POLITECNICO DI TORINO

Master's Degree in Mechanical Engineering

Master's Degree Thesis

Efficient Numerical Integration for the Rate Equations of Combustion Mechanisms

A High Order Convergence Method

Advisors
Prof. Carlos Pantano-Rubino,
University of Illinois at Urbana-Champaign
Prof. Alessandro Ferrari
Politecnico di Torino

Candidate
Federico Miretti

Academic Year 2017–2018

Abstract

Differential equations derived from the chemical kinetics of combustion mechanism are often complex and must be solved numerically. As the number of reactions involved increases, the computation time can increase dramatically. Thus, efficient algorithms that account for the structure of the differential equations to be solved are required.

The purpose of this work is to explore a particular implementation of an implicit differentiation scheme that uses a third order root-finding algorithm to reduce the number of iterations performed at each time-step of the solution and the conditions under which this may lead to reduced computation times.

The Backward differentiation formula of the second order, BDF2, is coded with the commercial software Matlab to solve a simplified reaction mechanism for hydrogen oxidation. The algorithm is implemented with two different iteration functions. First, the widely known Newton's algorithm, which is of second order convergence, is used to advance the solution at every time-step. Then the Chebyshev iteration function, which is obtained by systematically increasing the order of convergence of Newton's iteration function by Schröder's process of the first kind, is introduced and the outcomes are compared. The effect of the step-size on the number of iterations performed by the two algorithms is compared.

While Newton's iteration function requires the computation of the Jacobian of the system of differential equations to be solved, the third order iteration function additionally requires the computations of the Hessian. Therefore, some techniques are introduced to speed up the computation of the Hessian, such as sparse storage and an inexact analytical form. It is found that under specific conditions for the time-step this can significantly reduce the overall execution time of the algorithm. It remains to be explored wether the application of this solutions to a complex combustion mechanism would prove beneficial, in terms of computation time, with respect to Newton's iteration function, as it is expected that this reduction would be even greater as the size of the problem increases. Further elaboration of this work may thus prove of interest in the development of software specifically designed to simulate the chemical kinetics of complex combustion mechanisms.

Contents

Chapter 1

Introduction

1.1 Stiff ordinary differential equations

The concentration of chemical species in a combustion process is described by a set of coupled differential equations:

$$\dot{y} = f(y),$$

where y denotes the vector of chemical composition and $f(y)$ denotes the inter-related reaction rates.

Realistic combustion mechanisms includes a great number of elementary reaction processes involving different chemical species, such as molecules, atoms, radicals or ions. An analytical solution for these problems may be practically unattainable or even inexistent[16]; therefore, a numerical solution is required to compute the evolution of the chemical compositions with respect to time. These profiles are useful in many applications, particularly for validating the mechanism by comparison with experimental data.

With regard to their numerical modelling and solution, these problems have some particular aspects that have to be accounted for. For example, because of the nature of combustion phenomena, the concentration of the species over time often shows steep variations. The numerical solution of these equations requires the use of very small timesteps. This however can be strongly computationally burdening, and it can be so unnecessarily due to the fact that such a small stepsize is actually needed in a restricted range of the solution, where rapid changes happen.

A more rigorous definition of stiffness in ODEs can be found in various sources in literature. For example,[13] proposes the following definition, based on the observation that stiffness occurs when there is a wide range of time scales in the solution.

A set of equations of the form

$$\dot{\mathbf{y}} = \mathbf{J} \cdot \mathbf{y} \tag{1.1}$$

where $\mathbf{J}$ is the Jacobian matrix, is stiff if

$$\begin{aligned} &(i) \quad \Re(\lambda_j) < 0, \quad \text{for } j = 1, ..., N, \quad \text{and} \\ &(ii) \quad \frac{\max(\Re(\lambda_j))}{\min(\Re(\lambda_j))} >> 1. \end{aligned} \tag{1.2}$$

If stiff systems are treated with an explicit numerical method, the timestep has to be very small in order for the solution to be stable. Thus, implicit numerical methods are often used to treat these problems, since they usually are more numerically stable at larger timesteps[12]. Despite involving some iteration method to compute the solution at every step, this usually results in less computations.

