Exploring implementations of the parareal algorithm

Owen Rowell, Jemma Shipton

August 2021

1 Parareal implementation

The parareal algorithm has three main steps:

1. An initial coarse solve
2. A fine solve between each point - this is done in parallel
3. Prediction using further coarse steps and correction using the fine solves

Steps 2 and 3 are then done iteratively until a desired tolerance is reached.

Initially, the parareal algorithm was implemented through a single function that took fine and coarse integrating functions as arguments. However, this was then replaced by an object-orientated approach to support adaptive time-stepping (see Section 3) and to increase ease of expansion and readability.

The algorithm is now implemented through the use of an abstract class - BaseParareal - containing relevant virtual and abstract methods. A solve function then manages the iteration and parallelisation of the problem. The internal workings of the class are described in appendix A. Within the class, there are four methods for the user to define:

coarse_integration_func An abstract method that must be defined by the user to compute one coarse step between two given times.

fine_integration An abstract method that does a fine integration between two given times. The method can use as many time steps as desired but must return the time values of these steps if the save_fine flag is True. The flexibility in the number of time steps used allows for the implementation of adaptive solvers. For fixed fine time steps see Section 1.1.

get_coarse_t A virtual method that generates the times for each of the coarse steps. By default, this uses uniformly separated times but could be overridden to use time steps spaced to even the computation load on each fine solve.

is_within_tol A virtual method that determines if the solution has been calculated within the desired tolerance. If the method returns that the tolerance has been reached, this will trigger the solve to end. Any method of error calculation could be defined by the user as only a Boolean return value is required. By default, the method always returns False, meaning that the solve will continue until it reaches the maximum number of iterations, irrespective of the precision reached.

The results of the solve can be accessed through four attributes within the class: t_coarse, x_coarse_corr, t_fine and x_fine. t_fine and x_fine only contain meaningful values if the save_fine flag is set to True, otherwise the values will be discarded during solving to save memory.

These are multi-dimensional arrays of the following forms:

t_coarse This is a simple 1-d numpy array containing the time values for each coarse step

x_coarse_corr This is a 3-d numpy array containing the value of the function at each coarse step, for each iteration. The first dimension indicates the coarse step and the second dimension the iteration.

t_fine This is a 2-d numpy array of lists describing the time values for each fine time step for each coarse step and iteration. As before, the first dimension indicates the coarse step and the second the iteration. In general, the lists are of different lengths, but when using a fixed number of time steps the lists will contain the same number of values.

x_fine This is a 2-d numpy array of lists containing numpy arrays giving the value of the function at each fine time step. The first dimension indicates the coarse step, the second giving the iteration and the position in the lists ("dimension 3") being the fine step.

1.1 Implementation for fixed fine time step

When adaptive time-stepping for the fine solve if not needed, the subclass - FixedParareal - is provided that simplifies the implementation for the user. The fine_integration method is defined and instead a new abstract method must be overridden in a subclass - fine_integration_func. This takes in a list of times and an initial set of conditions and expects back a list of the value of the function at each of those times. Or in the case where the user does not wish the fine values to be saved, a list containing only the value of the function at the last time value should be returned.

1.2 Caching of results

For some systems, the solves can take large amounts of time to solve. To reduce this bottleneck, a mixin class, CachedPR, is provided. Caching is done by hashing the parameters provided to __init__ and comparing to any past runs.

Python's built-in hash function is randomly salted meaning that it does not generate the required consistent values across runs. Instead, each argument is recursively broken into a suitable, fully descriptive string representation that is hashed using md5 from the hashlib library. However, this requires hashing for different variable types to be manually implemented. Currently, this has been done for strings; integers & floats; sets, tuples, lists & numpy arrays; dictionaries; functions and functools.partial. Other data types may work provided they have a suitable str implementation, but it should be checked that this aptly describes the value (i.e. is not just a memory location that will change).

This is implemented through three functions:

get_cache This function should be called at the start of __init__ and provided with all arguments given to the class. The arguments are then hashed and a cache with the corresponding hash is checked for. If one is found then the object is updated with the cached values and True returned.

The rest of the initialisation should then be aborted to avoid overwriting the cached solution. If no cache is found, then initialisation and solving can continue as normal.

save_cache This pickles and saves the object to be used later. It is saved under the hash of its arguments to be easily located when required but another file name can be provided for human-readable storage.

load_cache Unpickles a cached Parareal object saved with the given file name.

The caching could be further improved if the maximum number of iterations was moved from an initialisation parameter to one specified when solving. Currently, a cache with 10 iterations completed will not be utilised in a solve requiring 9 or 11 iterations, for example. The solver could be expanded to detect any completed iterations (or start from scratch if the user wishes) and continue from there. This would mean that the hash of each system would no longer have a dependence on iteration number, allowing systems with different max iterations to be detected as equal.

1.3 Example systems

To test and demonstrate the parareal implementation, solves of two systems were used, taken from [1].

