The parareal algorithm on the model for combustion of methane

by

L. van der Linden

to obtain the degree of Bachelor of Science in Applied Mathematics
at the Delft University of Technology

Student number:	4702697	
Project duration:	April 20, 2020 – July 10, 2020	
Thesis committee:	Dr. D. J. P. Lahaye,	TU Delft, supervisor
	Dr. J. L. A. Dubbeldam,	TU Delft
	Prof. dr. ir. C. Vuik,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

In this work, the parareal algorithm is analysed and executed on the model for combustion of methane. The parareal algorithm is designed to generate an approximation to an initial value problem faster than a serial numerical time-integration method by using two propagators, the coarse propagator and the fine propagator. With the use of two different propagators, some computations can be carried out in parallel, which leads to a faster method. In this research, the parareal algorithm is executed on the two-step mechanism for combustion of methane. The temperature rise due to the combustion is assumed to be zero. The model is implemented in Python and with the use of the library multiprocessing, computations are executed in parallel. Different time-integration methods are implemented that can be used in the coarse and fine propagator. In this research, we will focus on the case that the both propagators use the same time-integration method. One can distinguish the propagators by using a different time-step for a chosen time-integration method.

With the use of the absolute error the accuracy can be examined. Because the analytic solution to the problem for combustion of methane is unknown, a time-integration method, from which we know that it gives a small absolute error, is used as representation of the analytic solution. The parareal algorithm executed on the model for combustion of methane gives an accurate result for the right choice of propagators. However, for this choice of propagators, the parareal algorithm does not result in a significant speedup compared to the fine propagator in serial, assuming that we have enough processors available. This is because the running time of the propagators do not differ much. To generate a better speedup, two different time-integration methods can be considered for the propagators. Moreover, the model for combustion of methane can be divided into more than two partial reaction and then the multi-level parallelization [1] can be examined.

Acknowledgements

I would like to express my special thanks to my supervisor Dr. Lahaye to introduce me to this fascinating algorithm. With the help of him, I explored the numerous properties of parareal. Even in these turbulent times, I enjoyed the virtual meetings with Dr. Lahaye and I hope to meet him in person soon. I would also like to thank Prof. Dr. Sebastian Schöps for his feedback on my report.

Besides my supervisor I would like to thank the rest of my thesis committee: Dr. Dubbeldam and Prof.dr.ir. Vuik for taking the time to read an review the report.

L. van der Linden
Delft, July 2020

Contents

List of used symbols

A	Pre-exponential factor (unit varies).
E	Activation energy [J mol^{-1}].
$F(t_{i+1}, t_i, \mathbf{u}_i)$	Fine propagator executed on $\mathbf{u}_i$.
$G(t_{i+1}, t_i, \mathbf{u}_i)$	Coarse propagator executed on $\mathbf{u}_i$.
N	Number of grid points on which the approximation of $\mathbf{x}(t)$ is determined.
R	Universal gas constant [J mol^{-1} K^{-1}].
$\Delta G(t_{i+1}, t_i, \mathbf{u}_i)$	Correction term in the parareal procedure on the i^{th} grid point.
ΔT	Time-step between the grid points on which the approximation of $\mathbf{x}(t)$ is determined.
$\mathbf{u}_i^k$	Approximation for $\mathbf{x}(t)$ on the i^{th} grid point in iteration k of the parareal algorithm.
$\mathbf{u}_i$	Approximation for $\mathbf{x}(t)$ on the i^{th} grid point.
$\mathbf{x}(t)$	Solution to the differential equation.
$\mathbf{x}_0$	Initial value, i.e. value of $\mathbf{x}(t)$ on $t = t_0$.
$\mathscr{F}(\Delta t, t_i, \mathbf{u}_i, \mathbf{u}_{i+1})$	Update term in one step of a single-step time-integration method executed on $\mathbf{u}_i$.
T	Temperature [K].
b	Temperature exponent (no unit).
U^k	Approximation of $\mathbf{x}(t)$ on all the grid points in iteration k of the parareal algorithm.

1 Introduction

In this chapter, the motivation for using the parareal algorithm is explained, the thesis statement is introduced and the structure of the thesis is outlined.

1.1. Motivation

Today, almost every computer has multiple processors. This means multiple tasks can be completed in parallel. For large numerical simulations this seems very useful. However, most numerical approximation methods have to be executed linearly. This means, the ability to compute in parallel can not be utilized. Therefore, research on parallel numerical algorithms, one of which is the parareal algorithm, is expanding.

The parareal algorithm is first proposed by Jacques-Louis Lions, Yvon Maday, and Gabriel Turinici in [9] in 2001. In this note, they introduce a new method to numerically solve a initial value problem. The idea of the algorithm is that you first generate a inaccurate result by a method with low computational effort. After that you use these values to generate a more and more accurate result in an iterative process. This method alters from most well-known numerical methods, since it enables the computer to make parallel computations. By making parallel computations, the result can be generated faster, required that you have enough processors.

The parareal algorithm is ideal for a model that has to be integrated on two different timescales, since the algorithm uses two different propagators. The coarse propagator generates a rough approximation on a grid point using the value in the previous grid point. The fine propagator operates the same, but generates a finer approximation. The combustion of methane is a combination of multiple chemical reactions. The different chemical reactions do not have the same reaction rate and therefore ask for integration on different timescales. The parareal algorithm allows to use numerical methods on multiple timescales in only one algorithm. Hence, in this research, the parareal algorithm is executed on the model for combustion of methane.

Even though the parareal algorithm is recently introduced, it is a subject that is widely studied. However, it is not the first proposed algorithm that solves an evolution equation in a time-parallel manner. The first suggestion of time decomposing is already suggested in 1964 by Nievergelt [12]. After that some parallel numerical methods were introduced such as the multiple shooting method [7]. As previously mentioned, the parareal algorithm was firstly proposed by Lions, Maday and Turinici in [9]. An improved version was introduced by Bal and Maday in 2002 [3]. In [1], the speedup achieved by the parareal algorithm is analysed and improved by considering multi-level parallelization. In [5], the use of overlap in time in parareal is considered, which will sometimes lead to a faster algorithm. Recently, multiple adapted parareal algorithms are presented. One of which is the

adapted parareal algorithm by Maday in 2019, to increase the parallel efficiency [10]. Another adaptation to parareal is presented by Gander et al. in 2019 [6]. In this paper, a new parareal algorithm is presented which can deal with discontinuous right-hand side functions in ordinary differential equations.

1.2. Thesis statement

In this thesis, I will analyse the parareal algorithm and discover its properties and benefits compared to the in serial time-integration methods. Moreover, I will implement the algorithm in Python and I will apply the parareal algorithm on the model of combustion of methane. This gets me to my research question: What are the properties of the parareal algorithm and how do they appear in the model for combustion of methane?

1.3. Thesis outline

In chapter 2 the parareal algorithm is introduced. Next to that, the choice for propagators in the algorithm are specified. At the end of this chapter, properties of the parareal algorithm are shown and proven. In chapter 3, I will focus on the implementation of the parareal algorithm with the use of parallel computing in Python. In chapter 4, this implementation will be tested for two test models. With this test, some properties of the algorithm presented in section 2.5 will be checked. After the parareal algorithm is fully understood and the implementation behaves like expected, the parareal algorithm is executed on the model of combustion of methane in chapter 5. Thereby, it examines the advantages of using the parareal algorithm for this application. At last, in chapter 6, the conclusion is made and ideas for further studies are recommended.

2

Introduction to parareal

In this chapter, the parareal algorithm is introduced. First the general initial value problem is posed. Next, the iterative procedure is presented whereafter the used terms are explained. In the end some properties of the parareal algorithm are mentioned and proven.

2.1. Problem description

In this research, we will consider a first order initial value problem.

$$\begin{aligned} \mathbf{x}'(t) &= \mathbf{f}(t, \mathbf{x}(t)) \\ \mathbf{x}(t_0) &= \mathbf{x}_0 \end{aligned} \tag{2.1}$$

We want to estimate the solution $\mathbf{x}(t)$ on the interval $t_0 \leq t \leq T$. The interval will be split into N even subintervals of width ΔT. The solution will be approximated on the grid points, $t_i = t_0 + i\Delta T$. After executing the parareal algorithm, we have found an approximation $\mathbf{u}_i$ for $i = 1, ..., N$.

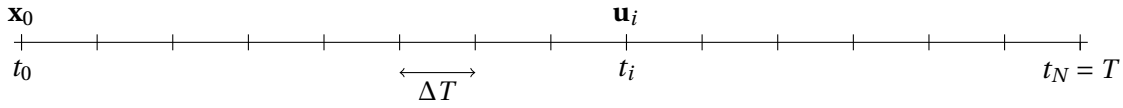


Figure 2.1: Interval over which will be integrated.

2.2. Parareal scheme

The parareal algorithm is an iterative procedure. In every iteration it will update the approximation on all the grid points. The notation for the approximation in iteration k is $U^k = \{\mathbf{u}_i^k : i = 0, ..., N\}$.

We want to find an approximation for the solution to (2.1) on the grid points in the interval $t_0 \leq t \leq T$, with a grid space of ΔT. The parareal algorithm uses a scheme in which it recalculates the approximation in the grid points using an iterative update procedure. In each iteration, it updates the approximation of the previous iteration. This is done by using a fine propagator, $F(t_{i+1}, t_i, \mathbf{u}_i)$, and coarse propagator, $G(t_{i+1}, t_i, \mathbf{u}_i)$. The fine propagator executed on $\mathbf{u}_i$ will give a approximation for $\mathbf{x}(t)$ on $t = t_{i+1}$ to the initial value problem

$$\mathbf{x}'(t) = \mathbf{f}(t, \mathbf{x}(t)), \quad t_i \leq t \leq t_{i+1}, \quad \mathbf{x}(t_i) = \mathbf{u}_i \tag{2.2}$$

The coarse propagator will function the same, but, as the name indicates, will give a less accurate result, though is cheaper to compute than the fine propagator. More on the coarse and fine propagators will be elaborated in section 2.3.

The update scheme is as follows. Given U^k, U^{k+1} is computed as

$$\begin{cases} \mathbf{u}_0^{k+1} = \mathbf{x}_0; \\ \mathbf{u}_{i+1}^{k+1} = G(t_{i+1}, t_i, \mathbf{u}_i^{k+1}) + F(t_{i+1}, t_i, \mathbf{u}_i^k) - G(t_{i+1}, t_i, \mathbf{u}_i^k) \quad \text{for } i = 0, ..., N-1. \end{cases} \tag{2.3}$$

This update scheme calculates $\mathbf{u}_{i+1}^{k+1}$ by executing the coarse propagator on $\mathbf{u}_i^{k+1}$ and adding an correction term $\Delta G(t_{i+1}, t_i, \mathbf{u}_i) := F(t_{i+1}, t_i, \mathbf{u}_i^k) - G(t_{i+1}, t_i, \mathbf{u}_i^k)$ which uses the value $\mathbf{u}_i^k$. The values of $\mathbf{u}_i^k$ from the previous iteration are all known. Therefore, the computation of the correction terms for $i = 0, ..., N-1$ can be calculated at the same time, in parallel. However, the value of $\mathbf{u}_i^{k+1}$ is at the start of iteration $k+1$ only known for $i = 0$, that is why the first term in the update scheme $G(t_{i+1}, t_i, \mathbf{u}_i^{k+1})$ has to be computed sequentially. But this is cheap, since it only requires the coarse propagator.
To start the first iteration of the parareal algorithm, an approximation on all the grid points is required. This initial approximation will be called the zeroth approximation. It will be calculated using the coarse propagator in serial, that is

$$\begin{cases} \mathbf{u}_0^0 = \mathbf{x}_0 \\ \mathbf{u}_{i+1}^0 = G(t_{i+1}, t_i, \mathbf{u}_i^0) \end{cases} \tag{2.4}$$

Subsequently, the solution in the next iterations is calculated using the update scheme. To sum up, the complete parareal algorithm is described in algorithm 1. The algorithm will stop when the stop condition is satisfied. This can either be that the maximum of iterations is reached or that an error constraint is satisfied. In chapter 4 a visualisation of the process of the algorithm is presented using a test equation.

Algorithm 1: Parareal algorithm.

Data: $\mathbf{x}_0$, $\mathbf{f}(\mathbf{x}, t)$
Result: approximation for $\mathbf{x}(t_i)$ in $i = 0, \ldots, N$
begin
  $\mathbf{u}_0^0 \leftarrow \mathbf{x}_0$;
  for $i = 0, \ldots, N-1$ **do**
    $\mathbf{u}_{i+1}^0 \leftarrow G(t_{i+1}, t_i, \mathbf{u}_i^0)$;
  end
  $k \leftarrow 1$;
  while *true* **do**
    solve $F(t_{i+1}, t_i, \mathbf{u}_i^{k-1})$ and $G(t_{i+1}, t_i, \mathbf{u}_i^{k-1})$ for $i = 0, \ldots, N-1$ in parallel;
    $\mathbf{u}_0^k \leftarrow \mathbf{x}_0$;
    for $i = 0, \ldots, N-1$ **do**
      solve $G(t_{i+1}, t_i, \mathbf{u}_i^k)$;
      $\mathbf{u}_{i+1}^k \leftarrow G(t_{i+1}, t_i, \mathbf{u}_i^k) + F(t_{i+1}, t_i, \mathbf{u}_i^{k-1}) - G(t_{i+1}, t_i, \mathbf{u}_i^{k-1})$;
    end
    if *stop condition = true* **then**
      break;
    end
    $k \leftarrow k+1$
  end
end

2.3. Choice of propagators

The parareal scheme uses two propagators, the fine propagator $F(t_{i+1}, t_i, \mathbf{u}_i)$ and the coarse propagator $G(t_{i+1}, t_i, \mathbf{u}_i)$. Both these propagators compute an approximation for $\mathbf{x}(t)$ on $t = t_{i+1}$ using a time-integration method on $\mathbf{u}_i$. The difference is that the fine propagator is more accurate than the coarse propagator, but more expensive to carry out as well. The choice of propagators is based on the numerical stability and the computational effort of the propagator. For each particular model, a suitable coarse and fine propagator has to be selected.

In this section, we will focus on certain choices for the coarse and fine propagators. First, the choice of a single-step time-integration method with different time-steps for the coarse and fine propagator will be treated. After that, the use of the Backward Differentiation Formulas with an order from 1 to 5 using different tolerances for the error will be treated.

2.3.1. Single-step method with different time-steps

As first option, the coarse propagator is taken to be a single-step time-integration method with time-step $\Delta t = \Delta T$, where ΔT corresponds with the time-step of the defined grid over which we want to approximate the solution. The fine propagator is taken to be the same single-step method using time-step $\delta t = \Delta T / m$. Thereby F requires m derivations to be done to determine $\mathbf{u}_{i+1}$ using $\mathbf{u}_i$ but results in a more accurate approximation to $\mathbf{x}(t)$ than G.