The simplest explicit numerical scheme is the *Forward Euler* method, taking the form:

$$y^{n+1} = y^n + hf(t^n, y^n). \tag{1.3}$$

A simple example case can show the previously mentioned limitations of explicit methods, when solving stiff equations.

Consider the scalar initial value problem

$$\begin{aligned} &\frac{\mathrm{d}y}{\mathrm{d}t} = a(1-y)e^{-\frac{b}{y+1}}, \\ &y(0) = 0. \end{aligned} \tag{1.4}$$

with parameters $a = 1$, $b = 2$.

Figure 1.1 compares the numerical solution computed using the *Forward Euler method*, at various stepsizes, with the solution obtained using an accurate solver, based on the *Rosenbrock method*[7].

For the largest stepsize value, the method fails to converge as the solution is unbounded. For the second largest stepsize, the solution stays bounded but fails to converge rapidly. Only the smallest stepsizes achieve acceptable accuracy of the solution.

An implicit method would achieve better accuracy for larger stepsize values. Common examples of such a method are those belonging to the family of *Backward Differentiation Formulas*, which is the object of the next section.

1.2 Implicit methods for stiff equations

Backward Differentiation Formulas are linear multistep methods[13], of the form:

$$\begin{aligned} &y^{n+k} = h\beta^k f^{n+k} - \sum_{j=0}^{k-1} \alpha^j y^{n+j}, \\ &\text{with } \alpha^0 \neq 0 \text{ and } \beta^k \neq 0. \end{aligned} \tag{1.5}$$

The number of steps k considered for the computation of each step n, is also the order of convergence of the method.

The simplest of the method is that having first order, the *Backward Euler* method, with $\alpha^0 = -1$ and $\beta^1 = 1$:

$$y^{n+1} = y^n + hf(t^{n+1}, y^{n+1}). \tag{1.6}$$

The implicitness of this method comes from the fact that the computation of $f(t^{n+1}, y^{n+1})$ requires the solution of the algebraic equation