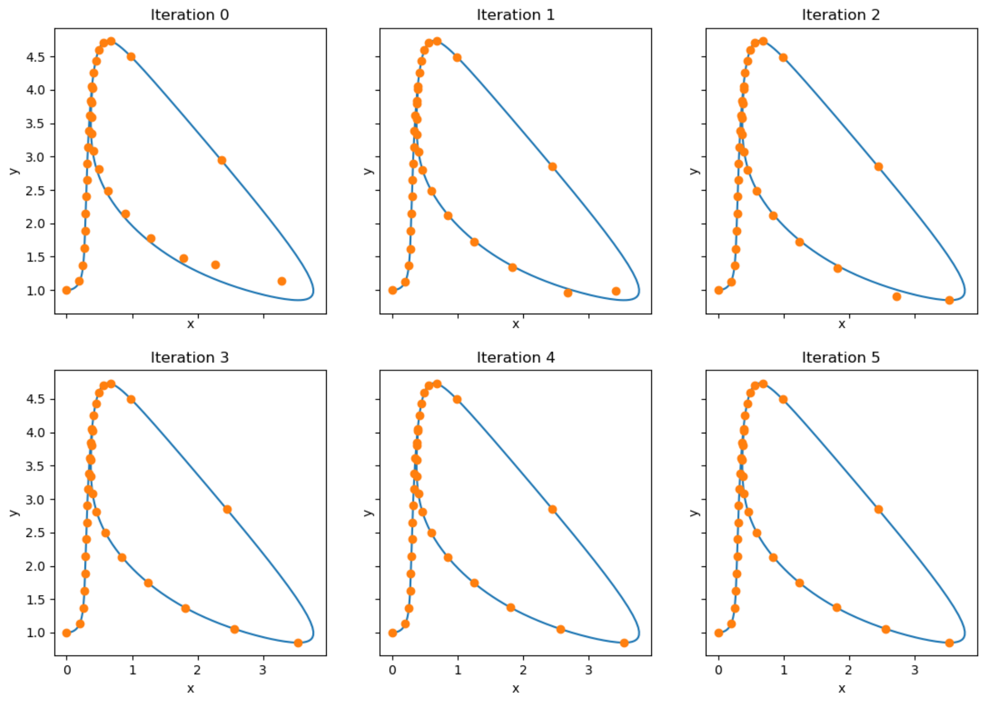


Figure 1: Solve of Brusselator system using the parareal algorithm. The blue line is a solve using fourth-order Runge-Kutta

1.3.1 Brusselator

The Brusselator system is defined by a pair of coupled ODEs given by

$$\dot{x} = \frac{dx}{dt} = A + x^2y - (B+1)x \tag{1}$$

$$\dot{y} = \frac{dy}{dt} = Bx - x^2y, \tag{2}$$

where equation 2 can also be written as

$$\dot{y} = A - x - \dot{x}. \tag{3}$$

The system was first solved using fourth-order Runge-Kutta (RK4) to compare with the parareal results. All solves were done with initial condition $(x, y) = (0, 1)$ and over time period $t \in [0, 12]$. The parameters A and B were set to $A = 1$, $B = 3$. For RK4 a time step of $\Delta t = 1/640$ was used.

For the parareal solve, RK4 was used as both the coarse and fine solver with time steps $\Delta T = 1/32$ and $\Delta t = 1/640$ respectively. The results of the first five iterations of the solve using parareal when compared to the serial solve can be seen in Figure 1, showing the progress of the parareal solve as it converges towards the serial solve with iteration.

This can further be seen in Figure 2, which shows the error between the parareal solution and the RK4 solution. The errors were calculated using the distance between the points given by the two solutions at each time step.

Comparing these plots with those found by Gander and Hairer [1] shows them to be very similar, suggesting the implementation of parareal is correct.

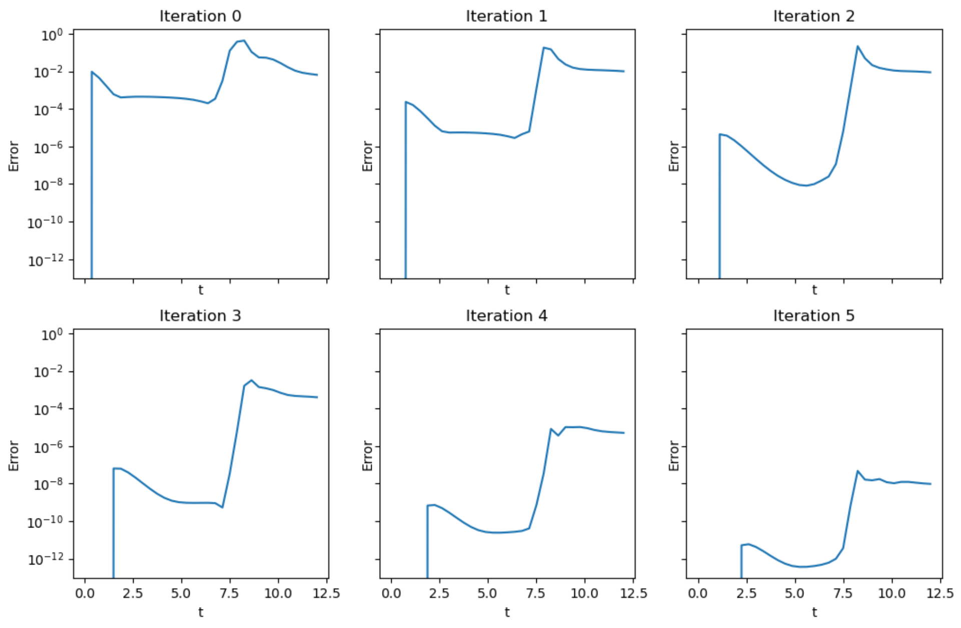


Figure 2: Errors for each iteration of the parareal solve compared to using RK4

1.3.2 Burger's equation

Next, Burger's equation was considered. This is a PDE given by,

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}, \tag{4}$$

which was solved for

$$u(x,0) = \sin(2\pi x), \qquad x, t \in [0,1], \qquad \nu = 0.02.$$

A backwards Euler discretisation in time was used along with centred finite differencing in space. That is

$$\frac{u^{n+1} - u^n}{\Delta t} = f(u^{n+1}), \qquad f(u) = -u\frac{\partial u}{\partial x} + \nu\frac{\partial^2 u}{\partial x^2}, \tag{5}$$