A lot of single-step time-integration methods exists, and new single-methods are developed every moment. But in this research, we will only consider the well-known single-step methods: the Forward Euler method, the Backward Euler method, the Trapezoidal method, the Modified Euler method and the Runge-Kutta method. More on these methods can be found in [15].

To generalise for all considered different single-step methods, the notation $\mathscr{F}(\Delta t, t_i, \mathbf{u}_i, \mathbf{u}_{i+1})$ is used: the single-step methods we consider, are composed of the term $\mathbf{u}_i$ and an update term $\mathscr{F}(\Delta t, t_i, \mathbf{u}_i, \mathbf{u}_{i+1})$. This update term is written out for the single-step methods we consider in equation (2.6).

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \mathscr{F}(\Delta t, t_i, \mathbf{u}_i, \mathbf{u}_{i+1}) \tag{2.5}$$

$$\begin{aligned}
\text{Forward Euler method:} \quad & \mathscr{F}(\Delta t, t_i, \mathbf{u}_i, \mathbf{u}_{i+1}) = \Delta t \mathbf{f}(t_i, \mathbf{u}_i) \\
\text{Backward Euler method:} \quad & \mathscr{F}(\Delta t, t_i, \mathbf{u}_i, \mathbf{u}_{i+1}) = \Delta t \mathbf{f}(t_{i+1}, \mathbf{u}_{i+1}) \\
\text{Trapezoidal method:} \quad & \mathscr{F}(\Delta t, t_i, \mathbf{u}_i, \mathbf{u}_{i+1}) = \frac{\Delta t}{2}(\mathbf{f}(t_i, \mathbf{u}_i) + \mathbf{f}(t_{i+1}, \mathbf{u}_{i+1})) \\
\text{Modified Euler method:} \quad & \mathscr{F}(\Delta t, t_i, \mathbf{u}_i, \mathbf{u}_{i+1}) = \frac{\Delta t}{2}(\mathbf{f}(t_i, \mathbf{u}_i) + \mathbf{f}(t_{i+1}, \tilde{\mathbf{u}}_{i+1})) \\
& \text{where } \tilde{\mathbf{u}}_{i+1} = \mathbf{u}_i + \Delta t \mathbf{f}(t_i, \mathbf{u}_i) \\
\text{Runge-Kutta method:} \quad & \mathscr{F}(\Delta t, t_i, \mathbf{u}_i, \mathbf{u}_{i+1}) = \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4) \\
& \text{where } \mathbf{k}_1 = \Delta t \mathbf{f}(t_i, \mathbf{u}_i) \\
& \qquad \mathbf{k}_2 = \Delta t \mathbf{f}(t_i + \frac{\Delta t}{2}, \mathbf{u}_i + \frac{\mathbf{k}_1}{2}) \\
& \qquad \mathbf{k}_3 = \Delta t \mathbf{f}(t_i + \frac{\Delta t}{2}, \mathbf{u}_i + \frac{\mathbf{k}_2}{2}) \\
& \qquad \mathbf{k}_4 = \Delta t \mathbf{f}(t_i + \Delta t, \mathbf{u}_i + \mathbf{k}_3)
\end{aligned} \tag{2.6}$$

The schemes for the coarse and fine propagator are written out in equation (2.7) and (2.8). $\mathbf{u}_{i+\frac{j}{m}}$ is an approximation for $\mathbf{x}(t)$ on $t_{i+\frac{j}{m}} = t_i + j \cdot \delta t$. The single-step method will be executed until an approximation on t_{i+1} is reached. Then the resulting value for $\mathbf{u}_{i+1}$ is assigned to $G(t_{i+1}, t_i, \mathbf{u}_i)$ and

$F(t_{i+1}, t_i, \mathbf{u}_i)$.

$$\begin{aligned} \text{solve for } \mathbf{u}_{i+1}\text{:} \quad & \mathbf{u}_{i+1} = \mathbf{u}_i + \mathscr{F}(\Delta t, t_i, \mathbf{u}_i, \mathbf{u}_{i+1}) \\ & G(t_{i+1}, t_i, u_i) = \mathbf{u}_{i+1} \end{aligned} \tag{2.7}$$

$$\begin{aligned} \text{solve for } \mathbf{u}_{i+\frac{1}{m}}\text{:} \quad & \mathbf{u}_{i+\frac{1}{m}} = \mathbf{u}_i + \mathscr{F}(\delta t, t_i, \mathbf{u}_i, \mathbf{u}_{i+\frac{1}{m}}) \\ \text{solve for } \mathbf{u}_{i+\frac{2}{m}}\text{:} \quad & \mathbf{u}_{i+\frac{2}{m}} = \mathbf{u}_{i+\frac{1}{m}} + \mathscr{F}(\delta t, t_{i+\frac{1}{m}}, \mathbf{u}_{i+\frac{1}{m}}, \mathbf{u}_{i+\frac{2}{m}}) \\ & \vdots \\ \text{solve for } \mathbf{u}_{i+1}\text{:} \quad & \mathbf{u}_{i+1} = \mathbf{u}_{i+\frac{m-1}{m}} + \mathscr{F}(\delta t, t_{i+\frac{m-1}{m}}, \mathbf{u}_{i+\frac{m-1}{m}}, \mathbf{u}_{i+1}) \\ & F(t_{i+1}, t_i, u_i) = \mathbf{u}_{i+1} \end{aligned} \tag{2.8}$$

In figure 2.2, one fine and one coarse propagation are executed, with $\Delta t = 2$ and $\delta t = \frac{2}{8} = \frac{1}{4}$. As single-step method, a Forward Euler scheme is used, i.e. $\mathscr{F}(\Delta t, t_i, \mathbf{u}_i, \mathbf{u}_{i+1}) = \Delta t \mathbf{f}(t_i, \mathbf{u}_i)$. The analytic solution is represented by the grey line. You can see that the fine propagation gives a more accurate approximation, it is closer to the analytic solution.

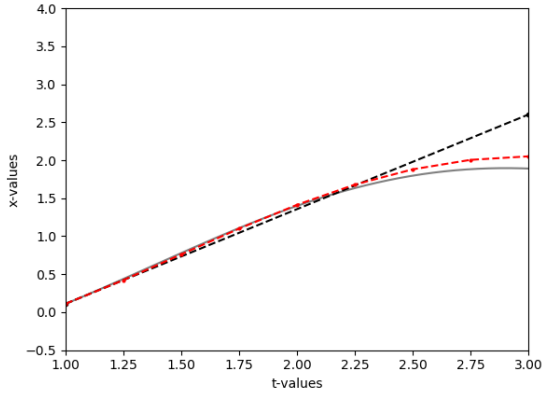


Figure 2.2: One fine propagation executed on $(3, 1, x(1))$ (red) with $\delta t = 1/4$ and one coarse propagation executed on $(3, 1, x(1))$ (black) with $\Delta t = 2$ compared to the analytic solution (grey). IVP:
$x'(t) = sin(\frac{2\pi}{5} t) + \frac{1}{2} sin(\frac{2\pi}{10} t), \quad 1 \leq t \leq 3, \quad x_0 = x(1)$.

2.3.2. Backward Differentiation Formulas with different tolerances

The parareal algorithm will be implemented in Python. Python offers a lot of integration methods and it would be a pity if we did not make use of that. The function `solve_ivp` from the library `scipy.integrate` can numerically integrate a system of ordinary differential equations given an initial value. In our case, the initial value problem we want to solve with `solve_ivp` is described in equation (2.2). We will use the `solve_ivp` function as both the coarse and the fine propagator.

In the function, the initial value problem as in equation (2.2) will be implemented. Also, the integration method used in the function will be defined to be the Backward Differentiation Formulas (BDF) method. This is a family of implicit numerical time-integration methods. The order varies from 1 to 5 and the `solve_ivp` function selects the order automatically based on the case. Also, the number of time-steps taken to find the approximation is automatically selected. Therefore, we could not distinguish the coarse and fine propagator by the time-step. To define a difference between the coarse and the fine propagators, we will make use of the optional input argument for the tolerances. With the input arguments for absolute tolerance and relative tolerance the error estimates are controlled. The lower the tolerances, the better the approximation is. So for the fine propagator low tolerances are set and for the coarse propagator, high tolerances are set.

2.4. Speedup

The Parareal algorithm is developed to generate an approximation in a more time efficient manner by using parallel computations. In this section the speedup of the algorithm is treated. The running time of the algorithm is compared to the running time of executing the fine propagator in serial. The following analysis of the speedup will be performed, provided that we have $\frac{N}{M}$ processors available. We will assume that the communication time between the processors is negligible.

First, we will look at the running time of one coarse propagation $G(t_{i+1}, t_i, \mathbf{u}_i)$ and one fine propagation $F(t_{i+1}, t_i, \mathbf{u}_i)$. In this analysis we will only need a value for the ratio $\frac{\Delta t}{\delta t}$, not an explicit value for the running times of both propagators.

The analysis on the speedup asks for a different approach for either a single-step time-integration method and the BDF method executed by `solve_ivp`. If a single-step method is used for both, the coarse propagator requires to do one step of the single-step method. Therefore we will say that the costs for one coarse propagation is equal to 1. To execute one fine propagation takes more time integration steps, namely $\frac{\Delta t}{\delta t}$. So the costs for one fine propagation is $\frac{\Delta t}{\delta t}$. In the example in figure 2.2, the costs for one coarse propagation is 1, and the costs for the fine propagation is 8 ($=\frac{2}{1/4}$). In the case of using the BDF method, we will set the costs for one coarse propagation to 1. Yet, we can not assign an explicit number to the ratio $\frac{\Delta t}{\delta t}$. However, with using the `solve_ivp` function we can monitor the number of function evaluations made. The more evaluations are carried out, the longer the running time is. To determine the ratio $\frac{\Delta t}{\delta t}$, we can examine the ratio between the function evaluations carried out in one coarse propagation and in one fine propagation. However, this is fluctuating over the time-integration steps, since `solve_ivp` selects an appropriate time-step per integration step based on the particular case. To still be able to use an constant ratio for $\frac{\Delta t}{\delta t}$ over the grid in the following analysis, we will use the mean of this ratio over all the steps. To calculate this mean we will execute the BDF method in the function `solve_ivp` with the tolerances for the coarse propagator in serial over the grid, resulting in the number of function evaluations in each time-step, and do the same for the BDF method with the tolerances for the fine propagator. Then take the ratio of these function evaluations and take the mean of these ratios.

Suppose we have $\frac{N}{M}$ processors available, then $\frac{N}{M}$ computations can be executed in parallel. To make optimal use of these processors, the grid will be partitioned into M sub grids with a grid size of $\tau = \frac{T}{M}$. On these sub grids, the parareal iterations will be executed.

To begin, the costs for executing k iterations on one sub grid are determined. For the zeroth approximation, the coarse propagator is executed in serial over the whole sub grid, this costs $\frac{\tau}{\Delta t}$. In every next iteration, first off the parallel computations are done. In the parallel computations, $\frac{N}{M}$ fine propagators are executed and $\frac{N}{M}$ coarse propagators. The $\frac{N}{M}$ coarse propagations take 1 time-unit, and the $\frac{N}{M}$ fine propagations $\frac{\Delta t}{\delta t}$. Since they are computed at the same time, the highest costs determines the costs for the parallel computations, which is $\frac{\Delta t}{\delta t}$. Following, the update procedure as in equation (2.3) is done where only the coarse propagations have to be calculated in serial. This will cost $\frac{\tau}{\Delta t}$. Altogether, the costs for k iterations of the Parareal algorithm on one sub-grid are

$$\frac{\tau}{\Delta t} + k(\frac{\Delta t}{\delta t} + \frac{\tau}{\Delta t}). \tag{2.9}$$

Thus, the costs for the Parareal algorithm on M sub grids are

$$M(\frac{\tau}{\Delta t} + k(\frac{\Delta t}{\delta t} + \frac{\tau}{\Delta t})). \tag{2.10}$$

The costs for the fine propagator in serial over the entire interval are

$$\frac{T}{\delta t} = M\frac{\tau}{\delta t}. \tag{2.11}$$

The speedup S is the costs for k iterations of the parareal algorithm divided by the costs for the fine propagator in serial. The algorithm is only effective in terms of running time if the costs for

k iterations of parareal are less than the costs for the fine propagator in serial, which is equivalent with saying $S > 1$. Moreover, the speedup is ideal for a high value for S.

$$S = \frac{\frac{\tau}{\delta t}}{\frac{\tau}{\Delta t} + k(\frac{\Delta t}{\delta t} + \frac{\tau}{\Delta t})} = \frac{1}{(k+1)\frac{\delta t}{\Delta t} + k\frac{\Delta t}{\tau}}. \tag{2.12}$$

To calculate the system efficiency, the speedup per processors is considered. We assume that the number of processors is exactly the number of parallel computations that has to be done. This is equal to the number of grid points in one sub grid.

$$P = \frac{\tau}{\Delta t} \tag{2.13}$$

Therefore the system efficiency is

$$E = \frac{S}{P} = \frac{1}{(k+1)\frac{\tau\delta t}{(\Delta t)^2} + k} \tag{2.14}$$

The speedup is ideal if the system efficiency is close to 1. Since the numerator is always larger than k, it is easy to see that this system efficiency is bounded by $\frac{1}{k}$. Thus, the number of iterations that have to be executed to meet the required accuracy, has to be significantly low to achieve speedup.

Assuming $k \geq 0$, the speedup and system efficiency are maximized at $k = 1$ and $\Delta t = \sqrt{2\tau\delta t}$ [14]. Then the speedup and system efficiency are

$$S = \frac{\tau}{2\Delta t}, \quad E = \frac{1}{2}. \tag{2.15}$$

In [1] by Bal, more about the speedup and system efficiency can be found. In that paper, also the speedup is increased by using multi-level parallelization. In [11], the speedup is increased by using a pipelined version of Parareal.

Note that the above estimate for speedup and system efficiency is based on an ideal situation. Often, the communication between processors takes a lot of time and this time can dominate the total running time. However, the estimates do give an indication on the time advantage of using the parareal algorithm. In section 5.2, the speedup will be analysed for the case that both the propagators use the BDF method as time-integration method.

2.5. Convergence of the algorithm

The advantage of using the parareal algorithm instead of a time-integration method in serial is the time efficiency. Nonetheless, the parareal algorithm has some properties in terms of convergence. In this chapter, these properties of the parareal algorithm are presented.