$$y^{n+1} - hf(t^{n+1}, y^{n+1}) = y^n, \tag{1.7}$$

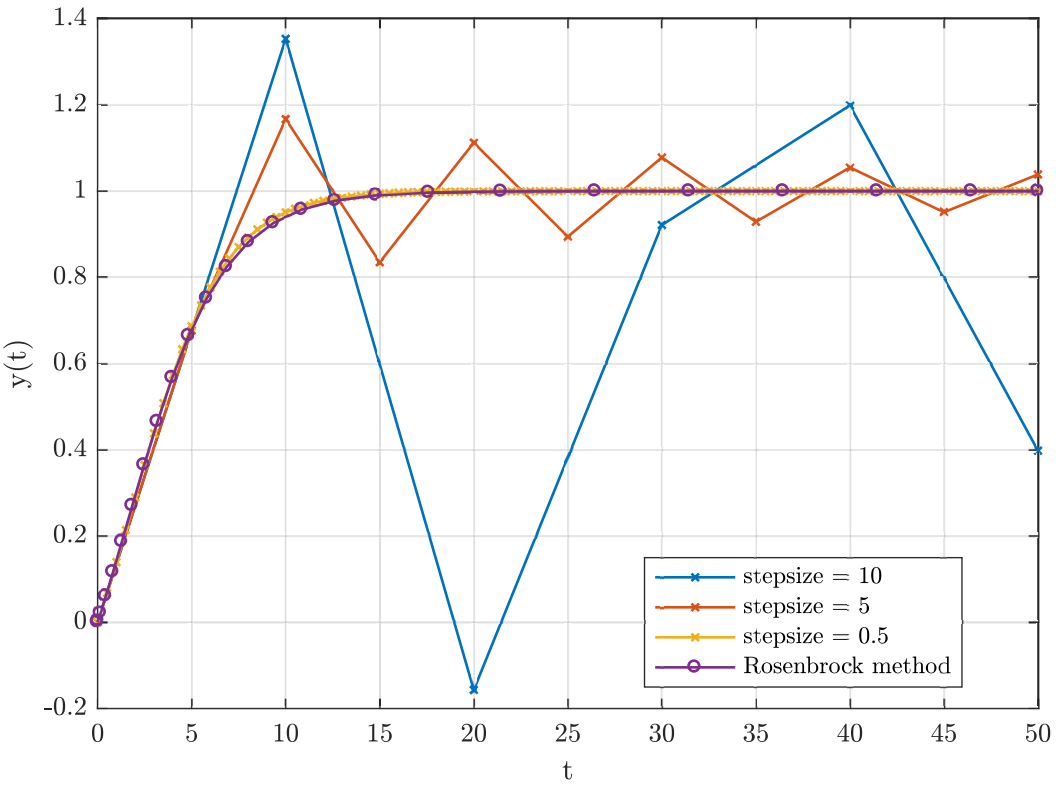

Figure 1.1: Solution with forward Euler Method, for various stepsize values.

for every timestep n. This can be done by applying a root-finding algorithm, such as Newton's method, to the function

$$F(y^{n+1}) = y^{n+1} - hf(t^{n+1}, y^{n+1}) - y^n. \tag{1.8}$$

The iterative execution of the algorithm means that implicit methods require more computation effort compared to explicit methods. However, for stiff equations, the larger stepsize requirement of the former still makes them more advantageous.

To find the solution, for any timestep, by Newton's method, requires an initial guess. Then, for each iteration, the difference between the next tentative value of the solution and the current one, is given by the iteration function:

$$\Delta y = y_{l+1}^{n+1} - y_l^{n+1} = -\left(\frac{\partial F(y_l^{n+1})}{\partial y}\right)^{-1} F(y_l^{n+1}), \tag{1.9}$$

where subscript l indicates the current iteration. The algorithm is repeated until this difference becomes smaller than a set tolerance.

Figure 1.2 compares the solution obtained with this method with the previously discussed Forward Euler method, for a relatively large stepsize. It is evident that, while the explicit method fails to converge in a reasonable time, the implicit one does, even if it does so more slowly.

The issue of the computational cost of implicit methods has to be addressed. The number of iterations required to solve for each step n, depends on the goodness of the first guess, which is always required to execute the root-finding algorithm, and the algorithm itself.

Thus, the rate of convergence of the algorithm itself is fundamental in determining the computational time required to solve the initial value problem. Several methods have been developed to systematically increase the order Newton's iteration function, which is a second order method.

For example, using Schröder's process of the first kind, as discussed in [6], one could build the third order iteration function

$$\Delta y = y_{k+1}^{n+1} - y_k^{n+1} = -\frac{F(y_k^{n+1})}{F^{(1)}(y_k^{n+1})} - \frac{1}{2}\frac{F^{(2)}(y_k^{n+1})}{F^{(1)}(y_k^{n+1})}\left(\frac{F(y_k^{n+1})}{F^{(1)}(y_k^{n+1})}\right)^2, \tag{1.10}$$

where $F^{(1)}$ and $F^{(2)}$ indicate the first and second derivatives of F. This method is also known as the third order *Chebyshev* method.

This algorithm requires more computational effort for each iteration, due to the additional terms in the iteration function, but also requires less iterations to estimate the solution at each timestep.

It is important to note that, by setting a reasonably small tolerance for the root-finding algorithm, the solution is largely unaffected by the order of the algorithm itself, as shown in figure 1.3.

The advantage of using a higher order iteration function lies in the reduced number of iterations required to evaluate the solution at each timestep. Despite the added computational complexity at each iterations, this in general leads to an overall reduction in computational effort.