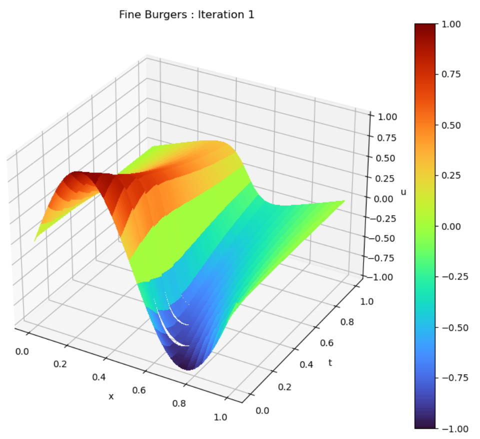
$$\frac{du_i}{dx} = \frac{u_{i+1} - u_{i-1}}{2\Delta x}, \qquad \frac{d^2u_i}{dx^2} = \frac{u_{i+1} - 2u_i + u_{i-1}}{(\Delta x)^2}. \tag{6}$$

Combining Equations 5 and 6 gives the implicit equation

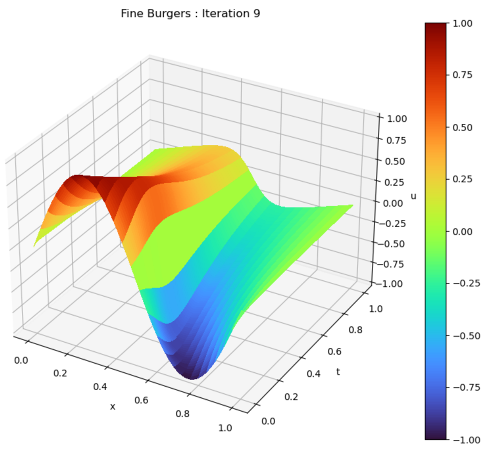
$$u_i^{n+1} = u_i^n - \Delta t u_i^{n+1}\frac{u_{i+1}^{n+1} - u_{i-1}^{n+1}}{2\Delta x} + \Delta t\nu\frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{(\Delta x)^2}, \tag{7}$$

which is then solved iteratively using scipy to give the next time step u_i^{n+1}.

This method is used for both the coarse and fine solvers in the parareal algorithm, with $\Delta T = 0.1$ and $\Delta t = 0.01$ respectively. For the spatial discretisation $\Delta x = 0.02$ was used. The results of the solve can be seen in Figure 3. Initially, the fine solves do not join up due to inaccuracies in the coarse solver, however once the ninth iteration is reached, a stable solution is found creating a single surface.



(a) Solution after one iteration with the fine solves not connection up.



(b) Solution after 9 iterations with each fine solve joining to create one smooth surface.

Figure 3: Results from solving Burger's equation using the parareal algorithm after 1 and then 9 iterations

2 Animating parareal solves

Animation is a good way to help visualise and present the process of solving using parareal. To do this matplotlib's FuncAnimation is used as the back end along with the class _PRanimation to control each frame. _PRanimation is then subclassed with appropriate matplotlib functions for 2d or 3d animation while using the same core code. Freeze frames from an example animation can be seen in Figure 4.

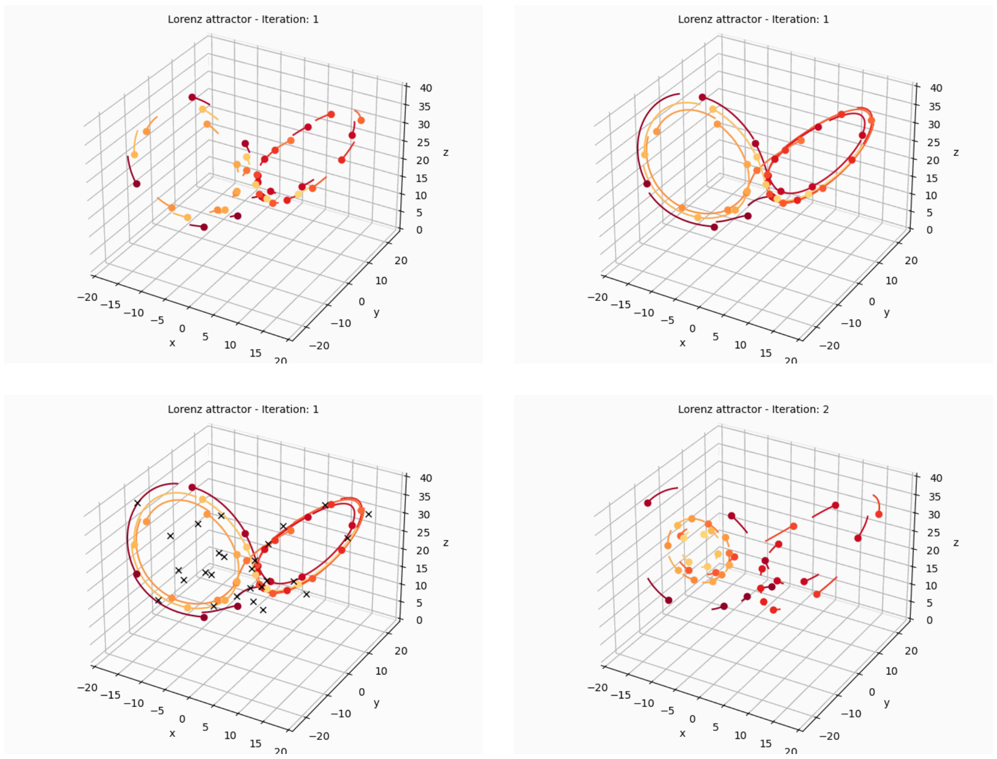


Figure 4: Freeze frames from the animation of the Lorenz system solved with parareal