2.5.1. Order of accuracy

For the following theorems we assume that the fine propagator is sufficiently accurate, so that we may use the equality $\mathbf{x}(t_{i+1}) = F(t_{i+1}, t_i, \mathbf{x}(t_i))$. In the first publication on parareal [9], the order of accuracy was analysed for a scalar linear case.

$$\frac{dx}{dt} = ax, \quad x(0) = x_0, \quad \text{with } a \in \mathbb{R} \tag{2.16}$$

Theorem 2.5.1. *Let $\Delta t = \frac{T}{N}$, $t_i = i\Delta t$ for $i = 0, 1, ..., N$. Consider equation (2.16). Let $G(t_{i+1}, t_i, u_i^k)$ be the corresponding Backward Euler approximation with time step Δt. Then,*

$$max_{1 \leq i \leq N} |x(t_i) - u_i^k| \leq C_k \Delta t^{k+1}. \tag{2.17}$$

This means that the order of accuracy for the parareal algorithm is $k+1$. In [3], this proposition is extended to the coarse propagator as a time-integration method of order p and the fine propagator as a method with sufficient accuracy. For these choice of propagators, the order of accuracy is $p(k+1)$.

For general systems, some more assumptions have to be added [2][1].

1. The system in (2.1) is stable for $0 \leq t \leq T$ in the sense that
$$||\mathbf{x}(t)|| \leq C||\mathbf{x}_0|| \tag{2.18}$$
where C is a constant independent of $\mathbf{x}_0$ and t.

2. The coarse propagator is Lipschitz continuous in the sense that
$$||G(t_{i+1}, t_i, \mathbf{u}) - G(t_{i+1}, t_i, \mathbf{v})|| \leq (1 + C\Delta t)||\mathbf{u} - \mathbf{v}|| \text{ for } i \in (0, N-1) \tag{2.19}$$
where C is a constant independent of Δt, $\mathbf{u}$ and $\mathbf{v}$.

3. The correction term $\Delta G(t_{i+1}, t_i, \mathbf{u}_i)$ is of order p with Lipschitz regularity in the sense that
$$||\Delta G(t_{i+1}, t_i, \mathbf{u}) - \Delta G(t_{i+1}, t_i, \mathbf{v})|| \leq C(\Delta t)^{p+1}||\mathbf{u} - \mathbf{v}|| \text{ for } i \in (0, N-1) \tag{2.20}$$
This means that the coarse propagator is a discretisation of order p and satisfies a Lipschitz regularity in its initial conditions.

Theorem 2.5.2. *By assuming the above hypotheses, the order of accuracy of the Parareal algorithm in the k^{th} iteration is $p(k+1)$. That is,*
$$||\mathbf{x}(t_N) - \mathbf{u}_N^k|| \leq C(\Delta t)^{p(k+1)}||\mathbf{x}_0|| \tag{2.21}$$

Proof. This theorem will be proven using induction. We will use the notation C for any constant. For the case $k = 0$ it follows from hypothesis 1 and 3. Now suppose it holds for some given $k-1$, then it holds for k:

$$\begin{aligned}
||\mathbf{x}(t_i) - \mathbf{u}_i^k|| &= ||F(t_i, t_{i-1}, \mathbf{x}(t_{i-1})) - G(t_i, t_{i-1}, \mathbf{u}_{i-1}^k) - \Delta G(t_i, t_{i-1}, \mathbf{u}_{i-1}^{k-1})|| \\
&= ||G(t_i, t_{i-1}, \mathbf{x}(t_{i-1})) - G(t_i, t_{i-1}, \mathbf{x}(t_{i-1})) + F(t_i, t_{i-1}, \mathbf{x}(t_{i-1})) - G(t_i, t_{i-1}, \mathbf{u}_{i-1}^k) - \Delta G(t_i, t_{i-1}, \mathbf{u}_{i-1}^{k-1})|| \\
&\leq ||G(t_i, t_{i-1}, \mathbf{x}(t_{i-1})) - G(t_i, t_{i-1}, \mathbf{u}_{i-1}^k)|| + ||\Delta G(t_i, t_{i-1}, \mathbf{x}(t_{i-1})) - \Delta G(t_i, t_{i-1}, \mathbf{u}_{i-1}^{k-1})|| \\
&\leq (1 + C\Delta t)||\mathbf{x}(t_{i-1}) - \mathbf{u}_{i-1}^k|| + C(\Delta t)^{p+1}||\mathbf{x}(t_{i-1}) - \mathbf{u}_{i-1}^{k-1}|| \quad \text{(by hypothesis 2 and 3)} \\
&\leq (1 + C\Delta t)||\mathbf{x}(t_{i-1}) - \mathbf{u}_{i-1}^k|| + C(\Delta t)^{p(k+1)+1}||\mathbf{x}_0|| \quad \text{(by the induction assumption)}
\end{aligned} \tag{2.22}$$

Because $\mathbf{x}(t_0) = \mathbf{u}_0^k$, we can calculate this inequality repeatedly till we get an expression for $i = N$.

$$\begin{aligned}
||\mathbf{x}(t_0) - \mathbf{u}_0^k|| &= 0 \\
||\mathbf{x}(t_1) - \mathbf{u}_1^k|| &\leq C(\Delta t)^{p(k+1)+1}||\mathbf{x}_0|| \\
||\mathbf{x}(t_2) - \mathbf{u}_2^k|| &\leq C(\Delta t)^{p(k+1)+1}||\mathbf{x}_0|| + C(\Delta t)^{p(k+1)+2}||\mathbf{x}_0|| \\
||\mathbf{x}(t_3) - \mathbf{u}_3^k|| &\leq C(\Delta t)^{p(k+1)+1}||\mathbf{x}_0|| + C(\Delta t)^{p(k+1)+2}||\mathbf{x}_0|| + C(\Delta t)^{p(k+1)+3}||\mathbf{x}_0|| \\
&\vdots \\
||\mathbf{x}(t_N) - \mathbf{u}_N^k|| &\leq C||\mathbf{x}_0|| \sum_{j=1}^{N} (\Delta t)^{p(k+1)+j} \\
&= C||\mathbf{x}_0||(\Delta t)^{p(k+1)} \sum_{j=1}^{N} (\Delta t)^j \\
&\leq C(\Delta t)^{p(k+1)}||\mathbf{x}_0|| \quad \text{for sufficiently small } \Delta t.
\end{aligned} \tag{2.23}$$

□

Note that the inequality holds for all i, but it is only stated for $n = N$ in the theorem.

2.5.2. Convergence

The approximation generated by the parareal algorithm will never converge better towards the solution of the problem than the approximation generated by using the fine propagator in serial. This will be proven by considering two cases. In both cases, the assumption is that the coarse and fine propagators are numerically stable for $0 \leq t \leq T$.

Theorem 2.5.3. *When the right-hand side function* $\mathbf{f}$ *in the ordinary differential equation (2.1) does not depend on* $\mathbf{x}$*, the solution in every iteration of the parareal algorithm will be equal in each iteration (except for the zeroth iteration). Moreover, the approximation is equal to*

$$\begin{aligned} \mathbf{u}_0 &= \mathbf{x}_0 \\ \mathbf{u}_{i+1} &= F(t_{i+1}, t_i, \mathbf{u}_i) \quad \textit{for } i = 0, \ldots, N-1. \end{aligned} \tag{2.24}$$

Proof. In this proof we consider the coarse propagator to be a single-step time-integration method with time-step Δt. Every time-integration step is structured like

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \mathscr{F}(\Delta t, t_i, \mathbf{u}_i, \mathbf{u}_{i+1}). \tag{2.25}$$

The $\mathscr{F}(\Delta t, t_i, \mathbf{u}_i, \mathbf{u}_{i+1})$ depends on the method you chose. However, the $\mathbf{u}_i$ and $\mathbf{u}_{i+1}$ in the operator $\mathscr{F}$ only appears inside the function $\mathbf{f}$ and since $\mathbf{f}(t, \mathbf{x})$ does not depend on x, ($\mathbf{f}(t, \mathbf{x}) = \mathbf{f}(t)$), without loss of generality, we can write one time-integration step as

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \mathscr{F}(\Delta t, t_i). \tag{2.26}$$

The coarse propagator is defined as one time-integration step.

$$G(t_{i+1}, t_i, \mathbf{u}_i) = \mathbf{u}_i + \mathscr{F}(\Delta t, t_i) \tag{2.27}$$

The fine propagator is defined as k time-integration steps with time-step $\Delta t / m$.

$$\begin{aligned} \mathbf{u}_{i+\frac{1}{m}} &= \mathbf{u}_i + \mathscr{F}(\Delta t/m, t_i) \\ \mathbf{u}_{i+\frac{2}{m}} &= \mathbf{u}_{i+\frac{1}{m}} + \mathscr{F}(\Delta t/m, t_{i+\frac{1}{m}}) \\ &= \mathbf{u}_i + \mathscr{F}(\Delta t/m, t_i) + \mathscr{F}(\Delta t/m, t_{i+\frac{1}{m}}) \\ \mathbf{u}_{i+\frac{3}{m}} &= \mathbf{u}_{i+\frac{2}{k}} + \mathscr{F}(\Delta t/m, t_{i+\frac{2}{m}}) \\ &= \mathbf{u}_i + \mathscr{F}(\Delta t/m, t_i) + \mathscr{F}(\Delta t/m, t_{i+\frac{1}{k}}) + \mathscr{F}(\Delta t/m, t_{i+\frac{2}{m}}) \\ &\vdots \\ F(t_{i+1}, t_i, \mathbf{u}_i) = \mathbf{u}_{i+1} &= \mathbf{u}_i + \sum_{j=0}^{m-1} \mathscr{F}(\Delta t/m, t_{i+\frac{j}{m}}) \end{aligned} \tag{2.28}$$

If we want to approximate the value of $\mathbf{x}$ on grid point $i+1$ in iteration $k+1$ we get

$$\begin{aligned} \mathbf{u}_{i+1}^{k+1} &= G(t_{i+1}, t_i, \mathbf{u}_i^{k+1}) + F(t_{i+1}, t_i, \mathbf{u}_i^k) - G(t_{i+1}, t_i, \mathbf{u}_i^k) \\ &= \mathbf{u}_i^{k+1} + \mathscr{F}(\Delta t, t_i) + \mathbf{u}_i^k + \sum_{j=0}^{k-1} \mathscr{F}(\Delta t/m, t_{i+\frac{j}{k}}) - \mathbf{u}_i^k - \mathscr{F}(\Delta t, t_i) \\ &= \mathbf{u}_i^{k+1} + \sum_{j=0}^{k-1} \mathscr{F}(\Delta t/m, t_{i+\frac{j}{k}}) \\ \mathbf{u}_{i+1}^{k+1} &= F(t_{i+1}, t_i, \mathbf{u}_i^{k+1}) \end{aligned} \tag{2.29}$$

Since $\mathbf{u}_0^k = \mathbf{x}_0$ for all k, this gives that in each iteration of the parareal algorithm (except for the zeroth one) the same solution is obtained. □

Now we will consider the case that $\mathbf{f}(t,\mathbf{x}(t))$ does depend on $\mathbf{x}$. This gives a more general, but weaker result.

Theorem 2.5.4. *After i iterations of the parareal algorithm, the approximation in the i^{th} grid point will be constant. Moreover, the approximation in this grid point will be*

$$\mathbf{u}_i^k = F(t_i, t_{i-1}, \mathbf{u}_{i-1}^{i-1}) \quad \textit{for } k = i, \ldots, N. \tag{2.30}$$

Proof. This theorem will be proven by induction. G and F are the general coarse and fine propagators.

1. To start the proof we will check it for the case $i = 1$.

$$\begin{aligned}
\mathbf{u}_0^k &= \mathbf{x}_0 \text{ for } k = 0,1,2,... \\
\mathbf{u}_1^{k+1} &= G(t_1, t_0, \mathbf{u}_0^{k+1}) + F(t_1, t_0, \mathbf{u}_0^k) - G(t_1, t_0, \mathbf{u}_0^k) \\
&\text{it holds that } \mathbf{u}_0^{k+1} = \mathbf{u}_0^k = \mathbf{u}_0^0 \text{ for } k = 0,1,2,... \\
\text{So } \mathbf{u}_1^{k+1} &= F(t_1, t_0, \mathbf{u}_0^k) = F(t_1, t_0, \mathbf{u}_0^0) \text{ for } k = 0,1,2,... \\
\Rightarrow \mathbf{u}_1^k &= F(t_2, t_0, \mathbf{u}_0^0) \text{ for } k = 1,2,...
\end{aligned}$$

2. By assuming it holds for $i = n$, we will prove that it holds for $i = n+1$.

$$\begin{aligned}
\mathbf{u}_n^k &= F(t_n, t_{n-1}, \mathbf{u}_{n-1}^{n-1}) \text{ for } k = n, n+1, ... \text{ (assumption)} \\
\mathbf{u}_{n+1}^{k+1} &= G(t_{n+1}, t_n, \mathbf{u}_n^{k+1}) + F(t_{n+1}, t_n, \mathbf{u}_n^k) - G(t_{n+1}, t_n, \mathbf{u}_n^k) \\
&\text{it holds that } \mathbf{u}_n^{k+1} = \mathbf{u}_n^k = \mathbf{u}_n^n \text{ for } k = n, n+1, ... \\
\text{So } \mathbf{u}_{n+1}^{k+1} &= F(t_{n+1}, t_n, \mathbf{u}_n^k) = F(t_{n+1}, t_n, \mathbf{u}_n^n) \text{ for } k = n, n+1, ... \\
\Rightarrow \mathbf{u}_{n+1}^k &= F(t_{n+1}, t_n, \mathbf{u}_n^n) \text{ for } k = n+1, n+2, ...
\end{aligned}$$

□

3

Parallel computing

The parareal algorithm is only worthwhile in terms of speedup if the corrector term can be executed in parallel. In this chapter, the parallel implementation is highlighted with the use of `multiprocessing`.

3.1. Multiprocessing

Python offers different packages for parallel computing, in this paper the package `multiprocessing` is used. `Multiprocessing` seems to work the best for CPU bound processes and is easy to understand and implement. For a simple demonstration by a code, this is the perfect choice. In this section the procedure of parallel computing with `multiprocessing` is presented.

After the package is imported, a `Pool` is created by

```
pool = multiprocessing.Pool(multiprocessing.cpu_count())
```

The argument in the function `Pool` gives an restriction on how many processes can be executed in parallel. In this way, as much computations as there are CPU's can be executed in parallel. Note that this does not imply that you can only assign this number of processes to the `Pool`. For example, if you assign 10 processes to `Pool` which will all take 1 second, and you have 4 CPU's in your computer, the running time is 3 seconds in total (when neglecting the communication time between processors).

To assign processes to the `Pool`, we use

```
result = [pool.apply_async(job,args=(j,)) for j in list]
```

By applying the jobs asynchronously, they can run in parallel. If only `apply` is used, the `Pool` blocks until the result of one job is ready, after that it will start executing the next job. This is therefore not suitable for parallel computing. `apply_async` can account for this problem, therefore this is used. Though, `apply_async` does ask for one more step. After applying the jobs for the `Pool` asynchronously, the return is a list of objects. To get the result out of the list of objects, `get()` is used in the following way. Note that it is in a linear manner, but it has small computational costs.

```
result = [res.get() for res in result]
```

At the end of the `Pool`, you should close the `Pool` by running

```
pool.close()
```

The complete documentation on `multiprocessing` can be found in [13]. In the following sections, multiprocessing is applied to the specific case treated in this research.

