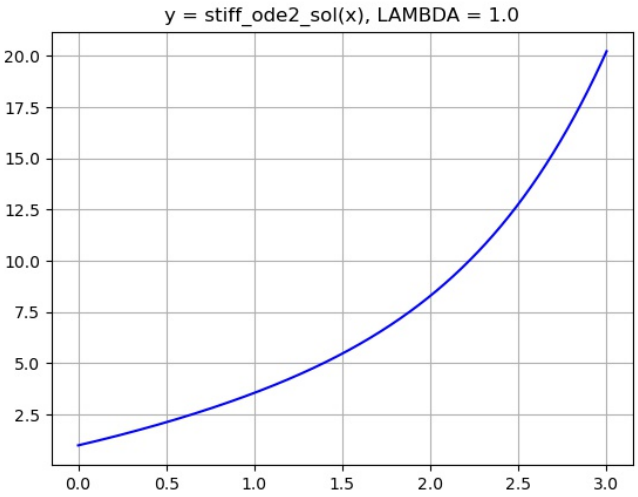

Figure 14: True Solution for Stiff ODE 2 when Lambda = 1.0

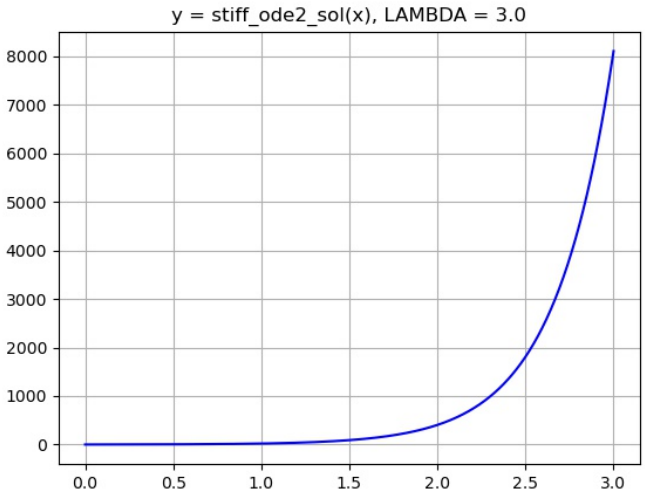

Figure 15: True Solution for Stiff ODE 2 when Lambda = 3.0

Table 10: Stiff ODE 2, Lambda = -12.0 Error Table

Number of Steps	IE-Pre-2	BDF2	IE-Pre-Post-3	BDF3
40	3.7332E-04	1.4954E-04	3.2668E-04	2.7097E-06
80	9.4161E-05	3.7686E-05	4.1778E-05	2.9170E-07
160	2.3643E-05	9.4555E-06	5.2664E-06	3.3490E-08
320	5.9234E-06	2.3679E-06	6.6085E-07	3.9995E-09
640	1.4824E-06	5.9247E-07	8.2759E-08	4.8824E-10
1280	3.7080E-07	1.4818E-07	1.0354E-08	6.0298E-11
2560	9.2725E-08	3.7052E-08	1.2948E-09	7.4910E-12
5120	2.3185E-08	9.2639E-09	1.6162E-10	9.3400E-13
10240	5.7961E-09	2.3161E-09	2.0634E-11	1.1700E-13

Table 11: Stiff ODE 2, Lambda = 1.0 Error Table

Number of Steps	IE-Pre-2	BDF2	IE-Pre-Post-3	BDF3
40	3.4167E-01	8.8169E-02	2.1288E-02	8.0383E-03
80	8.8982E-02	2.2602E-02	2.8775E-03	1.0515E-03
160	2.2804E-02	5.7305E-03	3.7419E-04	1.3452E-04
320	5.7790E-03	1.4433E-03	4.7708E-05	1.7012E-05
640	1.4550E-03	3.6222E-04	6.0227E-06	2.1390E-06
1280	3.6508E-04	9.0731E-05	7.5656E-07	2.6815E-07
2560	9.1439E-05	2.2705E-05	9.4796E-08	3.3568E-08
5120	2.2881E-05	5.6790E-06	1.1848E-08	4.1993E-09
10240	5.7229E-06	1.4201E-06	1.4686E-09	5.2552E-10

Table 12: Stiff ODE 2, Lambda = 3.0 Error Table

Number of Steps	IE-Pre-2	BDF2	IE-Pre-Post-3	BDF3
40	3.5941E+03	1.1312E+03	6.5642E+02	1.9514E+02
80	8.3630E+02	2.8661E+02	9.3534E+01	2.7570E+01
160	2.1215E+02	7.3554E+01	1.2753E+01	3.6923E+00
320	5.4174E+01	1.8732E+01	1.6706E+00	4.7836E-01
640	1.3740E+01	4.7336E+00	2.1388E-01	6.0888E-02
1280	3.4633E+00	1.1902E+00	2.7059E-02	7.6805E-03
2560	8.6961E-01	2.9843E-01	3.4029E-03	9.6445E-04
5120	2.1789E-01	7.4721E-02	4.2663E-04	1.2083E-04
10240	5.4535E-02	1.8694E-02	5.3412E-05	1.5121E-05

3.3 Harmonic Oscillator

3.3.1 Problem Statement

The next problem we consider is a simplified Harmonic Oscillator, given as:

$$u'' + \omega^2 u = 0$$
$$u(0) = 1\,,\; u'(0) = 0$$

This problem has the solution:

$$u(t) = \cos \omega t$$

As we increase ω, the solution oscillates more rapidly. This differential equation also presents an opportunity to test our method implementations on vector inputs. Using the substitutions $y_1 = u$ and $y_2 = u'$, we can rewrite the problem as

$$\frac{dy}{dt} = \begin{bmatrix} y_2 \\ -\omega^2 y_1 \end{bmatrix} \quad , \quad y(0) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Solve on the interval $0 \leq t \leq 2\pi$.