The animation class is based around the update method which is passed to FuncAnimation. This is called each frame to update various lines and points and returns a list of those edited. Several counters are used to keep track of how far through each stage of the animation the current frame is and to time delays between sections. Once all frames have been animated, update will raise a StopIteration exception which is caught by the animate method, allowing the animation to be saved.

For a parareal solve with a fixed number of fine steps, each frame extends the fine solves by a single step, with the overall speed of the animation controlled via the frame rate. For an adaptive solve, a mixin - _PRanimationAdaptive - replaces this functionality, instead extending the fine solves by a fixed fraction of their solve each frame. The overall time of the fine solve is instead determined by the average number of fine steps across each parallel solve. To achieve this, _PRanimationAdaptive only needs to override iteration_init, using the new method to calculate n_fine_steps. Different animations could be achieved in a similar way by creating alternative mixins. For example, by overriding extend_fine_lines as well as iteration_init appropriately, each fine solve could be animated to progress at a constant rate, with some finishing faster than others.

3 Adaptive time steps for the Brusselator system

Like many other numerical methods, parareal can be improved through the use of an adaptive time-stepping scheme. From Maday and Mula [2], one such adaptive scheme suitable for use with scipy involves slowly increasing the accuracy of the fine solver with each iteration to saving unnecessary computation in the first few iterations. They found that the accuracy of the fine solver ζ_k^N for iteration k and coarse step N should be given by

$$\zeta_k^N = \begin{cases} \varepsilon_{\mathcal{G}}^{1-\frac{k+1}{K}} \left(\frac{\eta}{2}\right)^{\frac{k+1}{K}} & \text{for } k < K \\ \frac{\eta}{2} & \text{for } k \geq K \end{cases}, \tag{8}$$

where $\varepsilon_{\mathcal{G}}$ is related to the accuracy of the coarse solver, η is the desired accuracy of the solve and K is the number of iterations taken by a standard parareal algorithm to reach the accuracy η.

This varying accuracy requirement was implemented using the BaseParareal class discussed in Section 1 using scipy solve_ivp as the fine solver. The solver was given ζ_k^N as the absolute tolerance with the relative tolerance set to 0. However, these values correspond to the time step to time step accuracy rather than across the whole time interval. This can be solved using a mapping as described in [2].

However, more research could not be done with this implementation to replicate the plots in [2] due to the lack of an available high core count computer to compare speed up for different numbers of processors. The code should allow for this to be tested at a later date should the hardware become accessible through the use of the processors parameter in the solve function.

3.1 Adaptivity in coarse stepping

Discussion and testing in this work have so far only considered an algorithm for adaptive fine time steps, however an adaptive algorithm for the coarse time steps could also be considered. In the ideal case, a system solved with N coarse steps is solved with N processors, allowing every fine solve to be done simultaneously. This leads to the motivation that the coarse steps are spaced such that each fine solve is of equal complexity to avoid a bottleneck from one processor. With sufficient prior knowledge of the problem, this could be done by appropriately spacing the coarse steps at initialisation however often this is not possible.

Alternatively, the time values of the coarse steps could be shifted as the solve progresses. However, since the prediction and correction formula relies on the comparison of coarse steps at given times, either some form of interpolation or an adjustment to the formula must be used. As any part of the algorithm related to the coarse solver is run in serial, any such modifications must have sufficiently small costs that they are outweighed by the speed gained from not bottlenecking the fine solves.

4 Different coarse solvers for Burgers equation

Since the coarse solver is the most significant part of the parareal algorithm that must run in serial, it must have as small a cost as possible whilst remaining stable. To this end, three coarse solvers described by Schmitt et al. were investigated to work for Burgers equation (see section 1.3.2) along the lines of the work done in [3]. The descriptions of the methods below are summarised from [3] using centred finite differencing for the spatial discretisation (equation 6).

Explicit Runge-Kutta

$$u' = u^n + \Delta t\left(\nu\frac{\partial^2 u^n}{\partial x^2} + Q^n\right) - \frac{\Delta t}{2}u^n\frac{\partial u^n}{\partial x} \tag{9}$$

$$u^{n+1} = u^n + \Delta t\left(\nu\frac{\partial^2 u'}{\partial x^2} + Q^{n+\frac{1}{2}} - u'\frac{\partial u'}{\partial x}\right) \tag{10}$$

Implicit-explicit Runge-Kutta This is very similar to the explicit version but with equation 9 replaced with a version that is implicit in the diffusive term. This leads to equation 11 which is solved using scipy's root finding functionality fsolve. Equation 10 is kept the same - although now solved with a different value of u'.

$$u' = u^n + \Delta t\left(\nu\frac{\partial^2 u'}{\partial x^2} + Q^{n+\frac{1}{2}}\right) - \frac{\Delta t}{2}u^n\frac{\partial u^n}{\partial x} \tag{11}$$

Semi-Lagrangian This formulation relies on defining a so-called departure point, x_*, for each spatial grid point. This is given by the implicit equation

$$x_* = x - \frac{\Delta t}{2}\left(2u(t_n, x_*) - u(t_{n-1}, x_*) + u(t_n, x)\right), \tag{12}$$

which is again solved using scipy's fsolve, with linear interpolation to calculate the values of u between spatial grid points.