3.1.1. Create the pool

First off, the pool has to be specified. Time-integration steps executed on different grid points have to be assigned to the pool. With the knowledge of a right-hand side function, the Δt, the time on the grid points t, the x (possibly vector) value on the grid points and facultative tolerances, the time-integration step can be assigned to the pool with the use of a specified method.

```
def solver(method,f,dt,t,x,rtol=1e-1,atol=1e-1):
    Dt = t[1]-t[0]
    pool = multiprocessing.Pool(multiprocessing.cpu_count())
    result = [pool.apply_async(method,args=(f,Dt,dt,t[i],x[i],rtol,atol)) for i in range(len(t
        ))]
    result = [res.get() for res in result]
    pool.close()
    result = np.array(result)
    return result
```

The method function is specified as one of the time-integration methods considered in section 2.3. The code for different methods can be found in appendix A. Below, the Forward Euler method propagator is described to get an impression.

```
def forwardeuler(f,Dt,dt,t,x,dummy_1=0,dummy_2=0):
    t_end = t + Dt
    while t < t_end:
        x = x + dt*f(t,x)
        t = t + dt
    return x
```

3.1.2. Define the propagators with the use of a pool

Parallel computing will be used in the computation of the correction term $\Delta G(t_{i+1}, t_i, \mathbf{u}_i)$. The correction term uses the fine propagator and the coarse propagator executed on all the grid points.

In both the propagators, parallel computations have to be made. In the coarse propagator the computations involve one time-integration step over all the grid points. The `rtol_g` and `atol_g` are the tolerances that belong to the coarse propagator.

```
def G(rhs, array):
    t, x = array[:-1,0], array[:-1,1:]
    dt = t[1]-t[0]
    x = solver(method,rhs,dt,t,x,rtol=rtol_g,atol=atol_g)
    t = np.array([t + dt]).T
    return np.hstack((t,x))
```

The fine propagator comes in two variants. If a single-step method is used in the fine propagator, the fine propagator uses a small time-step to propagate. In addition, it carries out more propagations. This is accounted for in the functions for different methods. The method functions propagate until the time on the next grid point is reached in a while loop. For the coarse propagator this is simply after one loop. For the fine propagator, this is after m loops, where $m = \frac{\Delta t}{\delta t}$. The other variant is that the `solve_ivp` function is used in both propagators. For this variant, the `rtol_f` and `atol_f` values are used. In this case, δt is also an argument in the function `solver`, but is not used.

```
def F(rhs, array, division):
    t, x = array[:-1,0], array[:-1,1:]
    dt = (t[1]-t[0])/division
    x = solver(method,rhs,dt,t,x,rtol=rtol_f,atol=atol_f)
    t = np.array([t + dt*division]).T
    return np.hstack((t,x))
```

You can see that the only difference in the functions for the coarse and fine propagator is the time-step and the tolerances.

3.2. Code structure

Parareal is defined as a class. In the class, two functions for the approximation are defined. The function `startvector()` determines the zeroth approximation. The function `iterate(k)` makes use of the external two propagator functions `F` and `G` to execute k iterations of the Parareal algorithm. Both the `startvector()` and `iterate(k)` functions assign an array with the answer to the attribute `self.answer`. Furthermore, functions for plotting the approximation and the error are defined.

3.2.1. Input arguments

First, the initial value problem is described according to the structure in section 2.1 by a right-hand side function $\mathbf{f}(\mathbf{x}(t), t)$. This function is a function from $\mathbb{R}^d \times \mathbb{R}$ to $\mathbb{R}^d$. In Python, this means that it asks for a d-dimensional array and a t value, and then returns a d-dimensional array. Now the parareal algorithm can be executed on this problem.

The Parareal class asks for the following inputs

```
Parareal(rhs, start, end, coarsesteps, division, initialvalue, iterations)
```

The first input is the right-hand side function of the initial value problem. `start` and `end` are the boundaries of the time interval over which will be integrated. The integer `coarsesteps` tells in how many grid points the solution has to be approximated. With these values, also the time-step `Dt` is calculated. The integer `division` indicates in how many sub intervals each interval between grid points has to be divided for the fine propagation. `initialvalue` is the x_0 and `iterations` is the number of iterations of the parareal algorithm that has to be executed. Additionally, the method for time-integration is specified.

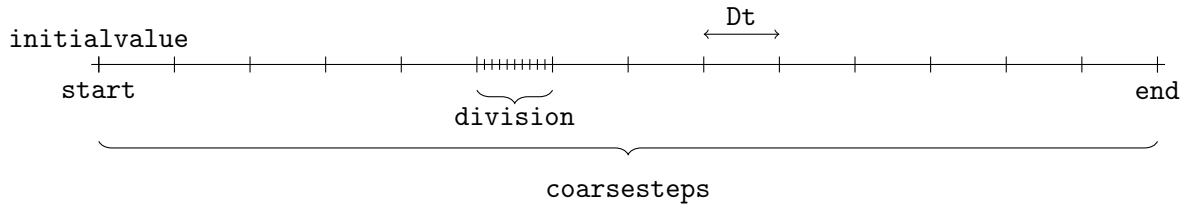


Figure 3.1: Interval over which will be integrated.

3.2.2. Performing the algorithm

In the zeroth approximation, the time-integration method will be performed on all the grid-points in serial by `startvector()`. The resulting `numpy` array is shown below. This array is assigned to the attribute `self.answer`.

$$\begin{array}{llllll} [[t_0, & x_{0,1}, & x_{0,2}, & \ldots, & x_{0,d}], \\ [t_1, & u^0_{1,1}, & u^0_{1,2}, & \ldots, & u^0_{1,d}], \\ [t_2, & u^0_{2,1}, & u^0_{2,2}, & \ldots, & u^0_{2,d}], \\ \vdots & & & & \\ [t_N, & u^0_N, & u^0_{N,2}, & \ldots, & u^0_{N,d}]] \end{array} \tag{3.1}$$

Then k iterations of the parareal algorithm are executed. In an iteration, first the parallel computations are carried out. Note that since `G` and `F` are two distinct functions, they are not executed in parallel. However, in the functions the propagations for each grid point are executed in parallel. With including this, an extra term $k \cdot 1$ has to be added to the numerator of the speedup. Though, the implementation is just used as a tool to execute the algorithm and is not ideal. Therefore, this note will be overlooked.

Then the approximation in the previous iteration will be updated in serial according to the scheme in (2.3). Next, the updated approximation will be added to the attribute `self.answer`. The final `self.answer` includes the approximation after every iteration.

$$
\begin{bmatrix}
[t_0, & x_{0,1}, & x_{0,2}, & \dots, & x_{0,d}], \\
[t_1, & u^0_{1,1}, & u^0_{1,2}, & \dots, & u^0_{1,d}], \\
[t_2, & u^0_{2,1}, & u^0_{2,2}, & \dots, & u^0_{2,d}], \\
\vdots & & & & \\
[t_N, & u^0_N, & u^0_{N,2}, & \dots, & u^0_{N,d}]],
\end{bmatrix}
$$

$$
\begin{bmatrix}
[t_0, & x_{0,1}, & x_{0,2}, & \dots, & x_{0,d}], \\
[t_1, & u^1_{1,1}, & u^1_{1,2}, & \dots, & u^1_{1,d}], \\
[t_2, & u^1_{2,1}, & u^1_{2,2}, & \dots, & u^1_{2,d}], \\
\vdots & & & & \\
[t_N, & u^1_N, & u^1_{N,2}, & \dots, & u^1_{N,d}]], \\
\vdots & & & & \\
[[t_0, & x_{0,1}, & x_{0,2}, & \dots, & x_{0,d}], \\
[t_1, & u^k_{1,1}, & u^k_{1,2}, & \dots, & u^k_{1,d}], \\
[t_2, & u^k_{2,1}, & u^k_{2,2}, & \dots, & u^k_{2,d}], \\
\vdots & & & & \\
[t_N, & u^k_N, & u^k_{N,2}, & \dots, & u^k_{N,d}]]]
\end{bmatrix}
$$

For the full implementation we refer to appendix A.

4

First numerical results

Before the implementation of the parareal algorithm is applied the model of interest in this research, it will be assessed with two one-dimensional examples for the right-hand side function. With these test problems, the algorithm process will become clear and some properties of the algorithm will be substantiated with the results.

4.1. One-dimensional test equations

4.1.1. Right-hand side function independent of the solution

First, we will consider a periodic right-hand side function which does not depend on **x**. The tested one-dimensional initial value problem is

$$\begin{aligned} \frac{dx}{dt}(t) &= f(t) = \sin\frac{2\pi}{5}t + \frac{1}{2}\sin\frac{2\pi}{10}t \qquad 0 \le t \le 10, \\ x(0) &= x_0 = 0. \end{aligned} \tag{4.1}$$

The analytic solution of this differential equation is known. Therefore, we can compare the obtained approximation with the analytic solution. The analytic solution is determined by just integrating the right-hand side with respect to t.

$$x(t) = -\frac{5}{2\pi}(\cos\frac{2\pi}{5}t + \cos\frac{2\pi}{10}t) + \frac{5}{\pi} \tag{4.2}$$

4.1.2. Right-hand side function dependent on the solution

To test the parareal implementation for f dependent on x, we will look at the following one-dimensional differential equation, from which the solution is known.

$$\begin{aligned} \frac{dx}{dt}(t) &= f(x(t),t) = x \cdot t \qquad 0 \le t \le 3, \\ x(0) &= x_0 = \frac{1}{9}. \end{aligned} \tag{4.3}$$

The solution to differential equation is $x(t) = x_0 \cdot e^{\frac{1}{2}t^2}$.

4.2. Parareal visualisation for the first test equation

By visualising the algorithm, we can get a better understanding of the process. We will visualise the algorithm for the first test equation (4.1). For this visualisation we will divide the interval into 8 steps. So there are 9 grid points, where 8 of them have an unknown value for $x(t)$. The propagators use

the Forward Euler method as time-integration method. The coarse propagator uses the same time-step as the time-step in the grid $\Delta t = \Delta T = \frac{T}{8}$. The fine propagator uses the time step of the coarse propagator divided by 6, $\delta t = \frac{\Delta t}{6}$. Therefore, one coarse propagation takes 1 function evaluation and one fine propagation 6 function evaluations.

The algorithm executed on the model in equation (4.1) is visualised in figure 4.3. The blue line indicates the approximation generated by parareal, which consists of points u_i^k. The black dots are the values for $G(t_{i+1}, t_i, u_i^k)$ and the red dots are the (intermediate) values for $F(t_{i+1}, t_i, u_i^k)$. The number of function evaluations is shown at the left corner of each figure. This gives an indication of the running time thus far.

After executing the zeroth iteration of the parareal algorithm we find the approximation shown in figure 4.3b. Thereafter, the parallel computations in the first iteration of the parareal algorithm are executed, which results in figure 4.3c. In figure 4.3e the approximation after one iteration is shown, this is a result of the sum of the coarse and fine propagator on u_i^0 and the coarse propagator on u_i^1 according to the update scheme in (2.3). The approximation after two iterations is shown in figure 4.3h and seems to be equal to the approximation after one iteration. This conjecture corresponds with theorem 2.5.3 and will be checked in section 4.4.1.

4.3. Order of accuracy

N	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
25	$6.045\cdot10^{1}$	$1.645\cdot10^{1}$	$3.051\cdot10^{0}$	$4.161\cdot10^{-1}$	$4.361\cdot10^{-2}$
rate		0.272	0.185	0.136	0.105
50	$8.948\cdot10^{0}$	$1.143\cdot10^{0}$	$1.048\cdot10^{-1}$	$7.392\cdot10^{-3}$	$4.193\cdot10^{-4}$
rate		0.128	0.092	0.071	0.057
100	$1.789\cdot10^{0}$	$0.111\cdot10^{-1}$	$5.031\cdot10^{-3}$	$1.796\cdot10^{-4}$	$5.251\cdot10^{-6}$
rate		0.062	0.045	0.036	0.029
200	$4.029\cdot10^{-1}$	$1.227\cdot10^{-2}$	$2.776\cdot10^{-4}$	$4.977\cdot10^{-6}$	$7.375\cdot10^{-8}$
rate		0.030	0.023	0.018	0.015

Table 4.1

In section 2.5, it is obtained that the parareal algorithm has an accuracy of $\mathcal{O}(\Delta t^{kp})$, where p is the order of accuracy of the coarse propagation scheme and k is the number of iterations executed. This is only true with some restrictions on the coarse propagator and with the knowledge that the fine propagator is sufficiently accurate. In this section this property will be tested for the test equation in (4.3) (Note that this is not interesting to test for test equation in (4.1), since the error does not change after one iteration (2.5.3)).

As coarse propagator we will take the Backward Euler method. This method has an order of $\mathcal{O}(\Delta t)$. So we expect that the order of accuracy in the k^{th} iterations is $\mathcal{O}(\Delta t^k)$.

$$G(t_{i+1}, t_i, u_i) = \frac{u_i}{1 - \Delta t\, t_{i+1}} \tag{4.4}$$

The fine propagator has to be sufficiently accurate and since we know the solution to the problem, we will use this as fine propagator. That is, u_{i+1} is the solution to the following initial value problem at $t = t_{i+1}$.

$$\begin{aligned} &\frac{dx}{dt}(t) = f(x(t), t) = x \cdot t \qquad t_i \le t \le t_{i+1}, \\ &x(t_i) = u_i. \end{aligned} \tag{4.5}$$

The solution is equal to

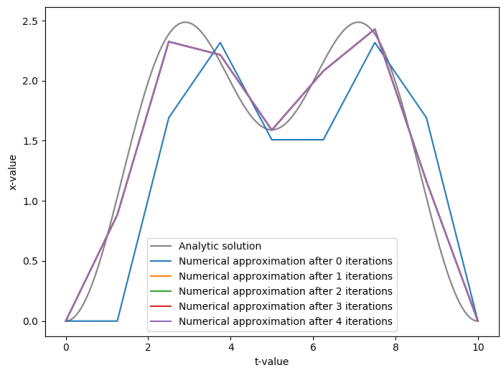
$$x(t) = u_i e^{\frac{1}{2}t^2 - \frac{1}{2}(t_i)^2}. \tag{4.6}$$

First the coarse time-step $\Delta t = \frac{T}{25}$ is used and 5 iterations of parareal are executed. Thereafter, the coarse time-step will be divided by two and 5 iterations are executed. This procedure will be done three times. In table 4.1 the absolute error at the final time is shown for different coarse time-steps and iterations. Next to that, the rate with which the error decreases in the iteration compared to the previous iteration is shown. The statement made before is: the parareal algorithm has an accuracy of $\mathcal{O}(\Delta t^{kp})$, with $p = 1$ in this case. This can also be concluded from table 4.1. The rate with which the error decreases in iteration k compared to iteration $k-1$ has to be proportional to Δt, since $\mathcal{O}(\Delta t^k)/\mathcal{O}(\Delta t^{k-1}) = \mathcal{O}(\Delta t)$. This is also observed in the table: whenever the number of grid points doubles, the rate between the error in iteration k and $k-1$ halves.