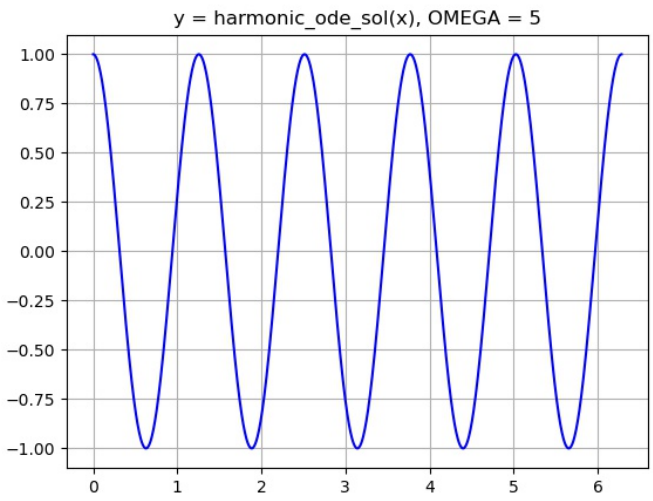

Figure 16: True Solution for the Harmonic Oscillator Omega = 5.0

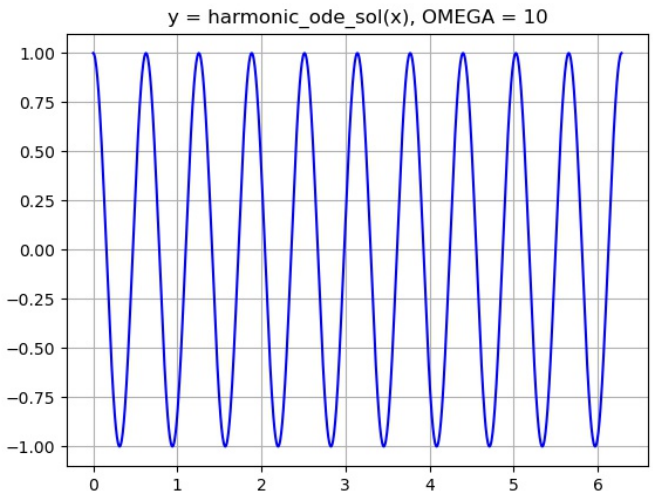

Figure 17: True Solution for the Harmonic Oscillator Omega = 10.0

Table 13: Harmonic ODE, Omega = 5.0 Error Table

Number of Steps	IE-Pre-2	BDF2	IE-Pre-Post-3	BDF3
40	9.6563E-01	1.1652E+00	1.0548E+01	2.5423E+00
80	1.3171E+00	8.4141E-01	8.0214E-01	4.2550E-01
160	4.9784E-01	1.2032E-01	2.6517E-01	5.7772E-02
320	6.6587E-02	1.2066E-02	3.2043E-02	7.3291E-03
640	8.4662E-03	1.2343E-03	4.0057E-03	9.2328E-04
1280	1.3146E-03	1.3554E-04	5.0194E-04	1.1580E-04
2560	2.4807E-04	1.5737E-05	6.2826E-05	1.4497E-05
5120	5.3292E-05	1.8910E-06	7.8580E-06	1.8133E-06
10240	1.2319E-05	2.3159E-07	9.8253E-07	2.2673E-07

3.3.2 Numerical Results

Table 14: Harmonic ODE, Omega = 10.0 Error Table

Number of Steps	IE-Pre-2	BDF2	IE-Pre-Post-3	BDF3
40	1.0000E+00	9.9918E-01	2.6031E+01	2.6382E+00
80	1.0002E+00	1.0068E+00	1.4079E+02	6.2370E+00
160	8.9692E-01	1.4605E+00	9.4896E+00	9.5662E-01
320	1.1396E+00	3.5303E-01	5.9785E-01	1.1985E-01
640	1.7369E-01	3.3796E-02	6.5369E-02	1.4780E-02
1280	1.7875E-02	3.1019E-03	8.0438E-03	1.8516E-03
2560	2.1467E-03	3.1101E-04	1.0052E-03	2.3189E-04
5120	3.2929E-04	3.3975E-05	1.2572E-04	2.9010E-05
10240	6.1997E-05	3.9383E-06	1.5720E-05	3.6276E-06

4.0 Variable Timestep

We follow the definition of discrete curvature presented in [8], pages 7-8. Briefly, we find the Lagrange interpolating polynomial for y_{n+1}, y_n and y_{n-1}. Call this quadratic interpolant $\phi(t)$:

$$\phi(t) = y_{n+1}\ell_{n+1}(t) + y_n\ell_n(t) + y_{n-1}\ell_{n-1}(t)$$

$$\phi(t) = y_{n+1}\frac{(t-t_{n-1})(t-t_n)}{(t_{n+1}-t_{n-1})(t_{n+1}-t_n)} + y_n\frac{(t-t_{n-1})(t-t_{n+1})}{(t_n-t_{n-1})(t_n-t_{n+1})}$$
$$+ y_{n-1}\frac{(t-t_n)(t-t_{n+1})}{(t_{n-1}-t_n)(t_{n-1}-t_{n+1})}$$

Differentiate twice to find the discrete second difference $\phi''(t)$:

$$\phi''(t) = \frac{2}{(t_{n+1}-t_{n-1})(t_{n+1}-t_n)}y_{n+1} + \frac{2}{(t_n-t_{n-1})(t_n-t_{n+1})}y_n$$
$$+ \frac{2}{(t_{n-1}-t_n)(t_{n-1}-t_{n+1})}y_{n-1}$$

Denote k_n as the timestep between t_n and t_{n+1}. Likewise, denote k_{n-1} as the timestep from t_{n-1} to t_n. Then we have:

$$\phi''(t) = \frac{2}{(k_{n-1}+k_n)k_n}y_{n+1} + \frac{2}{k_{n-1}(-k_n)}y_n + \frac{2}{(-k_n)(-(k_n+k_{n-1}))}y_{n-1}$$

Curvature κ_n, as defined in [8], is then $\phi''(t)$ scaled by $k_{n-1}k_n$:

$$\kappa_n = k_{n-1}k_n\phi''$$

$$\kappa_n = \frac{2k_{n-1}}{k_n+k_{n-1}}y_{n+1} - 2y_n + \frac{2k_n}{k_n+k_{n-1}}$$

Similar ideas are presented in [13] [11]

The underlying idea for both constant and variable timestep is

$$\begin{aligned}
&\text{Step 1) Pre-filter} && \tilde{y}_n = y_n - \frac{\alpha}{2}\kappa_{n-1} \\
&\text{Step 2) IE Solve} && \frac{y_{n+1} - \tilde{y}_n}{k_n} = f(t_{n+1}, y_{n+1}) \\
&\text{Step 3) Post-filter} && y_{n+1}^{3rd} = y_{n+1} - \beta(\kappa_n - \kappa_{n-1})
\end{aligned}$$

where α and β are chosen accordingly for 2nd and 3rd order. For the constant time step methods in the previous sections, the curvatures at steps $n-1$ and n are:

$$\kappa_{n-1} = y_n - 2y_{n-1} + y_{n-2}$$

$$\kappa_n = y_{n+1} - 2y_n + y_{n-1}$$

Now for variable timestep, we use

$$\kappa_{n-1} = \frac{2k_{n-2}}{k_{n-1} + k_{n-2}} y_n - 2y_{n-1} + \frac{2k_{n-1}}{k_{n-1} + k_{n-2}} y_{n-2}$$

$$\kappa_n = \frac{2k_{n-1}}{k_n + k_{n-1}} y_{n+1} - 2y_n + \frac{2k_n}{k_n + k_{n-1}} y_{n-1}$$

We can use direct Taylor expansions to determine pre-filter α.