Now defining u_* to be the value of u calculated at the corresponding value of x_* in place of x, u^{n+1} is given by,

$$u^{n+1} = u_*^n + \Delta t\nu\frac{\partial^2 u^{n+1}}{\partial x^2} + \Delta t Q^{n+1}. \tag{13}$$

This is again solved iteratively with cubic interpolation for the values of u_*.

All of these methods include a forcing term, $Q^n = Q(t_n, x)$, to allow for the use of manufactured solutions. To do this Burgers' equation is reformulated as

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2} + Q(t, x),$$

and solving for Q gives

$$Q(t, x) = \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \nu\frac{\partial^2 u}{\partial x^2}. \tag{14}$$

This allows a system to be created with a known, easy to use solution by defining a known u, thus allowing true errors to be calculated when solved with numerical methods.

These methods were first tested using the manufactured solution

$$u(t, x) = \sin(2\pi x)\sin(2\pi t) + \frac{1}{k}\sin(2\pi kx)\sin(2\pi kt), \tag{15}$$

with $Q(t, x)$ calculated from equation 14 using sympy. However, since sympy generated functions cannot be pickled and thus used in pools for parallelisation, a helper function get_funcs_source_code is used to generate the source code for the function that is used instead.

All the tests used $k = 3$ with the ranges $x \in [0, 1], t \in [0, 1]$.

$\Delta t = 10^{-4}$				$\Delta t = 10^{-3}$				$\Delta t = 10^{-2}$			
ν\N	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$	ν\N	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$	ν\N	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$
0	o	o	o	0	o	o	o	0	x	x	x
10^{-4}	o	o	o	10^{-4}	o	o	o	10^{-4}	x	x	x
10^{-3}	o	o	o	10^{-3}	o	o	o	10^{-3}	o	o	x
10^{-2}	o	o	o	10^{-2}	o	o	x	10^{-2}	x	x	x
10^{-1}	o	o	x	10^{-1}	x	x	x	10^{-1}	x	x	x
1	x	x	x	1	x	x	x	1	x	x	x

(a) Stability of the explicit Runge-Kutta method

$\Delta t = 10^{-4}$				$\Delta t = 10^{-3}$				$\Delta t = 10^{-2}$			
ν\N	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$	ν\N	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$	ν\N	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$
0	o	o	o	0	o	o	o	0	x	x	x
10^{-4}	o	o	o	10^{-4}	o	o	o	10^{-4}	x	x	x
10^{-3}	o	o	o	10^{-3}	o	o	o	10^{-3}	o	o	x
10^{-2}	o	o	o	10^{-2}	o	o	o	10^{-2}	o	o	o
10^{-1}	o	o	o	10^{-1}	o	o	o	10^{-1}	o	o	o
1	o	o	o	1	o	o	o	1	o	o	o

(b) Stability of the implicit-explicit Runge-Kutta method

$\Delta t = 10^{-4}$				$\Delta t = 10^{-3}$				$\Delta t = 10^{-2}$			
ν\N	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$	ν\N	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$	ν\N	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$
0	o	o	o	0	o	o	o	0	o	o	o
10^{-4}	o	o	o	10^{-4}	o	o	o	10^{-4}	o	o	o
10^{-3}	o	o	o	10^{-3}	o	o	o	10^{-3}	o	o	o
10^{-2}	o	o	o	10^{-2}	o	o	o	10^{-2}	o	o	o
10^{-1}	o	o	o	10^{-1}	o	o	o	10^{-1}	o	o	o
1	o	o	o	1	o	o	o	1	o	o	o

(c) Stability of the semi-Lagrangian method

Table 1: Stability in of the serial solves for the three time-stepping methods

4.1 Stability of serial solves

First, a standard serial solve was done with each of the 3 methods to help determine their stability. This was done for a range of Δt, Δx and ν with the results shown in table 1. These show similar results to those in [3] except for both RK and IMEX the solve is unstable for $\Delta t = 10^{-2}$, $\Delta x = \frac{1}{64}$ at small ν but also IMEX is always stable for larger values of ν. This is likely to Schmitt et al. using spectral methods to solve the implicit equations and spectral differentiation for the derivatives instead of finite differencing.

Since the explicit Runge-Kutta method shows significant instability at low time steps it will be ignored for the remaining tests as stability at large Δt is essential for a coarse solver.

The semi-Lagrangian method in contrast is stable for all tested values of Δt, Δx and ν. However, looking at table 2a shows that for low values of ν, the time-stepper has significant errors. One considered solution is to replace Q^{n+1} in equation 13 with Q^n giving,

$$u^{n+1} = u_*^n + \Delta t \nu \frac{\partial^2 u^{n+1}}{\partial x^2} + \Delta t Q^n. \tag{16}$$

The errors using this method are shown in table 2b, where it can be seen that this leads to a small improvement for small values of ν but a similarly sized loss in accuracy for larger ν values. The remainder of testing is only done considering the standard formulation (equation 13) but further investigation could be done using equation 16 as a method to improve stability for small ν.