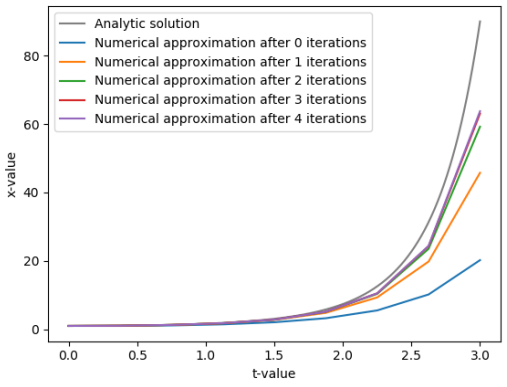
4.4. Examination of convergence

In this section we will use the Forward Euler scheme as time-integration method with for the coarse propagator a time-step of $\Delta t = \frac{T}{8}$ and for the fine propagator a time-step of $\delta t = \frac{\Delta t}{6} = \frac{T}{48}$ for both test equations.

The parareal algorithm is based on a iterative process. In each iteration, the approximation of the solution to the differential equation is updated. Therefore, the approximation in an iteration is expected to be closer to the analytic solution than the approximation in the previous iteration. This is brought to test for the two test equations 4.1 and 4.3. The resulting approximation after different iterations is shown in figure 4.1. In figure 4.1b, you can see that the more iterations of the parareal



(a) Parareal algorithm executed on differential equation 4.1.



(b) Parareal algorithm executed on differential equation 4.3.

Figure 4.1: Approximation by parareal after different iterations.

algorithm you execute, the closer the approximation will get to the analytic solution in the case of the second test equation. Figure 4.1a however, suggests that after 1 iteration, the approximation generated by the parareal algorithm will not improve for the first test equation (which is consistent with theorem 2.5.3). This assumption will be examined by analysing the difference in the approximations for each iteration.

4.4.1. Difference between iterations

Theorem 2.5.3 and figure 4.1a give us the assumption that the approximation for the solution to equation (4.1) will not improve after the first iteration of the parareal algorithm. This will be verified by calculating the difference between the approximation in the iterations. Table 4.2 shows the results. After one iteration, the approximation will not differ, this agrees with the statement we made in theorem 2.5.3.

In theorem 2.5.4, a statement about the parareal algorithm on a general initial value problem is proven. This will be verified by the same examination above. Table 4.3 shows the results. The results

		grid point								
		0	1	2	3	4	5	6	7	8
iteration	1	0.	0.883	0.633	-0.102	0.08	0.572	0.112	-0.527	0.
	2	0.	0.	0.	0.	0.	0.	0.	0.	0.
	3	0.	0.	0.	0.	0.	0.	0.	0.	0.
	4	0.	0.	0.	0.	0.	0.	0.	0.	0.
	5	0.	0.	0.	0.	0.	0.	0.	0.	0.

Table 4.2: Difference between the iterations in the parareal algorithm executed on problem 4.1.

are consistent with the theorem.

		grid point								
		0	1	2	3	4	5	6	7	8
iteration	1	0.	0.05990422	0.14405714	0.31043192	0.68327086	1.57277914	3.79929763	9.62491861	25.53591463
	2	0.	0.	0.00453648	0.0217079	0.08224442	0.29461505	1.04461981	3.72610623	13.47027277
	3	0.	0.	0.	0.00050056	0.00430454	0.02672006	0.14719564	0.76632767	3.87299929
	4	0.	0.	0.	0.	0.00008285	0.00117591	0.0112253	0.09043722	0.66324957
	5	0.	0.	0.	0.	0.	0.00002014	0.00044017	0.00613007	0.06925802
	6	0.	0.	0.	0.	0.	0.	0.00000695	0.00022137	0.00430924
	7	0.	0.	0.	0.	0.	0.	0.	0.00000329	0.00014629
	8	0.	0.	0.	0.	0.	0.	0.	0.	0.00000208
	9	0.	0.	0.	0.	0.	0.	0.	0.	0.

Table 4.3: Difference between the iterations in the parareal algorithm executed on problem 4.3.

4.4.2. Error after different iterations

To examine whether the equality in theorem 2.5.4 is applicable, the absolute error in different iterations is analysed. In addition, the serial Forward Euler scheme is executed on the model with the time step used for the fine propagator. To give a clear visualisation, after the error in each grid point is generated, the error is divided by the error of the serial Forward Euler scheme in that grid point. Thus, when the scaled error reaches 1, the error using parareal is equal to the error using the serial Forward Euler scheme. The iteration is plotted against the error in logarithmic scale in figure 4.2. From the figure, it can be noticed that in the end, the scaled errors in each grid point will be one.

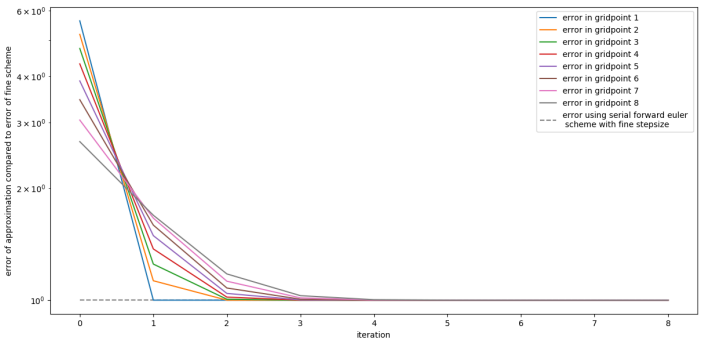
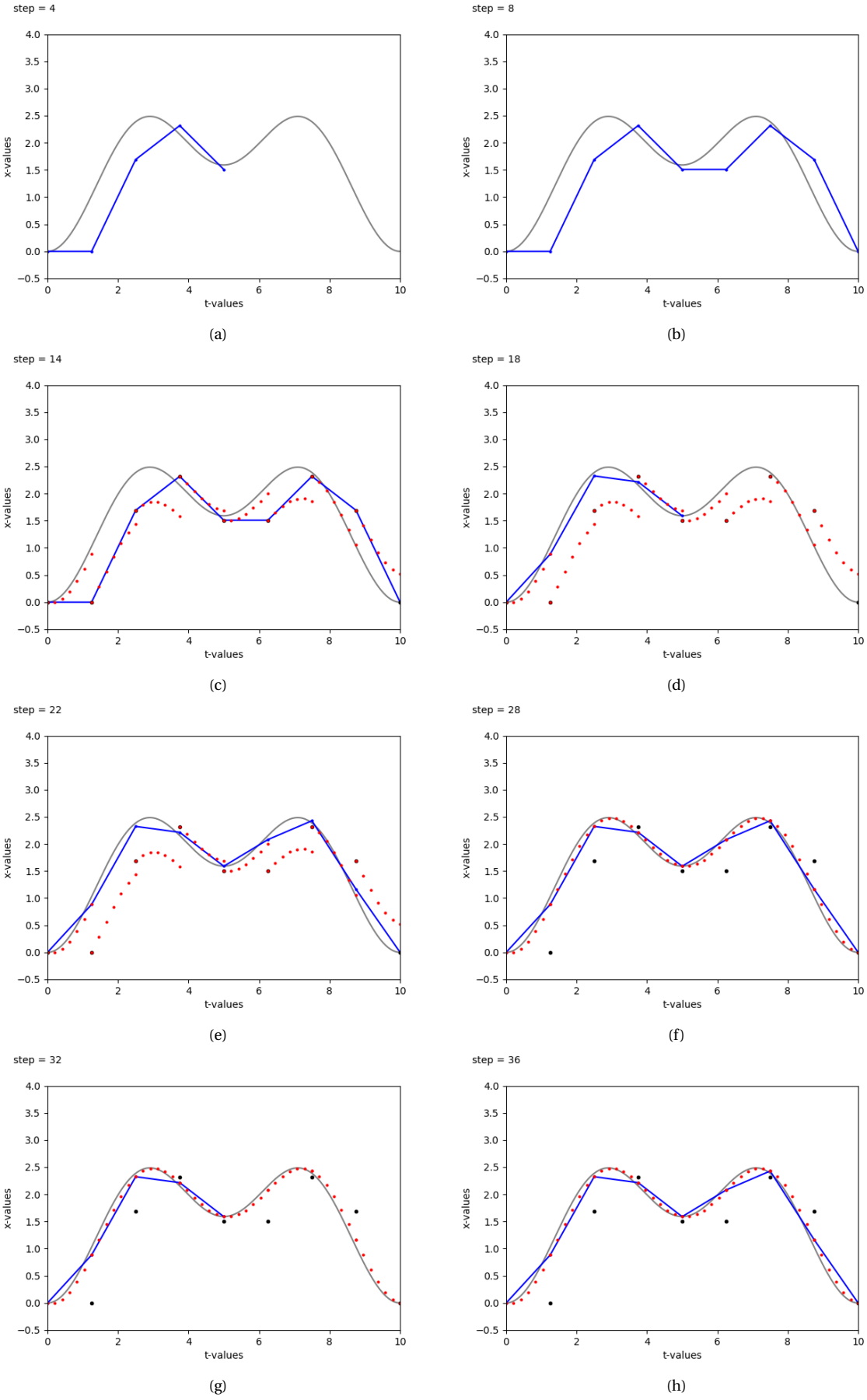


Figure 4.2: Scaled error in the grid points after different iterations compared to the error gained using the Forward Euler scheme in serial.

That means the errors in each grid point will be the same as the error for the serial Forward Euler scheme.



(a) (b) (c) (d) (e) (f) (g) (h)

Figure 4.3: Visualisation of 2 iterations in the parareal algorithm procedure on equation (4.1).

5

Application: combustion of methane

In this chapter, the parareal algorithm is executed on the model for the combustion of methane. First, the overall reaction will be introduced to get an idea of the reaction process. After that, the total reaction will be divided into two steps and here the functionality of the parareal algorithm will come forward.

5.1. One-step mechanism

First, we will model the total reaction. This, we will identify as the one-step mechanism of combustion of methane. The parareal scheme will not give a remarkable advantage for this model. This section is merely to give an impression of the reaction process.

5.1.1. Model description

We will study the reaction mechanism of combustion of methane. The combustion of methane is composed of more partial reactions and the total reaction equation is (5.1). When one methane molecule meets two oxygen molecules, they start to react and after different small reactions, one molecule carbon dioxide and two molecules hydrogen are formed [17].

$$CH_4 + 2\,O_2 \xrightarrow{k} CO_2 + 2\,H_2O \tag{5.1}$$

The k represents the rate coefficient of the reaction and gives an indication on how fast the reaction develops. In our model, it is expressed using the modified Arrhenius equation [8].

$$k = A\mathrm{T}^b \cdot e^{-\frac{E}{R\mathrm{T}}}, \tag{5.2}$$

where A is the pre-exponential factor, b is the temperature exponent, T is the temperature in Kelvin, E is the activation energy and R is the gas constant. The pre-exponential factor, the temperature exponent and the activation energy are different for each reaction.

To convert this reaction into a model of differential equations, the rate law is used. This is described in [16, Chapter 6]. The rate law results in the reaction rate in equation (5.3), p is the notation for the reaction order.

$$r = [CH_4]^{p_{CH_4}}\,[O_2]^{p_{O_2}} \tag{5.3}$$

For reaction (5.1) the rate equation for the different components in methane combustion is

shown in equation (5.4).

$$\begin{aligned} \frac{d[CH_4]}{dt} &= -k \cdot r \\ \frac{d[O_2]}{dt} &= -2k \cdot r \\ \frac{d[CO_2]}{dt} &= k \cdot r \\ \frac{d[H_2O]}{dt} &= 2k \cdot r \end{aligned} \tag{5.4}$$

This is a first order initial value problem in four dimensions of the form as in equation (2.1). Therefore, it rests us to numerically approximate the solution using the parareal algorithm.

5.1.2. Numerical results

A Forward Euler scheme with different time-steps is used for the coarse and fine propagators. The reaction order used in the implementation is 1 and 0.5 for CH_4 and O_2 respectively. The temperature rise caused by the reaction is assumed to be zero. More parameters of this reaction mechanism are summarized in table 5.1. In figure 5.1 the result of two parareal iterations with the Forward Euler

	Total reaction
Activation energy (E)	$1.1 \cdot 10^{10}$
Temperature exponent (b)	0
Pre-exponential factor (A)	$2 \cdot 10^4$
Reaction exponents (p)	$p_{CH_4} = 1$
	$p_{O_2} = 0.5$

Table 5.1: Parameters for the total reaction of combustion of methane [17].

scheme is shown. The values in table 5.2 are used. The initial value of the concentrations is taken to be in stoichiometric ratio, that is, there is exactly enough O_2 to let all the CH_4 react. The resulting

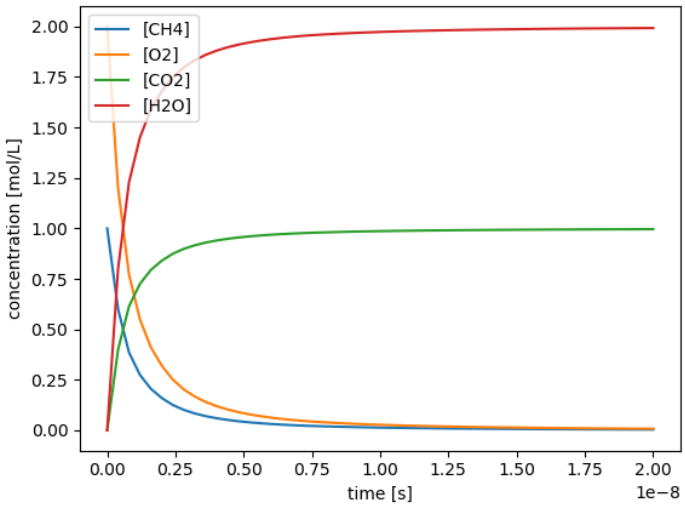


Figure 5.1: One step mechanism with the initial concentrations in stoichiometric ratio.

figure gives an impression of the reaction. Though, in this research, we are interested in the parareal algorithm executed on the two-step mechanism.