4.1 Adaptive Timestep Pre-filter

Consider first Steps 1) and 2) above with the new curvature:

$$\tilde{y}_n = y_n - \frac{\alpha}{2}\left(\frac{2k_{n-2}}{k_{n-1}+k_{n-2}}y_n - 2y_{n-1} + \frac{2k_{n-1}}{k_{n-1}+k_{n-2}}y_{n-2}\right)$$

$$\frac{y_{n+1}-\tilde{y}_n}{k_n} = f(t_{n+1}, y_{n+1})$$

Substitute the pre-filter into the solve step, gather terms and simplify:

$$y_{n+1} - \left(y_n - \frac{\alpha}{2}\left(\frac{2k_{n-2}}{k_{n-1}+k_{n-2}}y_n - 2y_{n-1} + \frac{2k_{n-1}}{k_{n-1}+k_{n-2}}y_{n-2}\right)\right) = k_n f(t_{n+1}, y_{n+1})$$

$$y_{n+1} + \frac{(\alpha-1)k_{n-2} - k_{n-1}}{k_{n-1}+k_{n-2}}y_n - \alpha y_{n-1} + \frac{\alpha k_{n-1}}{k_{n-1}+k_{n-2}}y_{n-2} - k_n(y')_{n+1} = 0$$

Taylor expand around each of the terms:

$$\begin{aligned}
y_{n+1} &= y_{n+1} \\
\frac{(\alpha-1)k_{n-2}-k_{n-1}}{k_{n-1}+k_{n-2}}y_n &= \frac{(\alpha-1)k_{n-2}-k_{n-1}}{k_{n-1}+k_{n-2}}y_{n+1} - \frac{(\alpha-1)k_{n-2}-k_{n-1}}{k_{n-1}+k_{n-2}}k_n y' \\
&\quad + \frac{((\alpha-1)k_{n-2}-k_{n-1})}{k_{n-1}+k_{n-2}}\frac{k_n^2}{2}y'' + \mathcal{O}(k_n^3) \\
-\alpha y_{n-1} &= -\alpha y_{n+1} + \alpha(k_n+k_{n-1})y' \\
&\quad - \alpha\frac{(k_n+k_{n-1})^2}{2}y'' + \mathcal{O}((k_n+k_{n-1})^3) \\
\frac{\alpha k_{n-1}}{k_{n-1}+k_{n-2}}y_{n-2} &= \frac{\alpha k_{n-1}}{k_{n-1}+k_{n-2}}y_{n+1} - \frac{\alpha k_{n-1}}{k_{n-1}+k_{n-2}}(k_n+k_{n-1}+k_{n-2})y' \\
&\quad + \frac{\alpha k_{n-1}}{k_{n-1}+k_{n-2}}\frac{(k_n+k_{n-1}+k_{n-2})^2}{2}y'' + \mathcal{O}(k_n+k_{n-1}+k_{n-2})^3) \\
-k_n(y')_{n+1} &= -k_n y'
\end{aligned}$$

α must be chosen so that the y_{n+1} , y' and y'' terms above sum to zero.

Summing the coefficients on the y_{n+1} terms above:

$$= 1 + \frac{(\alpha-1)k_{n-2} - k_{n-1}}{k_{n-1}+k_{n-2}} + (-\alpha) + \frac{\alpha k_{n-1}}{k_{n-1}+k_{n-2}}$$

$$= \frac{k_{n-1}+k_{n-2}+\alpha k_{n-2} - k_{n-2} - k_{n-1} - \alpha k_{n-1} - \alpha k_{n-2} + \alpha k_{n-1}}{k_{n-1}+k_{n-2}} = 0$$

This is zero regardless of the choice of α. Similarly, sum the coefficients on the y' terms above:

$$= -\frac{(\alpha-1)k_{n-2} - k_{n-1}}{k_{n-1}+k_{n-2}} k_n + \alpha(k_n + k_{n-1}) - \frac{\alpha k_{n-1}}{k_{n-1}+k_{n-2}}(k_n + k_{n-1} + k_{n-2}) - k_n$$

$$= \frac{-\alpha k_{n-2}k_n + k_{n-2}k_n + k_{n-1}k_n}{k_{n-1}+k_{n-2}} + \frac{\alpha k_{n-1}k_n + \alpha k_{n-2}k_n + \alpha k_{n-1}^2 + \alpha k_{n-2}k_{n-1}}{k_{n-1}+k_{n-2}}$$

$$+ \frac{-\alpha k_{n-1}k_n - \alpha k_{n-1}^2 - \alpha k_{n-2}k_{n-1}}{k_{n-1}+k_{n-2}} + \frac{-k_{n-1}k_n - k_{n-2}k_n}{k_{n-1}+k_{n-2}} = 0$$

This is also trivially zero. Finally, sum the coefficients on the y'' terms:

$$= \frac{((\alpha-1)k_{n-2} - k_{n-1})}{k_{n-1}+k_{n-2}} \frac{k_n^2}{2} + -\alpha \frac{(k_n + k_{n-1})^2}{2} + \frac{\alpha k_{n-1}}{k_{n-1}+k_{n-2}} \frac{(k_n + k_{n-1} + k_{n-2})^2}{2}$$

$$= \frac{k_n^2((\alpha-1)k_{n-2} - k_{n-1})}{2(k_{n-1}+k_{n-2})} + \frac{-\alpha(k_n + k_{n-1})^2(k_{n-1}+k_{n-2})}{2(k_{n-1}+k_{n-2})} + \frac{\alpha k_{n-1}(k_n + k_{n-1} + k_{n-2})^2}{2(k_{n-1}+k_{n-2})}$$

$$= \frac{\alpha k_n^2 k_{n-2} - k_n^2 k_{n-2} - k_n^2 k_{n-1}}{2(k_{n-1}+k_{n-2})}$$

$$+ \frac{-\alpha k_n^2 k_{n-1} - \alpha k_n^2 k_{n-2}^2 - 2\alpha k_n k_{n-1}^2 - 2\alpha k_n k_{n-1} k_{n-2} - \alpha k_{n-1}^3 - \alpha k_{n-1}^2 k_{n-2}}{2(k_{n-1}+k_{n-2})}$$

$$+ \frac{\alpha k_n^2 k_{n-1} + 2\alpha k_n k_{n-1}^2 + 2\alpha k_n k_{n-1} k_{n-2} + \alpha k_{n-1}^3 + 2\alpha k_{n-1}^2 k_{n-2} + \alpha k_{n-1} k_{n-2}^2}{2(k_{n-1}+k_{n-2})}$$

We require this to sum to zero:

$$0 = -k_n^2 k_{n-2} - k_n^2 k_{n-1} - \alpha k_{n-1}^2 k_{n-2} + 2\alpha k_{n-1}^2 k_{n-2} + \alpha k_{n-1} k_{n-2}^2$$

$$0 = -k_n^2 k_{n-2} - k_n^2 k_{n-1} + \alpha k_{n-1}^2 k_{n-2} + \alpha k_{n-1} k_{n-2}^2$$

$$\alpha k_{n-1}k_{n-2}(k_{n-1}+k_{n-2}) = k_n^2(k_{n-1}+k_{n-2}) \iff \alpha = \frac{k_n^2}{k_{n-1}k_{n-2}}$$

Now that α has been found, the variable timestep method so far is:

Step 1) Pre-filter $\quad \tilde{y}_n = y_n - \dfrac{k_n^2}{2k_{n-2}k_{n-1}}\left(\dfrac{2k_{n-2}}{k_{n-1}+k_{n-2}}y_n - 2y_{n-1} + \dfrac{2k_{n-1}}{k_{n-1}+k_{n-2}}y_{n-2}\right)$

Step 2) IE Solve $\quad \dfrac{y_{n+1}-\tilde{y}_n}{k_n} = f(t_{n+1}, y_{n+1})$

Step 3) Post-filter $\quad y_{n+1}^{3rd} = y_{n+1} - \beta(\kappa_n - \kappa_{n-1})$