$\Delta t = 10^{-3}$				$\Delta t = 10^{-2}$			
ν\N	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$	ν\N	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$
0	1.1×10^{-1}	4.7×10^{-2}	5.3×10^{-2}	0	4.5×10^{-1}	5.0×10^{-1}	4.6×10^{-1}
10^{-4}	5.3×10^{-2}	4.0×10^{-2}	4.3×10^{-2}	10^{-4}	4.2×10^{-1}	4.4×10^{-1}	4.2×10^{-1}
10^{-3}	5.2×10^{-2}	3.1×10^{-2}	2.5×10^{-2}	10^{-3}	2.7×10^{-1}	2.7×10^{-1}	2.7×10^{-1}
10^{-2}	6.5×10^{-2}	3.5×10^{-2}	1.8×10^{-2}	10^{-2}	1.3×10^{-1}	1.3×10^{-1}	1.3×10^{-1}
10^{-1}	7.2×10^{-2}	3.7×10^{-2}	1.9×10^{-2}	10^{-1}	6.9×10^{-2}	5.5×10^{-2}	5.9×10^{-2}
1	7.4×10^{-2}	3.7×10^{-2}	1.9×10^{-2}	1	7.3×10^{-2}	3.7×10^{-2}	1.9×10^{-2}

(a) Errors when using Q^{n+1}

$\Delta t = 10^{-3}$				$\Delta t = 10^{-2}$			
ν\N	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$	ν\N	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$
0	1.1×10^{-1}	4.5×10^{-2}	5.6×10^{-2}	0	1.8×10^{-1}	1.8×10^{-1}	2.8×10^{-1}
10^{-4}	5.8×10^{-2}	3.5×10^{-2}	3.1×10^{-2}	10^{-4}	1.6×10^{-1}	1.6×10^{-1}	1.6×10^{-1}
10^{-3}	5.2×10^{-2}	3.1×10^{-2}	1.7×10^{-2}	10^{-3}	9.8×10^{-2}	9.9×10^{-2}	9.9×10^{-2}
10^{-2}	6.4×10^{-2}	3.5×10^{-2}	1.8×10^{-2}	10^{-2}	6.3×10^{-2}	5.9×10^{-2}	5.8×10^{-2}
10^{-1}	7.2×10^{-2}	3.7×10^{-2}	1.9×10^{-2}	10^{-1}	1.0×10^{-1}	8.6×10^{-2}	8.5×10^{-2}
1	7.4×10^{-2}	3.7×10^{-2}	2.0×10^{-2}	1	1.3×10^{-1}	1.1×10^{-1}	1.0×10^{-1}

(b) Errors when using Q^n

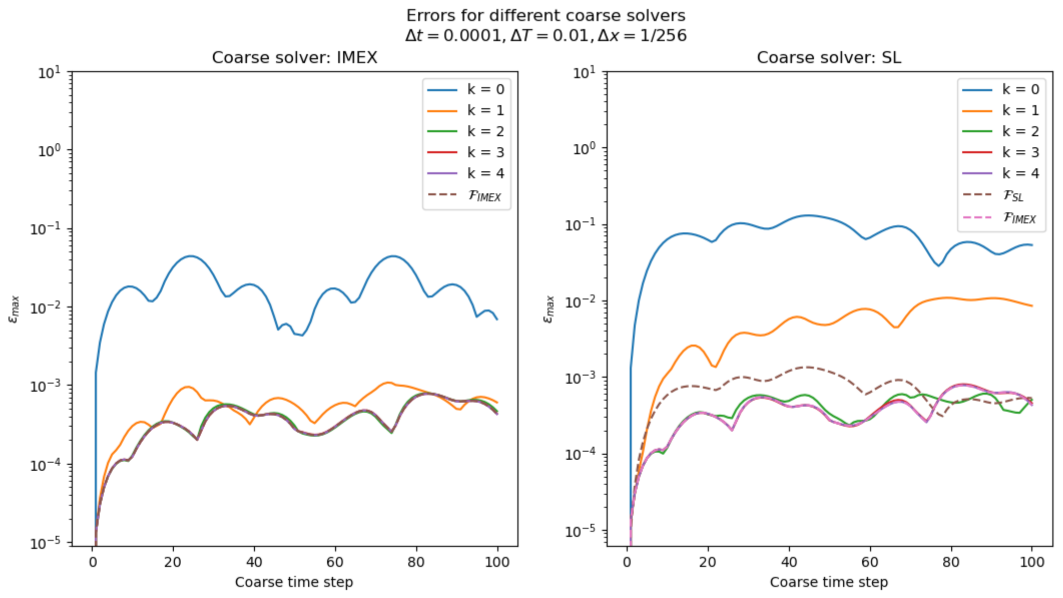
Table 2: Maximum error for the semi-Lagrangian method across both space and time