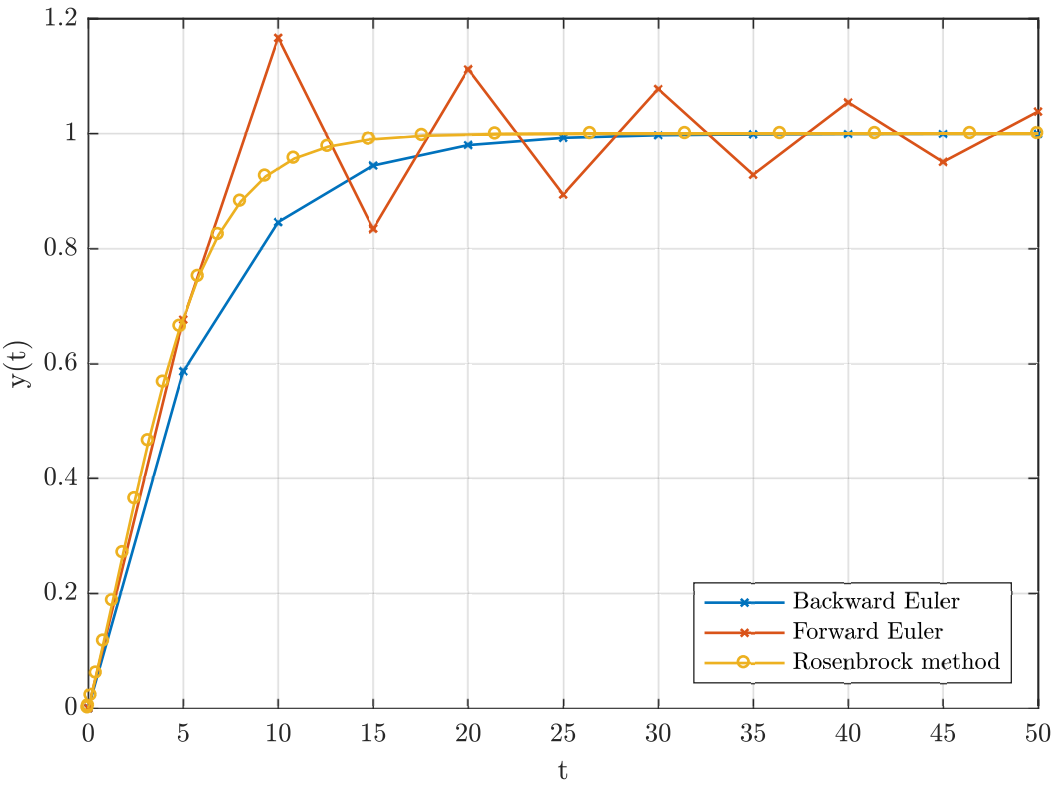

Figure 1.2: Solution with Forward Euler Method and Backward Euler method

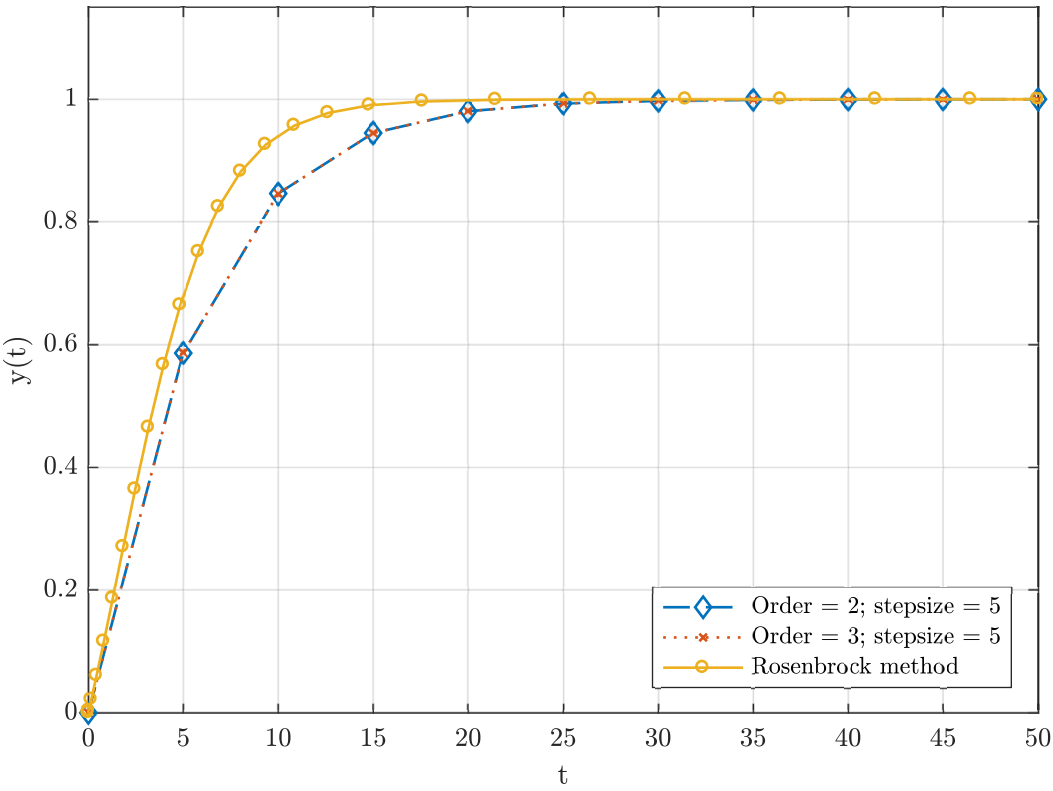

Figure 1.3: Solution with Forward Euler method, comparison between Newton algorithm (2^{nd} order) and Chebyshev (3^{rd} order).

Figure 1.4 compares the number of iterations, performed at every