4.2 Adaptive Timestep Post-Filter

To find the adaptive timestep post-filter, we use an alternative method to the Taylor expansions used in previous sections. We require the method to be exact on $y = t^3$. For simplicity, use $n = 3$. Then we'll work with y_1, y_2, y_3, y_4 in the method and:

$$y_1 = t_1^3 \qquad t_1 = k_0$$

$$y_2 = t_2^3 \qquad t_2 = k_0 + k_1$$

$$y_3 = t_3^3 \qquad t_3 = k_0 + k_1 + k_2$$

$$y_4 = t_4^3 \qquad t_4 = k_0 + k_1 + k_2 + k_3$$

The difference of curvatures in the post-filter step is given by:

$$\begin{aligned}\kappa_n - \kappa_{n-1} &= \frac{2k_2}{k_2+k_3}y_4 - 2y_3 + \frac{2k_3}{k_2+k_3}y_2 - \left(\frac{2k_1}{k_1+k_2}y_3 - 2y_2 + \frac{2k_2}{k_1+k_2}y_1\right) \\ &= -\frac{2k_2y_1}{k_1+k_2} + \left(\frac{2k_3}{k_2+k_3}+2\right)y_2 + \left(-\frac{2k_1}{k_1+k_2}-2\right)y_3 + \frac{2k_2}{k_2+k_3}y_4\end{aligned}$$

Then the post-filtered y_4 is:

$$\begin{aligned}y_4^{3rd} &= y_4 - \beta\left(\kappa_3 - \kappa_2\right) \\ &= \frac{2\beta k_2}{k_1+k_2}y_1 - \beta\left(\frac{2k_3}{k_2+k_3}+2\right)y_2 - \beta\left(-\frac{2k_1}{k_1+k_2}-2\right)y_3 + \left(1 - \frac{2\beta k_2}{k_2+k_3}\right)y_4\end{aligned}$$

Denote y_4 as y_4^*. Denote y_4^{3rd} as y_4:

$$y_4 = \frac{2\beta k_2}{k_1+k_2}y_1 - \beta\left(\frac{2k_3}{k_2+k_3}+2\right)y_2 - \beta\left(-\frac{2k_1}{k_1+k_2}-2\right)y_3 + \left(1-\frac{2\beta k_2}{k_2+k_3}\right)y_4^*$$

We solve for y_4^* and plug it into the method, along with the prefilter step:

$$y_4^* = \frac{2\beta k_2}{(k_1+k_2)\left(\frac{2\beta k_2}{k_2+k_3}-1\right)}y_1 - \frac{\beta\left(\frac{2k_3}{k_2+k_3}+2\right)}{\frac{2\beta k_2}{k_2+k_3}-1}y_2 - \frac{\beta\left(-\frac{2k_1}{k_1+k_2}-2\right)}{\frac{2\beta k_2}{k_2+k_3}-1}y_3 - \frac{1}{\frac{2\beta k_2}{k_2+k_3}-1}y_4$$

$$\tilde{y}_3 = y_3 - \frac{k_3^2}{2k_1k_2}\left(\frac{2k_2}{k_1+k_2}y_1 - 2y_2 + \frac{2k_1}{k_1+k_2}y_3\right)$$

The method with y_1, y_2, y_3, y_4 defined above is:

$$y_4^* - \tilde{y}_3 = k_3(y_4^*)'$$

$$\begin{aligned}
\iff & \frac{2\beta k_2}{(k_1+k_2)\left(\frac{2\beta k_2}{k_2+k_3}-1\right)}k_0^3 - \frac{\beta\left(\frac{2k_3}{k_2+k_3}+2\right)}{\frac{2\beta k_2}{k_2+k_3}-1}\left(k_0+k_1\right)^3 \\
& - \frac{\beta\left(-\frac{2k_1}{k_1+k_2}-2\right)}{\frac{2\beta k_2}{k_2+k_3}-1}\left(k_0+k_1+k_2\right)^3 - \frac{1}{\frac{2\beta k_2}{k_2+k_3}-1}\left(k_0+k_1+k_2+k_3\right)^3 \\
& - \left(\left(k_0+k_1+k_2\right)^3 - \frac{k_3^2}{2k_1k_2}\left(\frac{(2k_2)\,k_0^3}{k_1+k_2} - 2\left(k_0+k_1\right)^3 + \frac{(2k_1)\left(k_0+k_1+k_2\right)^3}{k_1+k_2}\right)\right) = \\
& k_3\left(\frac{2\beta k_2}{(k_1+k_2)\left(\frac{2\beta k_2}{k_2+k_3}-1\right)}3k_0^2 - \frac{\beta\left(\frac{2k_3}{k_2+k_3}+2\right)}{\frac{2\beta k_2}{k_2+k_3}-1}3(k_0+k_1)^2\right. \\
& \left. - \frac{\beta\left(-\frac{2k_1}{k_1+k_2}-2\right)}{\frac{2\beta k_2}{k_2+k_3}-1}3(k_0+k_1+k_2)^2 - \frac{1}{\frac{2\beta k_2}{k_2+k_3}-1}3(k_0+k_1+k_2+k_3)^2\right)
\end{aligned}$$

Solve the equation above for β:

$$\beta = \frac{-k_3^2\left(k_2+k_3\right)\left(k_1+2\left(k_2+k_3\right)\right)}{2k_2\left(2\left(k_2+k_3\right)k_1^2 + \left(k_2^2-5k_3k_2-7k_3^2\right)k_1 + 3k_0\left(k_1-k_3\right)\left(k_2+k_3\right) - 2k_2k_3\left(k_2+k_3\right)\right)}$$

Thus we choose the adaptive post-filter to be:

$$\frac{-k_n^2(k_{n-1}+k_n)(k_{n-2}+2(k_{n-1}+k_n))}{2k_{n-1}\left(2(k_{n-1}+k_n)k_{n-2}^2+\left(k_{n-1}^2-5k_nk_{n-1}-7k_n^2\right)k_{n-2}+3k_{n-3}(k_{n-2}-k_n)(k_{n-1}+k_n)-2k_{n-1}k_n(k_{n-1}+k_n)\right)}$$

See the appendices for using this idea to find the constant step filters and the adaptive step pre-filter.