Input argument	Value
t_0, t_n	$0,2 \cdot 10^{-8}$
Δt	$\frac{t_n}{50}$
δt	$\frac{\Delta t}{10}$
T	1000 K
$[CH_4]_{t_0}, [O_2]_{t_0}, [CO_2]_{t_0}, [H_2O]_{t_0}$	1,2,0,0 [mol/L]

Table 5.2: Inputs for Parareal for the one-step mechanism.

5.2. Two-step mechanism

5.2.1. Model description

The combustion of methane is a combination of more reactions. In this section, the two-step mechanism of combustion of methane is considered. The CH_4 and O_2 react with each other to CO and H_2O and then the formed CO reacts with the residual O_2 to CO_2 [4].

$$\begin{aligned} CH_4 + 0.5\,O_2 &\xrightarrow{k_1} CO + 2\,H_2O \\ CO + 1.5\,O_2 &\xrightarrow{k_2} CO_2 \end{aligned} \tag{5.5}$$

The reaction coefficients, k_1 and k_2, are expressed using the modified Arrhenius equation (5.2) with different values for the constants A, b and E. With the knowledge of the reaction coefficients and the reaction rates in (5.8), the model for the two-step mechanism of combustion of methane is described in equation (5.6). We want to find an approximation for the solution to this model at $0 \leq t \leq 1 \cdot 10^{-7}$.

$$\begin{aligned} \frac{d[CH_4]}{dt} &= -k_1 \cdot r_1 \\ \frac{d[O_2]}{dt} &= -0.5k_1 \cdot r_1 - 1.5k_2 \cdot r_2 \\ \frac{d[CO]}{dt} &= k_1 \cdot r_1 - 1.5k_2 \cdot r_2 \\ \frac{d[CO_2]}{dt} &= k_2 \cdot r_2 \\ \frac{d[H_2O]}{dt} &= 2k_1 \cdot r_1 \end{aligned} \tag{5.6}$$

$$0 \leq t \leq \cdot 10^{-7} \tag{5.7}$$

$$\begin{aligned} k_1 &= A_1 T^{b_1} \cdot e^{-\frac{E_1}{R_1 T}}, & k_2 &= A_2 T^{b_2} \cdot e^{-\frac{E_1}{R_1 T}} \\ r_1 &= [CH_4]^{p_{CH_4}} [O_2]^{p_{O_2,1}}, & r_2 &= [CO]^{p_{CO}} [O_2]^{p_{O_2,2}} \end{aligned} \tag{5.8}$$

The parameters for the this model are summed up in table 5.3. The second reaction develops faster than the first reaction. To numerically approximate this model, the second reaction calls for integrating over a small timescale and the first reaction for integrating over a large timescale. This can both be achieved by using the parareal algorithm, which will be demonstrated in the following sections.

To illustrate the need for parareal, we will examine what happens if we use the time-integration method Forward Euler, in serial. Take $T = 1 \cdot 10^{-7}$, $\Delta t = \frac{T}{350}$, the result is shown in figure 5.3. In this figure the concentrations for O_2, CO and CO_2 seems to be unstable for the numerical method with the selected time-step. These chemicals are exactly the ones which appear in the second reaction. This suggests that the time-step is incompetent to model the second reaction. However, the result gives the impression that the selected time-step gives a stable result for the first reaction. The result

	first reaction	second reaction
Activation energy (E)	$3.55 \cdot 10^{-4}$	$1.2 \cdot 10^{-4}$
Temperature exponent (b)	0	0.8
Pre-exponential factor (A)	$4.9 \cdot 10^{9}$	$2 \cdot 10^{8}$
Reaction exponents (p)	$p_{CH_4} = 0.5$	$p_{CO} = 1$
	$p_{O_2,1} = 0.65$	$p_{O_2,2} = 0.5$

Table 5.3: Parameters for the 2 step combustion of methane [4].

hereby illustrates the need to integrate over different timescales. To accommodate both the large timescale for the first reaction and the small timescale for the second reaction, the parareal algorithm is used. Note that, another possibility is to integrate over the small timescale on the whole grid, but this leads to an unnecessary small time-step for integrating the first reaction.

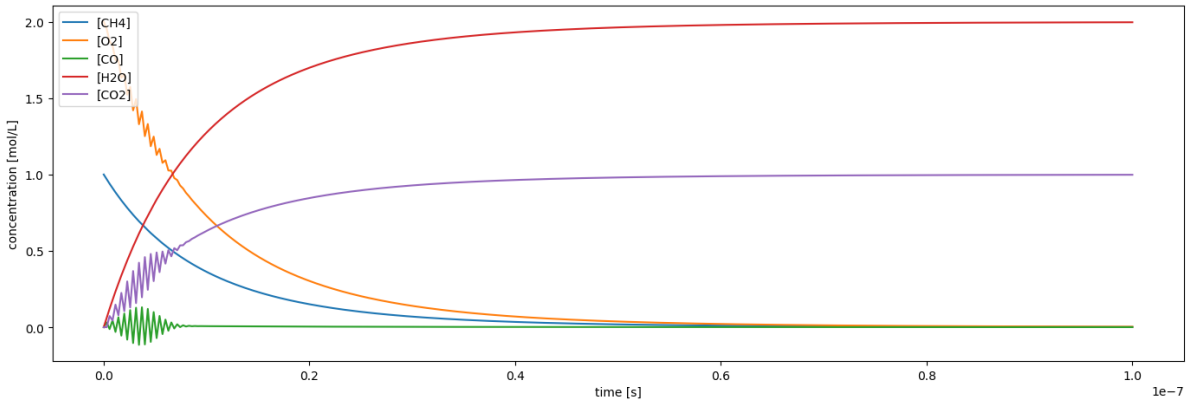


Figure 5.2

Figure 5.3: Two-step mechanism using the Forward Euler method in serial, with $T = 1 \cdot 10^{-7}$, $\Delta t = \frac{T}{350}$.

5.2.2. Selection of parareal parameters for two-step mechanism

To start the parareal algorithm, suitable parameters have to be defined. First, we will select an adequate time-integration method.

It is useful to know that, since the system in (5.6) could be stiff, a convenient choice for the time-integration method is an implicit method. In section 2.3, three implicit methods are considered: the Backward Euler method, the Trapezoidal method and the Backward Differentiation Formulas brought by the `solve_ipv` function. In this research, we will focus on the Backward Differentiation Formulas.

The choice for the time-step used by the coarse and fine propagator is based on the speedup and the system efficiency. For the calculations on speedup and system efficiency, we will refer to section 2.4. We will assume that we have enough processors available, that means $M = 1$, $\tau = T$. The number of iterations that has to be executed is based on the required accuracy. The required accuracy rests on an upper bound for the absolute error. To calculate the absolute error, the analytic solution to (5.6) is necessary. The analytic solution, though, is not known. Therefore we will use the BDF method with a relative tolerance of $3 \cdot 10^{-14}$ and a absolute tolerance of 10^{-20} over the grid as representation of the analytic solution. The absolute error is given as

$$\epsilon = ||\mathbf{x}(t_i) - \mathbf{u}_i^k|| \tag{5.9}$$

First we will define the propagators. In the fine propagator we will use a relative tolerance of $3\cdot10^{-14}$ and an absolute tolerance of $1\cdot10^{-20}$. The coarse propagator will be decided with the use of the speedup and system efficiency. The ratio $\frac{\Delta t}{\delta t}$ in the calculations for speedup and efficiency, is not uniform over the whole grid, therefore we will use the average. To calculate this average, the coarse propagator is executed over a grid with 100 grid points and the number of function evaluations for the calculation of each grid point, is gathered. The same will be done for the fine propagator. The average ratio between the function evaluations executed by the coarse and fine propagator gives an indication of the ratio $\frac{\Delta t}{\delta t}$. For some choices for the tolerances used in the coarse propagator, the estimated ratios are

$$\begin{aligned}
&rtol_F, atol_F = 3\cdot10^{-14}, 1\cdot10^{-20}, \quad rtol_G, atol_G = 1\cdot10^{-10}, 1\cdot10^{-10}, \quad \frac{\Delta t}{\delta t} = 2.3;\\
&rtol_F, atol_F = 3\cdot10^{-14}, 1\cdot10^{-20}, \quad rtol_G, atol_G = 1\cdot10^{-10}, 1\cdot10^{-10}, \quad \frac{\Delta t}{\delta t} = 5.3;\\
&rtol_F, atol_F = 3\cdot10^{-14}, 1\cdot10^{-20}, \quad rtol_G, atol_G = 1\cdot10^{-5}, 1\cdot10^{-5}, \quad \frac{\Delta t}{\delta t} = 11.9;\\
&rtol_F, atol_F = 3\cdot10^{-14}, 1\cdot10^{-20}, \quad rtol_G, atol_G = 1\cdot10^{-1}, 1\cdot10^{-1}, \quad \frac{\Delta t}{\delta t} = 26.0.
\end{aligned} \tag{5.10}$$

In section 2.4, it is stated that the optimal speedup and system efficiency are obtained at $\Delta t = \sqrt{2T\delta t}$, assuming that enough processors are available. This is equivalent with saying that

$$\frac{\Delta t}{\delta t} = 2\frac{T}{\Delta t} \tag{5.11}$$

If we use the tolerances of 10^{-1} for the coarse propagator, we need

$$\frac{T}{\Delta t} = \frac{1}{2}\frac{\Delta t}{\delta t} = \frac{1}{2}\cdot 26 = 13 \tag{5.12}$$

This means, we need 13 grid points to generate the optimal speedup. In this way, the speedup and system efficiency are, after one iteration

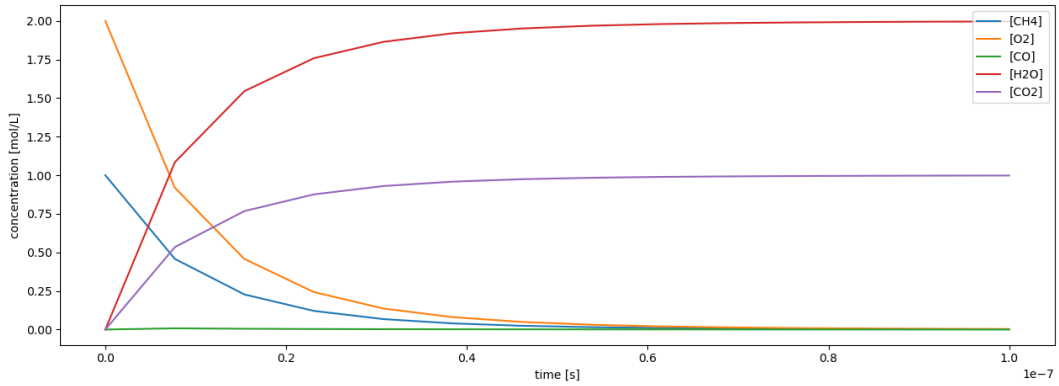
$$S = 6.5, \quad E = 0.5 \tag{5.13}$$

Moreover, the error after one iteration is equal to

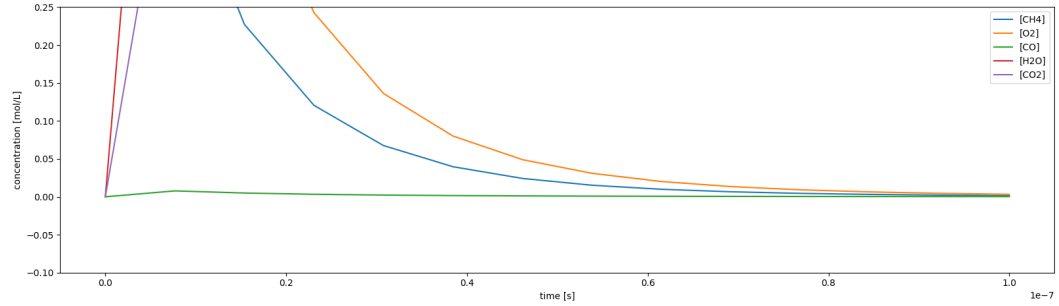
$$\epsilon = 4.27\cdot10^{-4} \tag{5.14}$$

However, the number of grid points is rather small. In this case, even when 4 iterations are executed, the second reaction can not really be observed in the simulation, see figure 5.4. Therefore, in the following simulations we will use 100 grid points. In order to get the optimal speedup and efficiency at $N = 100$, the ratio between the costs for the propagators has to be $\frac{\Delta t}{\delta t} = 200$. This can not be achieved by using the Backward Differentiation Formulas in `solve_ivp` for both the propagators as observed in the cases in equation (5.10). Even for extreme values for the tolerances used by the propagators, the ratio is not even close to 200. To get that large ratio, a different choice for the time-integration method can be used for the coarse propagator. This will not be of interest in this research, but could be examined in further research. Here, we will use the tolerances of 10^{-1} in the coarse propagator, since this gives the best speedup and system efficiency (compared to the other values in equation (5.10)). The resulting values for speedup and system efficiency after one iteration are

$$\frac{\Delta t}{\delta t} = 26.0, \quad \Delta t = 1\cdot10^{-9}, \quad T = 1\cdot10^{-7}, \quad P = 100, \quad k = 2, \quad S = 11.5, \quad E = 0.12. \tag{5.15}$$



(a)



(b) Zoomed

Figure 5.4: Approximation of two-step mechanism by four iterations of parareal in 13 grid points. The tolerances for the fine propagator are $rtol_F, atol_F = 3 \cdot 10^{-14}, 1 \cdot 10^{-20}$ and for the coarse propagator $rtol_G, atol_G = 1 \cdot 10^{-1}, 1 \cdot 10^{-1}$.

Though, the speedup has increased, the system efficiency has decreased much compared to the situation with 13 grid points. However, it gives a slightly more accurate approximation, the error after one iteration is equal to

$$\epsilon = 3.53 \cdot 10^{-5} \tag{5.16}$$

To decide how many iterations of the parareal algorithm have to be executed the error will be analysed. We will restrict the error to be less or equal to 10^{-9} at the final grid point. With 100 grid points and the defined coarse and fine propagators as above, this is satisfied when 5 iterations of the parareal algorithm are executed. In this case, the error, speedup and system efficiency are

$$\frac{\Delta t}{\delta t} = 26.0, \quad \Delta t = 1 \cdot 10^{-9}, \quad T = 1 \cdot 10^{-7}, \quad P = 100, \quad k = 5, \quad \epsilon = 4.95 \cdot 10^{-11} \quad S = 3.56, \quad E = 0.04. \tag{5.17}$$

The error in various iterations for these choices of parameters is displayed in figure 5.5. It is shown that in iteration 5, the error at the final grid-point is less than 10^{-9}. The error is calculated by using the Backward Differentiation Formulas method with order 1 to 5 and relative and absolute tolerances of 10^{-14} and 10^{-20} respectively, as analytic solution. This is the reason for the error to be zero at the first k grid points, when k iterations are executed (a result from theorem 2.5.4).