4.2 Comparison of time-stepping methods for parareal

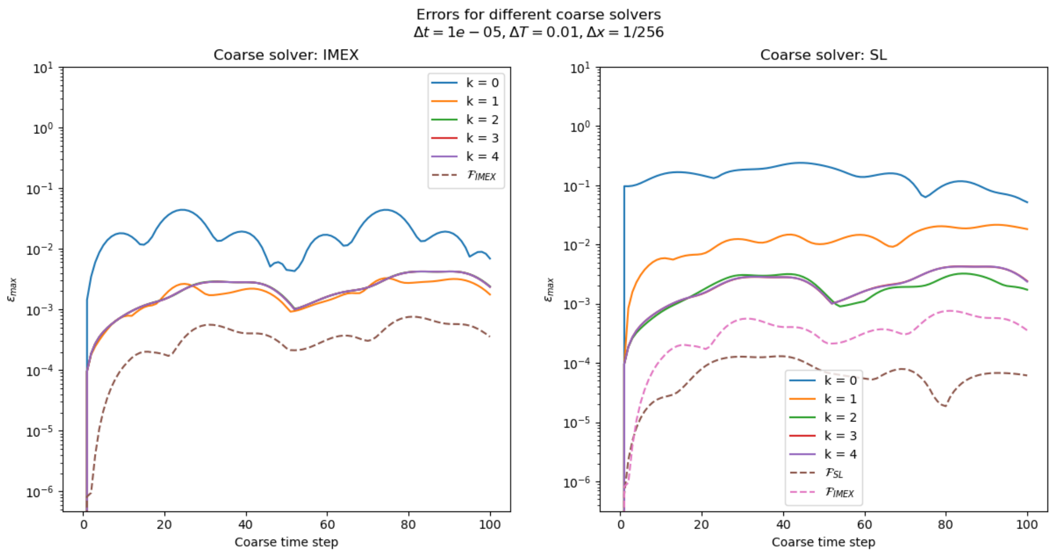
IMEX and SL were then used as different coarse solvers for solving Burgers' equation using parareal. This was done with a coarse time step of $\Delta T = 10^{-2}$, a spatial discretisation of $\Delta x = \frac{1}{256}$, $\nu = 10^{-2}$ and with IMEX as the fine solver. Figure 5 shows the convergence of parareal with each iteration when solved with $\Delta t = 10^{-4}, 10^{-5}$. Figure 5a shows the expected result of the parareal solution converging to the fine solution when solved with $\Delta t = 10^{-4}$. However, this does not seem to be the case in figure 5b for the solve with $\Delta t = 10^{-5}$. I am unsure as to the reasoning so this would need to be explored further.

These figures are also significantly different from those by Schmitt et al. in [3]. This is most significantly due to a larger fine time step ($\Delta t = 10^{-5}$ rather than 10^{-6}) due to hardware limitations making a solve using $\Delta t = 10^{-6}$ take far too long. This is likely an easy problem to fix, however, should in the future suitable hardware be accessible.

One significant difference however is that solving using the IMEX coarse solver was found to be stable. This is not particularly surprising since the parameters of $\Delta T = 10^{-2}$, $\Delta x = \frac{1}{256}$ and $\nu = 10^{-2}$ were found to be close to unstable conditions in [3]. This suggests that in their implementation, the coarse solver was close enough to instability to cause large enough errors to make the parareal solve unstable.



(a) Solve using $\Delta t = 10^{-4}$



(b) Solve using $\Delta t = 10^{-5}$

Figure 5: Errors for each parareal iteration for semi-Lagrangian and IMEX coarse solvers. Solved using $\Delta T = 10^{-2}$, $\Delta x = \frac{1}{256}$ and $\nu = 10^{-2}$.

Another manufactured solution is given by

$$u(t,x) = \frac{1}{2}\sum_{k=1}^{k_{max}} \sin(2\pi kx - \pi kt + \pi k)\frac{\epsilon}{\sinh(\frac{1}{2}\epsilon\pi k)}, \tag{17}$$

which was considered with $k_{max} = 3$ and $\epsilon = 0.1$. The parareal solves were done using $\Delta t = 10^{-5}$, $\Delta T = 10^{-2}$ and $\Delta x = \frac{1}{100}$.

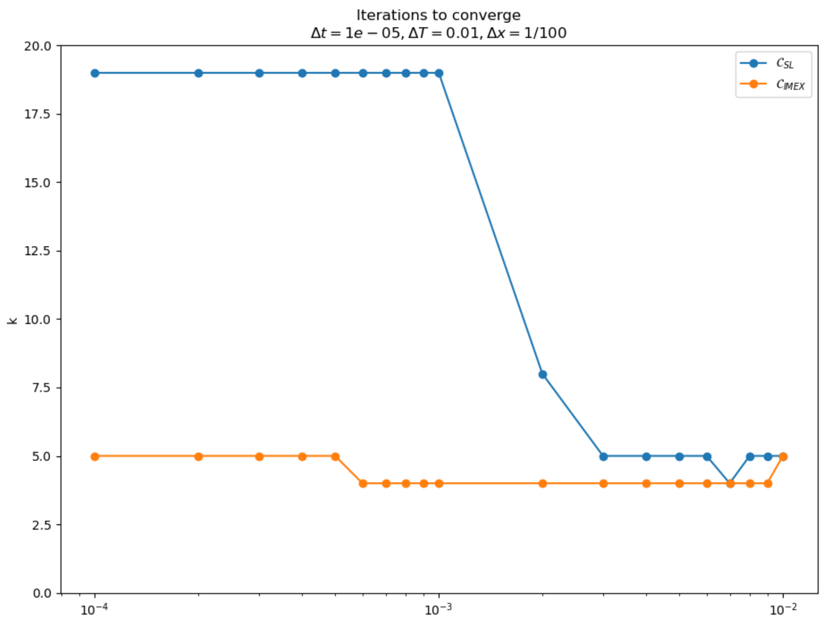


Figure 6: Plot of the number of iterations taken for parareal to reach a tolerance of 10^{-6} for a range of values for ν. The number of iterations was capped at 20, so solves with a reported value of 20 in actuality would likely take longer

Figure 6 shows how long it took for the parareal algorithm to converge to a solution with a tolerance of 10^{-6}. From this plot, it can be seen that for these spatial and temporal steps, IMEX is a much better coarse solver since it converges quickly for all values of ν. In comparison, for $\nu \lessapprox 0.002$, the semi-Lagrangian scheme fails to converge within 20 iterations. This is the reverse of what was found in [3] suggesting that either the implementation of semi-Lagrangian is incorrect or requires the use of spectral methods to prevent instabilities.