4.3 Code Implementation: Adaptive Timestep

Here is a prototype of the adaptive timestep method with halving and doubling. The script *test1_adapt.py* shows it achieves a comparable error to IE-Pre-Post-3 in fewer steps on the model problem. The result for IE-Pre-Post-3 can also be seen in row 6 of Table 4 in Section 2.2.3.

python *test1_adapt.py*

IE-Pre-Post-3

Number of steps taken: 1280

Error at final step: 1.3993890490837657e-09

IE-Pre-Post-Adapt

Number of steps taken: 1000

Error at final step: 2.9330733397614495e-09

```
# Adaptive IE-filtered embedded pair
def ie_pre_post_adapt ( f_ode , t_range , y_init , dt , TOL , max_steps ):
    """
        RK3 until there are enough steps for adaptive filter
    """
    import numpy as np
    from scipy.optimize import fsolve

    m = np.size ( y_init )
    t = np.zeros ( 1 )
    y = np.zeros ( ( 1 , m ) )
    e = np.zeros( 1 )

    k = 0
    t[k] = t_range[0]
    y[k,:] = y_init

    ### BEGIN RK3 for 0 < k < 2 ============================== ###
    while ( k < 2 ):

        # add new t , y , e values
        t = np.append ( t , 0.0 )
        y = np.vstack( ( y , np.zeros( m ) ) )
        e = np.append ( e , 0.0 )

        ta = t[k] + dt / 2
```

```
        ya = y[k] + dt / 2 * f_ode( t[k] , y[k] )

        tb = t[k] + dt
        yb = y[k] + dt * ( 2 * f_ode( ta , ya ) - f_ode( t[k] , y[k] ) )

        # estimate solution / true step
        y[k+1] = y[k,:] + \
        dt * ( f_ode ( t[k] , y[k] ) + 4.0 * f_ode ( ta , ya ) + f_ode( tb , yb ) ) / 6.0

        t[k+1] = t[k] + dt
        k = k + 1
    ### END RK3 ================================================== ###

    # initial the first three timesteps
    knm1 = t[k] - t[k-1]
    knm2 = t[k-1] - t[k-2]
    knm3 = t[k-2] - t[k-3]

    # to be updated in loop
    kn = dt

    while ( t[k] < t_range[1] ):

        # add new t , y , e values in outer loop
        # will be updated until inner loop is satisfied
        t = np.append ( t , 0.0 )
        y = np.vstack ( ( y ,np.zeros ( m ) ) )
        e = np.append ( e , 0.0 )

        while ( True ):
            t[k+1] = t[k] + kn

            if( t[k+1] > t_range[1] ):
                t[k+1] = t_range[1]
                kn = t[k+1] - t[k]

            temp_yk , kappa_nm1 = IE_adaptive_pre_filter( y[k,:] , y[k-1,:] , y[k-2,:] , kn
    , knm1 , knm2 )
            y[k+1,:] = fsolve ( bef_updated , y[k+1, :] , args = ( f_ode , kn , temp_yk , t[
    k+1] ) )
            ynp1_order2 = y[k+1,0]
            y[k+1,:] = IE_adaptive_post_filter ( y[k+1,:] , y[k,:] , y[k-1,:] , kappa_nm1 ,
    kn , knm1 , knm2 , knm3 )
            ynp1_order3 = y[k+1,0]


            e[k+1] = np.linalg.norm( ynp1_order2 - ynp1_order3 )

            if ( TOL * kn < e[k+1] ):
                kn = kn / 2.0
            elif ( e[k+1] < TOL * kn  / 32.0 ):
                kn = kn * 2.0
                break
            else:
                break

        k = k + 1
        # add this line to reset steps
        kn = dt
        knm1 = t[k] - t[k-1]
        knm2 = t[k-1] - t[k-2]
        knm3 = t[k-2] - t[k-3]

        # test
        if ( k > max_steps ):
            print('Reached maximum number of allowable steps.')
            break

    return t , y , e


```

```
95  def IE_adaptive_pre_filter( y_n , y_nm1 , y_nm2 , kn , knm1 , knm2 ):
96      kappa_nm1 = (2*knm2/(knm1+knm2)) * y_n - 2 * y_nm1 + (2*knm1/(knm1+knm2)) * y_nm2
97      y_ntilde = y_n - (kn**2/(2*knm1*knm2)) * kappa_nm1
98
99      return y_ntilde , kappa_nm1
100
101 def IE_adaptive_post_filter( y_np1 , y_n , y_nm1 , kappa_nm1 , kn , knm1 , knm2 , knm3 ):
102     kappa_n = (2*knm1/(kn+knm1)) * y_np1 - 2 * y_n + (2*kn/(kn+knm1)) * y_nm1
103     post_filter = -( kn**2 * (knm1 + kn) * (knm2 + 2 * ( knm1 + kn ) ) ) / \
104         ( 2*knm1*(2*knm2**2*(knm1+kn) + 3*knm3*(knm2-kn)*(knm1+kn) - 2*knm1*kn*(knm1+kn) +
        knm2*(kn**2 - 5*knm1*kn - 7*kn**2)) )
105     ynp1_first = y_np1 - post_filter * ( kappa_n - kappa_nm1 )
106
107     return ynp1_first
108
109 def bef_updated ( yp, f, kn, yo, tp ):
110
111   value = yp - yo - kn * f ( tp, yp )
112
113   return value
```

Listing 4.1: Python IE-Pre-Post-Adaptive

5.0 Conclusions

The filtered methods IE-Pre-2 and IE-Pre-Post-3 have errors close to or comparable to the errors of BDF2 and BDF3, respectively, in many of the constant step tests. One key advantage of IE-Pre-2 and IE-Pre-Post-3 is that they are a natural embedded pair. The methods have overlapping steps and only a single "solve" step. These are both beneficial from a computational standpoint when adapting the timestep.

Further benefits of the new filtered methods are that they require only minor modifications of existing code, they are computationally cheap and IE-Pre-2 has a relatively larger absolute stability region. The combination of these benefits suggests that pre- and post-filtered versions of many of the classic numerical methods convenient and practical solutions.

Herein, we have found adaptive timestep pre- and post-filters for the IE-based methods. Next steps are further research on the implementation and analysis of the adaptive timestep method. Also, adaptive timesteps can be explored for higher order filtered versions of the Midpoint, BDF2 and BDF3 methods.

Regarding the code implementations: In order for IE-Pre-Post-3 for achieve third order convergence, the initial steps need to be taken carefully. There are two versions in the code repository. One version uses RK3 in the first few steps until the method has enough points to begin filtering. The other version builds up with Implicit Euler, then BDF2 until the method has enough values to begin filtering. Both versions appear to work equally as well.