5.2.3. Numerical results

Altogether, the inputs for parareal are summarized in table 5.4. We will execute the parareal algorithm with 5 iterations. The approximation of the model in equation (5.6) will be calculated on 100 grid points in the interval between 0 and 10^{-7} seconds. The coarse propagator uses the time-integration method BDF in `solve_ivp` with the absolute and relative tolerances of 10^{-1}. The fine propagator uses the time-integration method BDF in `solve_ivp` with relative tolerance of 10^{-14}

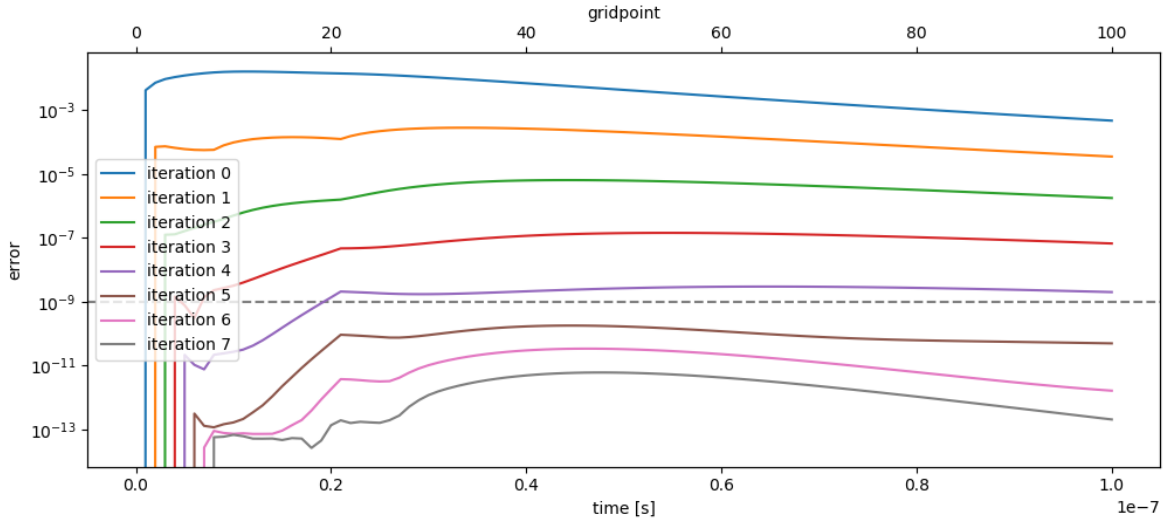


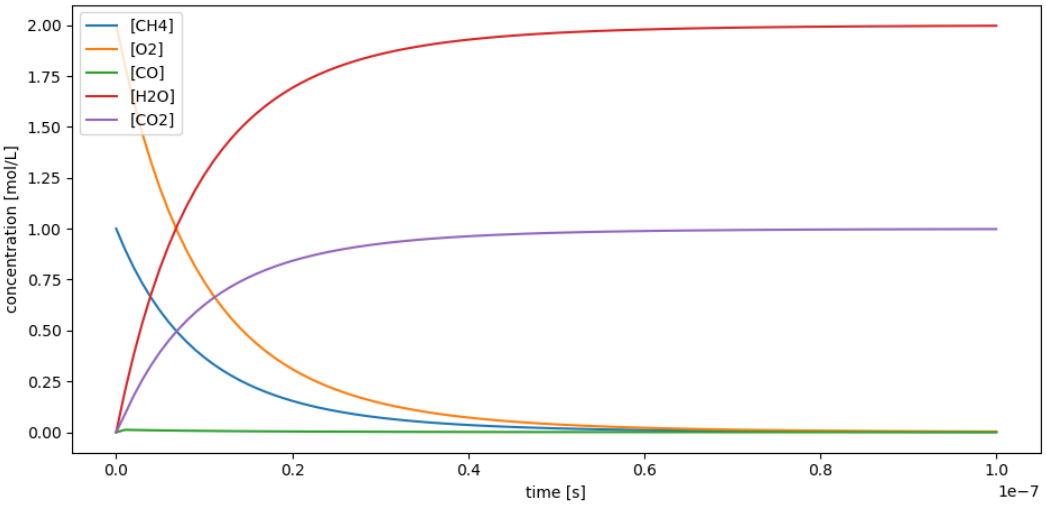
Figure 5.5: Absolute error in different iterations for the BDF method with order 1 to 5. The error is related to the serial approximation using BDF with order 1 to 5 and tolerances 10^{-14} (relative) and 10^{-20} (absolute). The dotted line is at 10^{-9} which is the criterion for the error.

and absolute tolerance of 10^{-20}. To get to an approximation error of 10^{-9} or less, 5 iterations have to be executed.

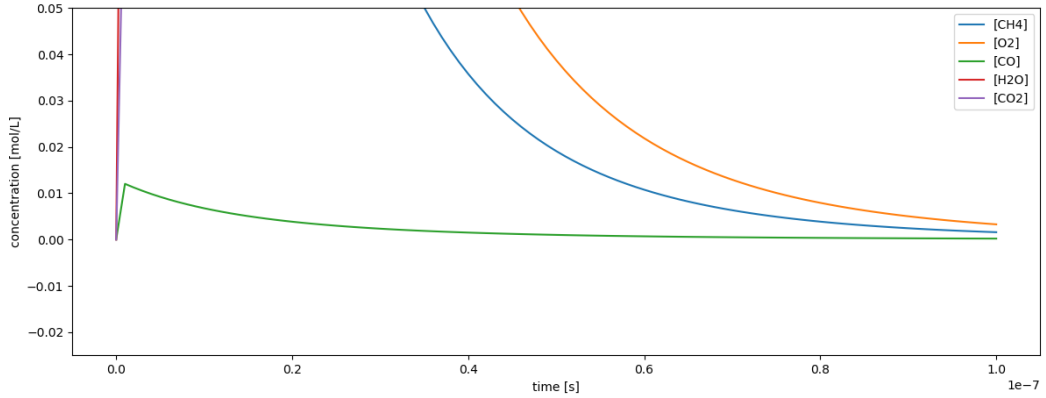
The final result is shown in figure 5.6. It is zoomed to observe the concentration of CO. Clearly, a difference in concentration of CO is visible. The line for the concentration however, is not smooth. This is not a result of the inaccuracy of the parareal algorithm, but of the selected grid size ΔT. Since the error is sufficient, it is not necessary to reduce this grid size.

Input argument	Value
t_0, t_n	$0, 1 \cdot 10^{-7}$
$[CH_4]_{t_0}, [O_2]_{t_0}, [CO]_{t_0}, [H_2O]_{t_0}, [CO_2]_{t_0}$	$1, 2, 0, 0, 0$ [mol/L]
T	1000 K
method	Backward Differentiation Formulas with order 1 to 5
Δt	$\frac{t_n}{100}$
$rtol_G, atol_G$	$1 \cdot 10^{-1}, 1 \cdot 10^{-1}$
$rtol_F, atol_F$	$10^{-14}, 10^{-20}$
k	5

Table 5.4: Inputs for parareal executed on the model for the two-step mechanism of combustion of methane. The initial concentrations are in stoichiometric ratio.



(a)



(b) Zoomed

Figure 5.6: Approximation of two-step mechanism by Parareal with inputs in table 5.4.

6

Conclusion and recommendations

In this research, we used the parareal algorithm to model the mechanism of combustion of methane. First, the parareal algorithm is introduced. With this introduction to parareal, specific choices for the coarse and fine propagator are considered. Following, the speedup, the order of accuracy and the convergence of parareal are treated and examined using two test equations.

Speedup
The parareal algorithm is designed to generate an approximation in less time, than the fine propagator executed in serial, but with comparable accuracy. The speedup of the parareal algorithm, related to the speed of the fine propagator in serial, is considered together with the system efficiency, that is, the speedup per processor. The speedup and system efficiency are ideal for $\frac{\Delta t}{\delta t} = 2\frac{\tau}{\Delta t}$ and $k = 1$. In this case, the speedup is $S = \frac{\tau}{\Delta t}$ and the system efficiency is $E = \frac{1}{2}$. However, for the case that $k = 1$ the approximation is could be insufficient in accuracy.

Order of accuracy
In the analysis of the order of accuracy, we assumed the fine propagator to be sufficiently accurate. To start, we mentioned that for a scalar linear initial value problem, the parareal algorithm increases the order of accuracy by a factor of $k+1$ compared to the order of the coarse propagator, where k is the number of iterations. Then this is proved for a non-linear initial value problem, where more properties are needed to generate the order of accuracy increase of $k+1$. First of all, the solution to the initial value system has to be stable. Moreover, the coarse propagator has to be Lipschitz continuous for a specific Lipschitz constant. Finally the coarse propagator has to be an discretisation with order p and has to satisfy a Lipschitz regularity in its initial condition. If all these constraints are met, the parareal algorithm increases the order with a factor of $k+1$.

Convergence
In this research, the assumption that the parareal will never converge better towards the serial fine propagation over the grid, is proven. This is done in two cases, one where the right-hand side of the initial value problem will only depend on the time and one where the right-hand side will also depend on the solution to the problem.

In both cases it is proven, that after N iterations, the approximation generated by the parareal algorithm will be equal to the approximation generated by the fine propagator executed in serial on the grid and will not improve by executing more iterations. In the case that the right-hand side does not depend on the solution to the problem, this theorem is even stronger: the approximation after only one iteration of the parareal algorithm will be equal to the approximation generated by the fine propagator executed in serial.

Combustion of methane

The parareal algorithm is executed on the two-step combustion mechanism of methane. First, the need for numerically integrating over two timescales is illustrated with applying the Forward Euler method in serial on the model. The result indicates that the Forward Euler method is for a part of the model a convenient method, but for the other part not applicable. The parareal algorithm can integrate over two timescales in one process and is therefore the right choice.

The approximation is generated by parareal using BDF method with an (automatically selected) order from 1 to 5, in the propagators. The coarse and fine propagators use different tolerances for the error in the time-integration method. Though this choice for BDF method in the propagators gives an accurate result, it does not lead to a significant speedup.

In this research, the same time-integration method, the BDF method with automatic selected time-step and order, is used for the two propagators. The difference in computational time is not high, even if extreme low values for the error tolerances are set in the fine propagator and extreme high values for the error tolerances are set in the coarse propagator. The parareal algorithm executed on this model therefore has a speedup of around 4. Which means it is 4 times faster than applying the serial fine propagator scheme on the same time grid. This speedup is with neglecting the communication between the processors and some implementation imperfections. The time for these matters can outweigh the time gain by the speedup. This can even lead to a longer running time for parareal than for the serial integration. Only if the difference in running time for the coarse and fine propagator is sufficient, there is a reasonable speedup. If different time-integration methods are used for the coarse propagator and the fine propagator, a reasonable ratio between the costs for executing the propagators, and therefore a reasonable speedup, can be obtained. This is therefore a matter that can be explored in further research.

Moreover, the mechanism of combustion of methane is a process with more than two partial equations. Further research can be about applying the parareal algorithm on the multiple (more than two) step mechanisms of combustion of methane. To increase speedup and efficiency, the multi-level parallelization as in [2] can be applied on this multiple-step mechanism.

Bibliography

[1] Guillaume Bal. Parallelization in time of (stochastic) ordinary differential equations. *Math. Meth. Anal. Num.(submitted)*, 2003.

[2] Guillaume Bal. On the convergence and the stability of the parareal algorithm to solve partial differential equations. In Timothy J. Barth, Michael Griebel, David E. Keyes, Risto M. Nieminen, Dirk Roose, Tamar Schlick, Ralf Kornhuber, Ronald Hoppe, Jacques Périaux, Olivier Pironneau, Olof Widlund, and Jinchao Xu, editors, *Domain Decomposition Methods in Science and Engineering*, pages 425–432, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-26825-3.

[3] Guillaume Bal and Yvon Maday. A "parareal" time discretization for non-linear pde's with application to the pricing of an american put. *Recent Developments in Domain Decomposition Methods*, 23, 01 2002. doi: 10.1007/978-3-642-56118-4_12.

[4] B. Franzelli, E. Riber, L.Y.M. Giquel, and T. Poinsot. Large-eddy simulation of combustion instabilities in a lean partially premixed swirled flame. *Combustion and Flame*, 159(2):621–637, 2012.

[5] Martin Gander, Felix Kwok, and Hui Zhang. Multigrid interpretations of the parareal algorithm leading to an overlapping variant and mgrit. *Computing and Visualization in Science*, 19, 06 2018. doi: 10.1007/s00791-018-0297-y.

[6] Martin J. Gander, Iryna Kulchytska-Ruchka, Innocent Niyonzima, and Sebastian Schöps. A new parareal algorithm for problems with discontinuous sources. *SIAM Journal on Scientific Computing*, 41(2):B375–B395, 2019. doi: 10.1137/18M1175653. URL `https://doi.org/10.1137/18M1175653`.

[7] H.B. Keller. *Numerical Methods for Two-Point Boundary Value Problems*. 1968.

[8] K. J. Laidler. A glossary of terms used in chemical kinetics, including reaction dynamics (iupac recommendations 1996). *Pure and Applied Chemistry*, 68(1):149 – 192, 1996. doi: https://doi.org/10.1351/pac199668010149. URL `https://www.degruyter.com/view/journals/pac/68/1/article-p149.xml`.

[9] Jacques-Louis Lions, Yvon Maday, and Gabriel Turinici. Résolution d'EDP par un schéma en temps "pararéel". *C. R. Acad. Sci., Paris, Sér. I, Math.*, 332(7):661–668, 2001. ISSN 0764-4442.

[10] Y. Maday and O. Mula. An adaptive parareal algorithm, 2019.

[11] Michael Minion. A hybrid parareal spectral deferred corrections method. *Communications in Applied Mathematics and Computational Science*, 5(2):265–301, 2011.

[12] J. Nievergelt. Parallel methods for integrating ordinary differential equations. *Commun. ACM*, 7(12):731–733, December 1964. ISSN 0001-0782. doi: 10.1145/355588.365137. URL `https://doi.org/10.1145/355588.365137`.

[13] Python Software Foundation. multiprocessing — process-based parallelism, 2020. URL `https://docs.python.org/3.6/library/multiprocessing.html`. Accessed: 2020-06-29.

[14] Gunnar A Staff. The parareal algorithm. *Science And Technology*, 60(2):173–184, 2003.

[15] C. Vuik, F. J. Vermolen, M. B. van Gijzen, and M.J. Vuik. *Numerical Methods for Ordinary Differential Equations*. VSSD, 2018.

[16] J. Warnatz, U. Maas, and R.W. Dibble. *Combustion*. Springer-Verlag Berlin Heidelberg, 4th edition, 2006. ISBN 978-3-540-25992-3.

[17] C.K. Westbrook and F.L. Dryer. Simplified reaction mechanisms for the oxidation of hydrocarbon fuel in flames. *Combustion Science and Technology*, 27(1-2):31–43, 1981.