timestep, when applying the two mentioned algorithms, based on Newton's method (of second order) and Chebyshev method (of third order), to solve the example initial value problem 1.4.

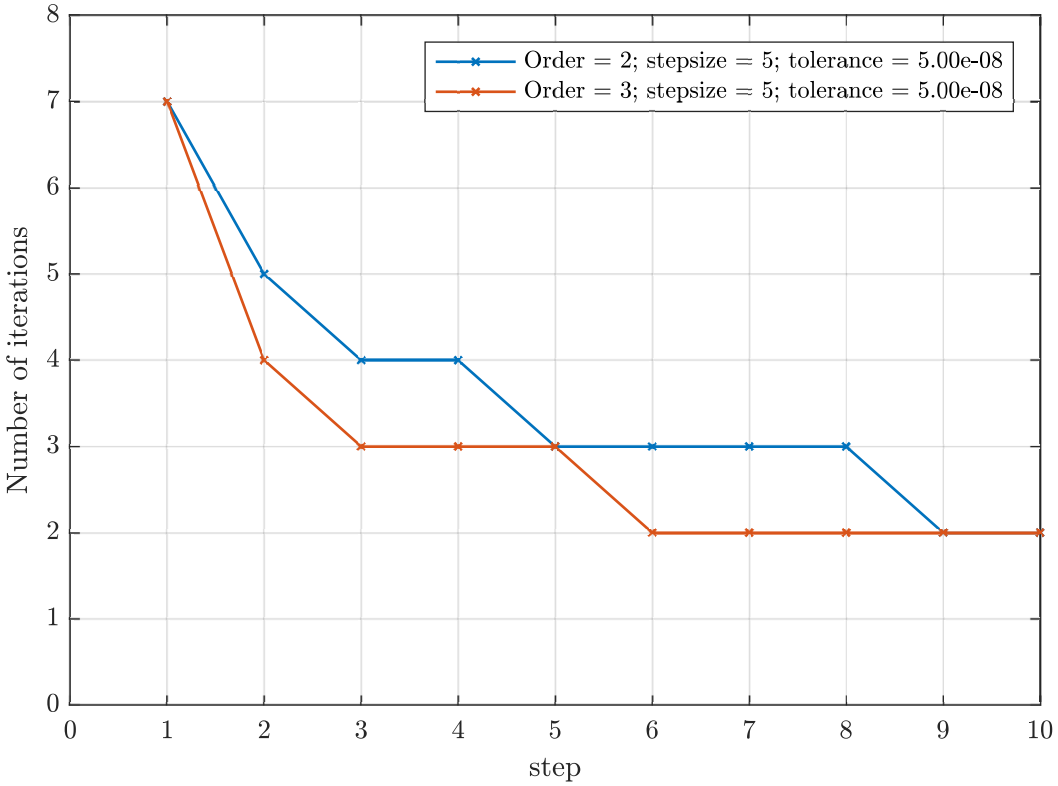

Figure 1.4: Number of iterations for each timestep, comparison between Newton algorithm (2^{nd} order) and Chebyshev (3^{rd} order).