Appendix A Graphing stability regions with the Boundary Locus Method

A.1 boundary_locus.py

```
1
2  def boundary_locus( jpg_name , title , h_hat , color , scalar  , x_range , y_range ):
3      import matplotlib.pyplot as plt
4      import matplotlib.colors as mcolors
5      import numpy as np
6
7      fig , ax = plt.subplots()
8      s = np.zeros( 10001 , dtype=complex)
9      theta = np.linspace ( 0.0 , scalar * np.pi , 10001 )
10
11     for i , val in enumerate( theta ):
12         s[i] = complex ( np.cos( val ) , np.sin ( val ) )
13     # print( s )
14
15     boundary_curve = h_hat( s )
16     x = np.real( boundary_curve )
17     y = np.imag( boundary_curve )
18     # plt.xlim(-1, 5)
19     # plt.ylim(-3, 3)
20     plt.xlim( x_range[0] , x_range[1] )
21     plt.ylim( y_range[0] , y_range[1] )
22     plt.plot ( x, y, color )
23     plt.grid ( True )
24     ax.axhline(y=0, color='k')
25     ax.axvline(x=0, color='k')
26     plt.xlabel('Re axis')
27     plt.ylabel('Im axis')
28     plt.title ( title )
29     plt.savefig ( jpg_name )
30     plt.show ( )
31     plt.close ( )
32
33 def boundary_locus_fill( jpg_name , title , h_hat , color , back_color , scalar  , x_range ,
       y_range ):
34     import matplotlib.pyplot as plt
35     import matplotlib.colors as mcolors
36     import numpy as np
37
38     fig , ax = plt.subplots()
39     s = np.zeros( 10001 , dtype=complex)
40     theta = np.linspace ( 0.0 , scalar * np.pi , 10001 )
41
42     for i , val in enumerate( theta ):
43         s[i] = complex ( np.cos( val ) , np.sin ( val ) )
44     # print( s )
45     ax.set_facecolor( back_color )
46     boundary_curve = h_hat( s )
47     x = np.real( boundary_curve )
48     y = np.imag( boundary_curve )
49     plt.xlim( x_range[0] , x_range[1] )
50     plt.ylim( y_range[0] , y_range[1] )
51     plt.fill ( x, y, color )
52     plt.grid ( False )
53     ax.axhline(y=0, color='k')
54     ax.axvline(x=0, color='k')
55     plt.xlabel('Re axis')
56     plt.ylabel('Im axis')
57     plt.title ( title )
58     plt.savefig ( jpg_name )
59     plt.show ( )
```

```
60      plt.close ( )
61
62  if( __name__ == '__main__' ):
63
64      # boundary curves IE-Pre-2 / IE-Pre-Post-3
65      boundary_locus( 'iepre2_boundary.jpg' , 'IE-Pre-2 Stability Region Boundary Curve' , \
66                      lambda r : ( r**3 - (1.0/2.0) * r**2 - r + (1.0/2.0)) / r**3 , 'b' , 2.0
        , [ -1 , 5 ] , [ -3 , 3] )
67      boundary_locus( 'ieprepost3_boundary.jpg' , 'IE-Pre-Post-3 Stability Region Boundary
        Curve' , \
68                      lambda r : (( 11.0 * r**3 - 18.0 * r**2 + 9.0 * r - 2.0 ) / (11.0 * r**3
         - 15.0 * r**2 + 15.0 * r - 5.0)) ,\
69                      'b' , 2.0 , [ -20 , 20 ] , [ -20 , 20 ] )
70
71      # boundary curves BDF2 / BDF3
72      boundary_locus( 'bdf2_boundary.jpg' , 'BDF2 Stability Region Boundary Curve' , \
73                  lambda r : ( 3.0 * r**2 - 4.0 * r + 1 ) / ( 2.0 * r**2 ) , 'r' , 2.0 , [ -1
        , 5 ] , [ -3 , 3] )
74      boundary_locus( 'bdf3_boundary.jpg' , 'BDF3 Stability Region Boundary Curve' , \
75                      lambda r : (( 11.0 * r**3 - 18.0 * r**2 + 9.0 * r - 2.0 ) / ( 6.0 * r**3
         )) ,\
76                      'r' , 2.0 , [ -2 , 10 ] , [ -6 , 6 ] )
77
78      # filled in stability regions
79      boundary_locus_fill( 'ieprepost3_stability_full.jpg' , 'IE-Pre-Post-3 Stability Region'
        , \
80                  lambda r : (( 11.0 * r**3 - 18.0 * r**2 + 9.0 * r - 2.0 ) / (11.0 * r**3 -
        15.0 * r**2 + 15.0 * r - 5.0)) ,\
81                  'w' , 'b' , 2.0 , [ -20 , 20 ] , [ -20 , 20 ] )
82      boundary_locus_fill( 'iepre2__stability_full.jpg' , 'IE-Pre-2 Stability Region' , \
83                  lambda r : ( r**3 - (1.0/2.0) * r**2 - r + (1.0/2.0)) / r**3  ,\
84                  'w' , 'b' , 2.0 , [ -1 , 5 ] , [ -3 , 3] )
85
86      boundary_locus_fill( 'bdf2_stability_full.jpg' , 'BDF2 Stability Region' , \
87                  lambda r : ( 3.0 * r**2 - 4.0 * r + 1 ) / ( 2.0 * r**2 ) ,  \
88                  'w' , 'r' , 2.0 , [ -1 , 5 ] , [ -3 , 3] )
89      boundary_locus_fill( 'bdf3_stability_full.jpg' , 'BDF3 Stability Region' , \
90                  lambda r : (( 11.0 * r**3 - 18.0 * r**2 + 9.0 * r - 2.0 ) / ( 6.0 * r**3 ))
        ,\
91                  'w' , 'r' , 2.0 , [ -2 , 10 ] , [ -6 , 6 ] )
```

Listing A.1: Code to generate stability regions

A.2 find_stability_roots.py

```
1
2  def find_stability_roots( r , h_hat):
3      import numpy as np
4
5      r_roots = np.roots( r )
6      print( r_roots )
7      found_greater_than_1 = False
8      print(f'root # \t\troot\t\t\t\t\t\tnorm(root)\t\tgreater than one?')
9      for i , root in enumerate ( r_roots ):
10         print(f'root {i} = \t {root}\t{abs(root)}')
11         if( abs(root) > 1 ):
12             found_greater_than_1 = True
13     print('')
14     if( found_greater_than_1 ):
15         print(f'\nFound abs(root) greater than 1 -> not stable in the region containing {
       h_hat}\n')
16     else:
17         print(f'\nStable in the region containing {h_hat}\n')
18     return
19
20 if ( __name__ == '__main__'):
21     # tests for IE-Pre-2
22     h_hat = complex( 1 , 1.25 )
23     find_stability_roots ( [ 1 - h_hat , -(1.0/2.0) , -1.0 , 1.0/2.0 ] , h_hat )
24
25     h_hat = complex( 1 , -1.25 )
26     find_stability_roots ( [ 1 - h_hat , -(1.0/2.0) , -1.0 , 1.0/2.0 ] , h_hat )
27
28     h_hat = complex( 1.0 , 0.0 )
29     find_stability_roots ( [ 1 - h_hat , -(1.0/2.0) , -1.0 , 1.0/2.0 ] , h_hat )
30
31     h_hat = complex( 3.0 , 0.0 )
32     find_stability_roots ( [ 1 - h_hat , -(1.0/2.0) , -1.0 , 1.0/2.0 ] , h_hat )
33
34
35     # tests for IE-Pre-Post-3
36     h_hat = complex( 5.0 , 0 )
37     find_stability_roots ( [ 11.0 - 11.0*h_hat , -18.0 + 15.0*h_hat, 9.0 - 15.0*h_hat , -2.0
       + 5.0*h_hat ] , h_hat )
38
39     h_hat = complex( -5.0 , 0 )
40     find_stability_roots ( [ 11.0 - 11.0*h_hat , -18.0 + 15.0*h_hat, 9.0 - 15.0*h_hat , -2.0
       + 5.0*h_hat ] , h_hat )
41
42     h_hat = complex( 5.0 , 5.0 )
43     find_stability_roots ( [ 11.0 - 11.0*h_hat , -18.0 + 15.0*h_hat, 9.0 - 15.0*h_hat , -2.0
       + 5.0*h_hat ] , h_hat )
```

Listing A.2: Test points in subregions formed from boundary curve

A.3 BDF2 / BDF3 Stability Regions

The stability regions for BDF2 and BDF3 are well-known. We recreate them here, following the same methodology as for the filtered methods, to demonstrate and test *boundary_locus.py*. Also, we can compare them to the stability regions for IE-Pre-2 and IE-Pre-Post-3.