A

Implementation parareal algorithm

The parareal algorithm is implemented in Python. The considered time-integration methods in section 2.3 are defined. If the solution is unknown, for the representation of the solution to the initial value problem, the BDF method with low tolerances is used.

```
1  from math import *
2  import numpy as np
3  import matplotlib.pyplot as plt
4  import matplotlib.animation as animation
5  from scipy.integrate import solve_ivp
6  import time as timemodule
7  import timeit
8  import multiprocessing as mp
9  from scipy.optimize import fsolve
10 #%% Time-integration methods defined ##########################################
11
12 ## Only uncomment exactly() if the solution to the problem is known
13 #def exactly(f,Dt,dt,t,x,dummy_1=0,dummy_2=0):
14 #  x = solution(t+Dt,t,x)
15 #  return x
16
17 def forwardeuler(f,Dt,dt,t,x,dummy_1=0,dummy_2=0):
18   t_end = t + Dt
19   while t < t_end:
20     x = x + dt*f(t,x)
21     t = t + dt
22   return x
23
24 def rungekutta(f,Dt,dt,t,x,dummy_1=0,dummy_2=0):
25   t_end = t + Dt
26   while t < t_end:
27     k1 = dt * f(t      ,x)
28     k2 = dt * f(t+dt/2,x+k1/2)
29     k3 = dt * f(t+dt/2,x+k2/2)
30     k4 = dt * f(t+dt   ,x+k3)
31     x = x + 1/6*(k1+2*k2+2*k3+k4)
32     t = t + dt
33   return x
34
35 def backwardeuler(f,Dt,dt,t,x,dummy_1=0,dummy_2=0):
36   t_end = t + Dt
37   while t < t_end:
38     x_old = x
39     def equations(x_new):
```

```
40        f_xnew = f(t+dt,x_new)
41        return (x_new - x_old - dt*f_xnew)
42      solved = fsolve(equations, x_old+dt*f(t,x_old),full_output=True)
43      x = solved[0]
44      t = t + dt
45    return x
46
47  def trapezoidal(f,Dt,dt,t,x,dummy_1=0,dummy_2=0):
48    t_end = t + Dt
49    while t < t_end:
50      x_old = x
51      def equations(x_new):
52        f_xold = f(t,x_old)
53        f_xnew = f(t+dt,x_new)
54        return (x_new - x_old - dt/2*(f_xnew+f_xold))
55      x = fsolve(equations, x_old+dt*f(t,x_old))
56      t = t + dt
57    return x
58
59  def solveivp(f,Dt,dt,t,x,rtol,atol):
60    solved = solve_ivp(f,(t,t+Dt),x,method='BDF',rtol=rtol,atol=atol)
61  #  print('The number of function evaluations is '+str(solved.nfev))
62    x_j = solved.y[:,-1]
63    return x_j
64
65  #%%#########################################################################
66
67  def solver(method,f,dt,t,x,rtol=1e-1,atol=1e-1):
68    """Integrate initial value problem with initial state x[i] on t[i] over range of i
69      input = t: [t0,t1,...,tn]
70              x: [x0,x1,...,xn] (possibly xi as vector)
71      output= x: [x1_new,x2_new,...,x3_new]"""
72    Dt = t[1]-t[0]
73    pool = mp.Pool(mp.cpu_count())
74    result = [pool.apply_async(method,args=(f,Dt,dt,t[i],x[i],rtol,atol)) for i in range(len(t))
          ]
75    result = [res.get() for res in result]
76    pool.close()
77    result = np.array(result)
78    return result
79
80  #%%#########################################################################
81
82  def G(rhs, array):
83    """Coarse propagator:
84      input = [[t_0,x_0],[t_1,x_1],...,[t_n,x_n]]
85      output = [[t_1,xnew-1],[t_2,xnew_2],...[[t_n,xnew_n]]"""
86    t, x = array[:-1,0], array[:-1,1:]
87    Dt = t[1]-t[0]
88    x = solver(method,rhs,Dt,t,x,rtol=rtol_g,atol=atol_g)
89    t = np.array([t + Dt]).T
90    return np.hstack((t,x))
91
92  def F(rhs, array, division):
93    """Fine propagator:
94      input = [[t_0,x_0],[t_1,x_1],...,[t_n-1,x_n-1]]
95      output = [[t_1,xnew-1],[t_2,xnew_2],...[[t_n,xnew_n]]"""
96    t, x = array[:-1,0], array[:-1,1:]
97    dt = (t[1]-t[0])/division
98    x = solver(method,rhs,dt,t,x,rtol=rtol_f,atol=atol_f)
99    t = np.array([t + dt*division]).T
```

```
100     return np.hstack((t,x))
101
102 class Parareal(object):
103     """Using the parareal algorithm to approximate the solution of dx/dt=rhs(t,x)"""
104     def __init__(self, rhs, start, end, initialvalue, coarsesteps, division, k):
105         self.initialvalue = initialvalue
106         self.coarsesteps = coarsesteps
107         self.division = division
108         self.start, self.end = start, end
109         self.time = np.linspace(self.start, self.end, num = self.coarsesteps+1)
110         self.Dt = self.time[1] - self.time[0]
111         self.rhs = rhs
112         self.answer= []
113
114         self = self.getstartvector()
115         self = self.iterate(k)
116
117     def getstartvector(self):
118         print('ITERATION␣0')
119         timer_begin = timeit.default_timer()
120         t = self.time
121         t, x = t[0], self.initialvalue
122         answer = [np.hstack([t,x])]
123         steps = 0
124         while steps < self.coarsesteps:
125             ti = answer[-1][0]
126             xi = answer[-1][1:]
127             x = method(self.rhs,self.Dt,self.Dt,ti,xi,rtol_g,atol_g)
128             t = t + self.Dt
129             steps += 1
130             new = np.hstack([t,x])
131             answer.append(new)
132         self.answer = np.array(answer)
133         timer_end = timeit.default_timer()
134         print('running␣time:',timer_end-timer_begin)
135         return self
136
137     def iterate(self, k):
138         answer = [self.answer] #get the start vector
139         t = self.time
140         for j in range(k): #execution of the parareal iterations
141             old = answer[-1]
142             print('\n\nITERATION',j+1)
143             # parallel computations
144             timer_begin = timeit.default_timer()
145             Garray = G(self.rhs, old)
146             timer_end = timeit.default_timer()
147             print('running␣time␣coarse:',timer_end-timer_begin)
148             timer_begin = timeit.default_timer()
149             Farray = F(self.rhs, old, self.division)
150             timer_end = timeit.default_timer()
151             print('running␣time␣fine:',timer_end-timer_begin)
152             # non-parallel computations
153             x = self.initialvalue
154             nex = [np.hstack([t[0],x])]
155             i = 0
156             timer_begin = timeit.default_timer()
157             while i < self.coarsesteps:
158                 ti = nex[-1][0]
159                 xi = nex[-1][1:]
160                 grove = method(self.rhs,self.Dt,self.Dt,ti,xi,rtol_g,atol_g)
```

```
161             correctie = Farray[i,1:] - Garray[i,1:]
162             x = grove + correctie
163             i += 1
164             new = np.hstack([t[i],x])
165             nex.append(new)
166           timer_end = timeit.default_timer()
167           print('running time update scheme: ',timer_end-timer_begin)
168           nex = np.array(nex)
169           answer.append(nex)
170         self.answer = np.array(answer)
171         return self
172
173       def plot(self, namen):
174         sol=[]
175         for k in range(len(self.answer)):
176           sol.append(self.answer[k])
177
178         fig = plt.figure()
179         ax = fig.add_subplot(111)
180         for i in range(len(self.initialvalue)):
181 #         for k in range(iterations): #uncomment these lines if the result has to be shown for
              different iterations
182 #           ax.plot(sol[k][:,0],sol[k][:,i+1])
183           ax.plot(sol[-1][:,0],sol[-1][:,i+1],label='['+namen[i]+']')
184         ax.legend()
185
186         ax.set_xlabel('time [s]')
187         ax.set_ylabel('concentration [mol/L]')
188         plt.show()
189         return
190
191       def plotdifference(self):
192         time = self.time
193         steps = []
194         for i in range(len(self.answer)-1):
195           step = (self.answer[i+1]-self.answer[i])[:,1:len(self.initialvalue)+1]
196           steps.append(step)
197
198         fig = plt.figure()
199         ax = fig.add_subplot(111)
200         color = ['b','g','r','c','m','y','k','grey','chocolate','olive']
201         linestyle = ['solid','dotted','dashed','dashdot']
202         for i in range(len(self.initialvalue)-1): #for loop over all the elements
203           for a, step in enumerate(steps): #for loop over all the steps
204             ax.plot(time,np.log(abs(step[:,i])),label='step'+str(a+1), color=color[a%10],
                    linestyle =linestyle[i])
205
206         # Makes sure no label is shown twice
207         handles, labels = ax.get_legend_handles_labels()
208         labels, ids = np.unique(labels, return_index=True)
209         handles = [handles[i] for i in ids]
210         ax.legend(handles, labels, loc='best')
211
212         # Make a second x-axis with the gridpoints
213         ax2 = ax.twiny()
214         gridpoints = (time-self.start)/self.end * self.coarsesteps
215         ax2.plot(gridpoints,time*0,alpha=0)
216         ax2.set_xlabel('gridpoint')
217
218         ax.set_xlabel('time [s]')
219         ax.set_ylabel('update term in logs')
```

```
220       plt.show()
221       return
222
223     def ploterror(self,solution):
224       fig = plt.figure()
225       ax = fig.add_subplot(111)
226       for k in range(len(self.answer)):
227         error_k = self.answer[k]-solution
228         sumerror_k = np.sqrt(np.sum(error_k**2,axis=1))
229         print('error in iteration '+str(k)+ ' is '+str(sumerror_k[-1]))
230         ax.plot(self.time,sumerror_k,label='iteration '+str(k))
231       ax.hlines(y=10**(-9),xmin=self.start-0.5e-8,xmax=self.end+0.5e-8,color='grey',linestyle='
              --')
232
233       # Make a second x-axis with the gridpoints
234       ax2 = ax.twiny()
235       gridpoints = (self.time-self.start)/self.end * self.coarsesteps
236       ax2.plot(gridpoints,self.time*0,alpha=0)
237       ax2.set_xlabel('gridpoint')
238
239       ax.set_xlim(self.start-0.5e-8,self.end+0.5e-8)
240       ax.legend()
241       ax.set_xlabel('time [s]')
242       ax.set_ylabel('error')
243       ax.set_yscale('log')
244       plt.show()
245
246   #%% Simple test model ############################################################
247   # Define rhs(t,x) ##############################################################
248   #def rhs(t, x):
249   #  ans = x[0]*t
250   ##  ans = np.sin(2/5*pi*t)+0.5*np.sin(2/10*pi*t)
251   #  return np.array([ans])
252   #
253   #def solution(t,t0,x0):
254   #  ans = x0[0]*np.exp(0.5*t**2-0.5*t0**2)
255   ##  C =  x0[0] + 5/(2*pi)*(np.cos(2/5*pi*t0)+np.cos(2/10*pi*t0))
256   ##  ans = -5/(2*pi)*(np.cos(2/5*pi*t)+np.cos(2/10*pi*t)) + C
257   #  return np.array([ans])
258   #
259   #start,end = 0,3
260   #coarsesteps = 8
261   #division = 6
262   #x_0 = np.array([1])
263
264   #%% Model 1-step ################################################################
265   # Define constants #############################################################
266   #R = 8.31446261815324
267   #T = 1000
268   #A, b, E = 1.1e10, 0.0, 20000
269   #order = [1,0.5]
270   #
271   ## Define rhs(t,x) #############################################################
272   #
273   #namen = ['CH4', 'O2', 'CO2', 'H2O']
274   #
275   ## Constant temperature
276   #def rhs(t, x):
277   #    CH4, O2, CO2, H2O = x[0],x[1],x[2],x[3]
278   #    k = A * T**b * np.exp(-E/(R*T))
279   #    rv1 = k * CH4**order[0] * O2**order[1]
```

```
#     dCH4, dO2, dCO2, dH2O = -rv1, -2*rv1, rv1, 2*rv1
#     return np.array([dCH4, dO2, dCO2, dH2O])
#start,end = 0,2e-8
#coarsesteps = 50
#division = 10
#x_0 = np.array([1,2,0,0])

### Model 2-step ###################################################################
# Define constants ###################################################################
R = 8.31446261815324
T = 1000

A1, b1, E1 = 4.9E+09, 0.0, 35500
order1 = [0.5, 0.65]

A2, b2, E2 = 2.00E+08, 0.7, 12000
order2 = [1, 0.5]

# Define rhs(t,x) ###################################################################

namen = ['CH4', 'O2', 'CO', 'H2O', 'CO2']

# Constant temperature
def rhs(t, x):
    CH4, O2, CO, H2O, CO2 = x[0],x[1],x[2],x[3],x[4]
    k1 = A1 * T**b1 * np.exp(-E1/(R*T))
    k2 = A2 * T**b2 * np.exp(-E2/(R*T))
    rv1 = k1 * CH4**order1[0] * O2**order1[1]
    rv2 = k2 * CO**order2[0] * O2**order2[1]
    dCH4, dO2, dCO, dH2O, dCO2 = -rv1, -1.5*rv1-0.5*rv2, rv1-rv2, 2*rv1, rv2
    return np.array([dCH4, dO2, dCO, dH2O, dCO2])
start,end = 0,1e-7
coarsesteps = 100
division = 1
x_0 = np.array([1,2,0,0,0])

### Execute parareal algorithm ######################################################
method = solveivp
rtol_g,atol_g = 1e-1,1e-1
rtol_f,atol_f = 3e-14,1e-20

if __name__ == '__main__':
  iterations = 12
  __spec__ = None
  timer_begin = timeit.default_timer()
  approx = Parareal(rhs, start, end, x_0, coarsesteps, division, iterations)
  timer_end = timeit.default_timer()
  print('\nRuntime: ',timer_end - timer_begin)

### Plot ############################################################################
  approx.plot(namen)

### Plot difference in each iteration of parareal algorithm ########################
  approx.plotdifference()

### Calculate solution
  rtol_g, atol_g = 3e-14,1e-20
  method = solveivp
  solution = Parareal(rhs,start,end,x_0,coarsesteps*division,1,0).answer[0]

### Approximate error ###############################################################
```

```
341   approx.ploterror(solution)
342
343 #%% Test the order of error (only if solution is known) ######################
344 #   t = approx.time
345 #   error = []
346 #   for i in range(iterations+1):
347 #     error.append(abs(approx.answer[i,-1,1] - solution(end,start,x_0)))
348 #   print('N = ',coarsesteps)
349 #   print('0: ',error[0])
350 #   for i in range(1,iterations+1):
351 #     diff = error[i]/error[i-1]
352 #     print(i,': ',error[i],' ratio: ',diff)
```