The lack of convergence for the semi-Lagrangian scheme can be further seen in figure 7 showing the maximum difference between each parareal difference. ε_{max} should tend to smaller values as each iteration converges on a given solution - even if the converged solution is incorrect. However, instead the SL solver plateaus for $\nu = 0.001$ and even starts to diverge for $\nu \leq 0.0001$.

Instabilities at low ν indicate that these must be caused by the advective term rather than the diffusive term (which is proportional to ν). This means that using spectral methods to solve equation 13 is unlikely to fix the instabilities since this tends to a straightforward, explicit equation for $\nu \to 0$. Instead, it is proposed that the linear and cubic interpolation used in the solve is of insufficient order. This could be checked by replacing the interpolation functions with scipy's UnivariateSpline or similar which supports up to fifth-order interpolation.

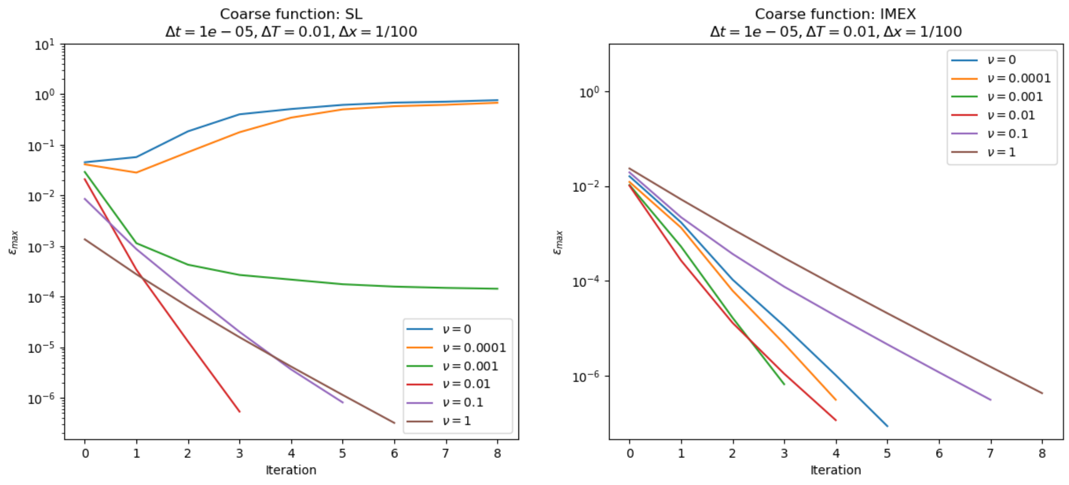


Figure 7: Plot of the progress of the parareal solve after each iteration for varying values of ν. ε_{max} indicates the maximum difference between two iterations of the parareal algorithm. For the SL coarse solver, parareal fails to converge for small values of ν.

References

[1] Martin J. Gander and Ernst Hairer. “Nonlinear Convergence Analysis for the Parareal Algorithm”. In: *Lecture Notes in Computational Science and Engineering* 60 (2008), pp. 45–56. DOI: `10.1007/978-3-540-75199-1{\_}4`. URL: `https://link.springer.com/chapter/10.1007/978-3-540-75199-1_4`.

[2] Y. Maday and O. Mula. “An adaptive parareal algorithm”. In: *Journal of Computational and Applied Mathematics* 377 (Oct. 2020), p. 112915. ISSN: 0377-0427. DOI: `10.1016/J.CAM.2020.112915`.

[3] A. Schmitt et al. “A numerical study of a semi-Lagrangian Parareal method applied to the viscous Burgers equation”. In: *Computing and Visualization in Science 2018 19:1* 19.1 (June 2018), pp. 45–57. ISSN: 1433-0369. DOI: `10.1007/S00791-018-0294-1`. URL: `https://link.springer.com/article/10.1007/s00791-018-0294-1`.

Appendices

A Parareal class internal workings

The parareal class contains several private methods dealing with different stages of the parareal solve or its parallelisation. These are indicated by a leading _ informing the user that they should not be calling these methods. This is a description of their functions.

_print The _print method allows for easy control of whether any print statements within the class to describe progress will be displayed. If print_name has a value then this is prepended to whatever else is to be displayed. If print_name is None then nothing is printed.

_coarse_correction This handles the third step of the parareal algorithm. Individual coarse steps are calculated using the coarse solver which are then corrected using the result from the fine solve as described in section 2.5 of Maday and Mula [2].

_do_fine_integ Fetches the initial conditions (unless already given - see below) and the time range to be passed to the user-provided fine solver.

A.1 Methods for managing parallelisation

The solving process is parallelised through a pool of processes provided by the subprocess module. These processes then work in parallel through each of the fine solves by calling the starmap function with a generator of each set of arguments - the iteration, coarse step number and the initial condition for that step.

The creation of the pool is managed by the solve function while the main solving code is within _solve instead. This means that even if any errors occur within _solve, the solve method can properly exit the pool before the program stops.

Importantly, however, if a standard method of BaseParareal is used as the parallelised function, the whole object will be passed to each process every call. This creates significant overhead that effectively restricts the solve to being done in serial. To avoid this when the pool is created, each process is passed an instance of the parareal object at initialisation using _child_process_init. A static method - _parallel_integ - is then called instead. However, because each process now has its own separate copy of the parareal object, data like the initial conditions must manually be passed into the method, rather than being accessed through self.