BDF2: $$y_{n+2} - \frac{4}{3}y_{n+1} + \frac{1}{3}y_n = \frac{2}{3}kf(t_{n+2}, y_{n+2})$$

BDF3: $$y_{n+3} - \frac{18}{11}y_{n+2} + \frac{9}{11}y_{n+1} - \frac{2}{11}y_n = \frac{6}{11}kf(t_{n+3}, y_{n+3})$$

Use the following inputs for the script:

BDF2: $$\hat{k}(r) = \frac{3r^2 - 4r + 1}{2r^2}$$

BDF3: $$\hat{k}(r) = \frac{11r^3 - 18r^2 + 9r - 2}{6r^3}$$

A.3.1 BDF2

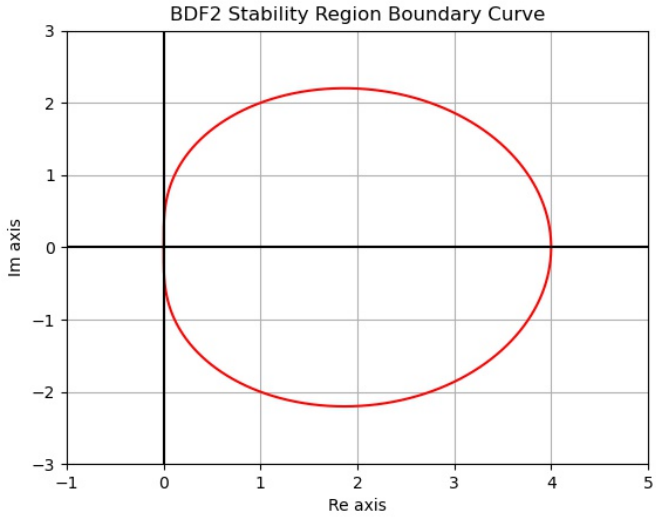

Figure 18: BDF2 Stability Region Boundary Curve

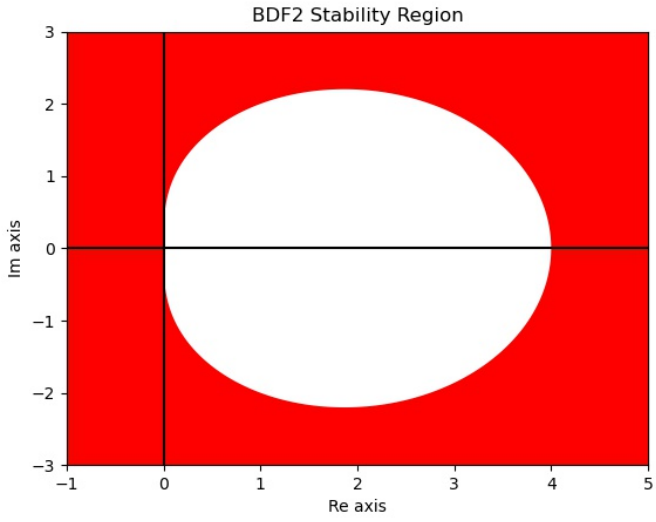

Figure 19: BDF2 Stability Region

A.3.2 BDF3

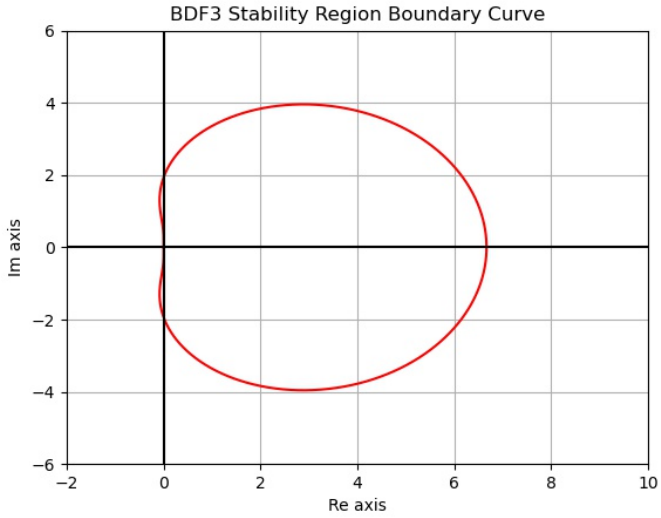

Figure 20: BDF3 Stability Region Boundary Curve

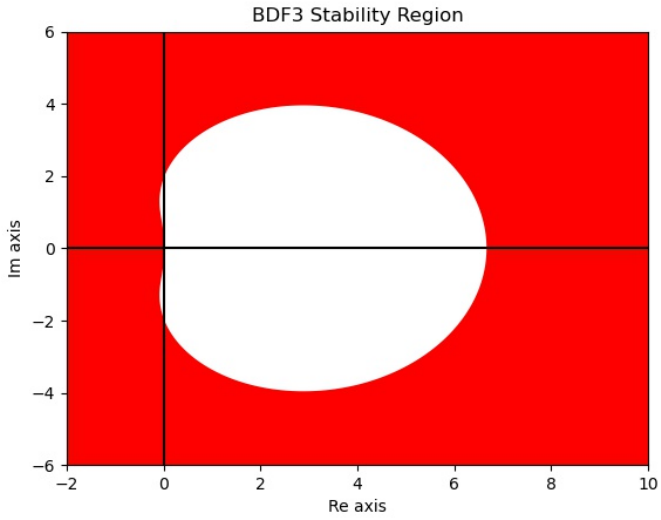

Figure 21: BDF3 Stability Region

Appendix B Constant Timestep Filters Revisited

Method outline

Step 1) Pre-filter	$\tilde{y}_n = y_n - \frac{\alpha}{2}\kappa_{n-1}$
Step 2) IE Solve	$\frac{y_{n+1} - \tilde{y}_n}{k_n} = f(t_{n+1}, y_{n+1})$
Step 3) Post-filter	$y_{n+1}^{3rd} = y_{n+1} - \beta(\kappa_n - \kappa_{n-1})$

For constant timestep:

$$\kappa_{n-1} = y_n - 2y_{n-1} + y_{n-2}$$

$$\kappa_n = y_{n+1} - 2y_n + y_{n-1}$$

B.1 Alternative Pre-filter Calculation

Step 1) Pre-filter $\tilde{y}_n = y_n - \frac{\alpha}{2}(y_n - 2y_{n-1} + y_{n-2})$

Step 2) IE Solve $y_{n+1} - \tilde{y}_n = kf(t_{n+1}, y_{n+1})$

For second order, we require the method to be exact on t^2.

$$y_1 = t_1^2 \qquad t_1 = k$$

$$y_2 = t_2^2 \qquad t_2 = 2k$$

$$y_3 = t_3^2 \qquad t_3 = 3k$$

$$y_4 = t_4^2 \qquad t_4 = 4k$$

When $n = 3$, we have:

$$y_4 - \tilde{y}_3 = kf(t_4, y_4)$$

$$y_4 - (y_3 - \frac{\alpha}{2}(y_3 - 2y_2 + y_1)) = k(y_4)'$$

$$y_4 - y_3 + \frac{\alpha}{2}y_3 - \alpha y_2 + \frac{\alpha}{2}y_1 = k(y_4)'$$

$$y_4 + \left(\frac{\alpha}{2} - 1\right) y_3 - \alpha y_2 + \frac{\alpha}{2}y_1 = k(y_4)'$$

Substitute:

$$(4k)^2 + \left(\frac{\alpha}{2} - 1\right)(3k)^2 - \alpha(2k)^2 + \frac{\alpha}{2}k^2 = k \cdot 2(4k)$$

We find that $\alpha = 1$. This is consistent with the Taylor expansion derivation in Section 2.1.2.

B.2 Alternative Post-filter Calculation

Step 1) Pre-filter $\tilde{y}_n = y_n - \frac{1}{2}(y_n - 2y_{n-1} + y_{n-2})$

Step 2) IE Solve $\frac{y_{n+1} - \tilde{y}_n}{k_n} = f(t_{n+1}, y_{n+1})$

Step 3) Post-filter $y_{n+1}^{3rd} = y_{n+1} - \beta(y_{n+1} - 2y_n + y_{n-1} - (y_n - 2y_{n-1} + y_{n-2}))$

We require the method to be exact on $y = t^3$. For simplicity, use $n = 3$. Then we'll work with y_1, y_2, y_3, y_4 in the method and:

$$y_1 = t_1^3 \qquad t_1 = k$$

$$y_2 = t_2^3 \qquad t_2 = 2k$$

$$y_3 = t_3^3 \qquad t_3 = 3k$$

$$y_4 = t_4^3 \qquad t_4 = 4k$$

In the post-filter step, we have:

$$y_4^{3rd} = y_4 - \beta(y_4 - 2y_3 + y_2 - (y_3 - y_2 + y_1))$$

$$= (1 - \beta)y_4 + 3\beta y_3 - 3\beta y_2 + \beta y_1$$

Denote y_4 as y_4^*. Denote y_4^{3rd} as y_4. Solve for y_4^*.

$$y_4 = (1 - \beta)y_4^* + 3\beta y_3 - 3\beta y_2 + \beta y_1$$

$$\iff y_4^* = \frac{1}{1 - \beta}(y_4 - 3\beta y_3 + 3\beta y_2 - \beta y_1)$$

The method becomes:

$$y_4^* - \tilde{y}_3 = kf(t_4, y_4) = k(y_4^*)'$$

$$\frac{1}{1-\beta}(y_4 - 3\beta y_3 + 3\beta y_2 - \beta y_1) - \left(y_3 - \frac{1}{2}(y_3 - 2y_2 + y_1)\right) =$$

$$\left(\frac{1}{1-\beta}(y_4 - 3\beta y_3 + 3\beta y_2 - \beta y_1)\right)'$$

$$\frac{1}{1-\beta}((4k)^3 - 3\beta(3k)^3 + 3\beta(2k)^3 - \beta(k)^3 - \left((3k)^3 - \frac{1}{2}((3k)^3 - 2(2k)^3 + (k)^3)\right) =$$

$$\left(\frac{1}{1-\beta}(3(4k)^2 - 3\beta \cdot 3(3k)^2 + 3 \cdot 3\beta(2k)^2 - \beta \cdot 3(k)^2)\right)$$

We solve the above equation and find that $\beta = \frac{5}{11}$. This is consistent with the derivation in Section 2.2.2.

Appendix C Adaptive Timestep Pre-filter Revisited

We require the method to be exact on $y = t^2$. For simplicity, use $n = 3$. Then we'll work with y_1, y_2, y_3, y_4 in the method and:

$$y_1 = t_1^2 \qquad t_1 = k_0$$

$$y_2 = t_2^2 \qquad t_2 = k_0 + k_1$$

$$y_3 = t_3^2 \qquad t_3 = k_0 + k_1 + k_2$$

$$y_4 = t_4^2 \qquad t_4 = k_0 + k_1 + k_2 + k_3$$

$$y_4 - \left(y_3 - \frac{\alpha}{2} \left(\frac{2k_1}{k_1 + k_2} y_3 - 2y_2 + \frac{2k_2}{k_1 + k_2} y_1 \right) \right) = k_3 (y_4)'$$

Substitute:

$$(k_0 + k_1 + k_2 + k_3)^2 -$$

$$\left((k_0 + k_1 + k_2)^2 - \frac{\alpha}{2} \left(\frac{2k_1}{k_1 + k_2} (k_0 + k_1 + k_2)^2 - 2(k_0 + k_1)^2 + \frac{2k_2}{k_1 + k_2} (k_0)^2 \right) \right)$$

$$= k_3 2(k_0 + k_1 + k_2 + k_3)$$

Solve the above for α and we find that $\alpha = \frac{k_3^2}{k_2 k_1}$. Then the pre-filter is $\frac{k_n^2}{k_{n-1} k_{n-2}}$. This is consistent with Section 4.1.

Bibliography

[1] A. Alexanderian. On continuous dependence of roots of polynomials on coefficients. https://aalexan3.math.ncsu.edu/articles/polyroots.pdf, 2013.

[2] U. Ascher and L. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. SIAM: Society for Industrial and Applied Mathematics, 1998.

[3] R.A. Asselin. Frequency filter for time integration. *Mon. Weather Review*, 100:487 – 490, 1972.

[4] V. DeCaria, S. Gottlieb, Z. Grant, and W. Layton. A general linear method approach to the design and optimization of efficient, accurate, and easily implemented time-stepping methods in cfd. *Journal of Computational Physics*, 455, 2022.

[5] A. Ditkowski. High order finite difference schemes for the heat equation whose convergence rates are higher than their truncation errors. *Spectral and High Order Methods for Partial Differential Equations ICOSAHOM 2014*, 2015.

[6] A. Ditkowski and S. Gottlieb. Error inhibiting block one-step schemes for ordinary differential equations. *Journal of Scientific Computing*, 73:691–711, 2017.

[7] D. Griffiths and D. Higham. *Numerical Methods for Ordinary Differential Equations Initial Value Problems*. Springer, 2010.

[8] A. Guzel and W. Layton. Time filters increase accuracy of the fully implicit method. *BIT Numerical Mathematics*, 58:301–315, 2018.

[9] G. Harris and C. Martin. The roots of a polynomial vary continuously as a function of the coefficients. *Proc. Amer. Math. Soc.*, 100:390 – 392, 1987.

[10] G. Harris and C. Martin. Large roots yield large coefficients: An addendum to 'the roots of a polynomial vary continuously as a function of the coefficients'. *Proc. Amer. Math. Soc.*, 102:993 – 994, 1988.

[11] E. Kalnay. Atmospheric modeling, data assimilation and predictability. *Cambridge Univ. Press*, 2003.

[12] A. Robert. The integration of a spectral model of the atmosphere by the implicit method. *Proc. WMO/IUGG Symposium on NWP, Japan Meteorological Soc.*, pages 19 – 24, 1969.

[13] P.D. Williams. The raw filter: An improvement to the robert-asselin filter in semi-implicit integrations. *Mon. Weather Rev.*, 139:1996 – 2007, 2011.