The root-finding algorithm stops its execution when the difference between two successive tentative values falls below a certain tolerance, which is usually set in terms of the stepsize. Therefore, the number of iterations performed for every timestep is also dependant on the choice of such tolerance, as shown by figure 1.5.

If the tolerance value for the root-finding algorithm is set relatively to the stepsize, then the number of iterations is also affected by this parameter. This means that decreasing the stepsize would decrease the tolerance, thus requiring more a more accurate solution to satisfy the requirement.

However, reducing the stepsize also reduces the variation of the actual solution $y(t_n)$ for any timestep n. Since the initial guess of the algorithm, used to calculate the solution y^{n+1} is in general depending on the value of the solution at the previous timestep y^n, as the difference between the solution at two successive timestep reduces, the inital guess gets closer to the final approximation. Thus, as figure 1.6 shows, the number of iterations for each timestep is generally reduced by reducing the timestep, despite the more stringent absolute tolerance enforced.

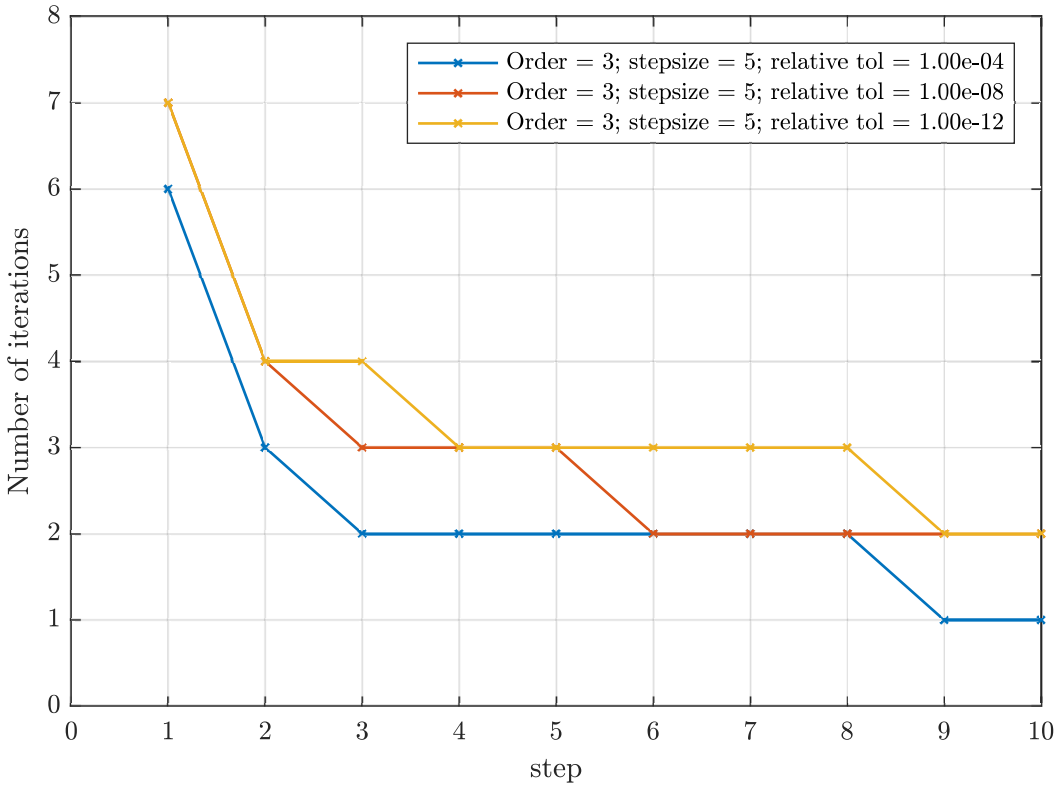

Figure 1.5: Number of iterations for each timestep, with Chebyshev (3^{rd} order) algorithm, for various relative tolerance values. Absolute tolerance is the product of the relative tolerance and stepsize.

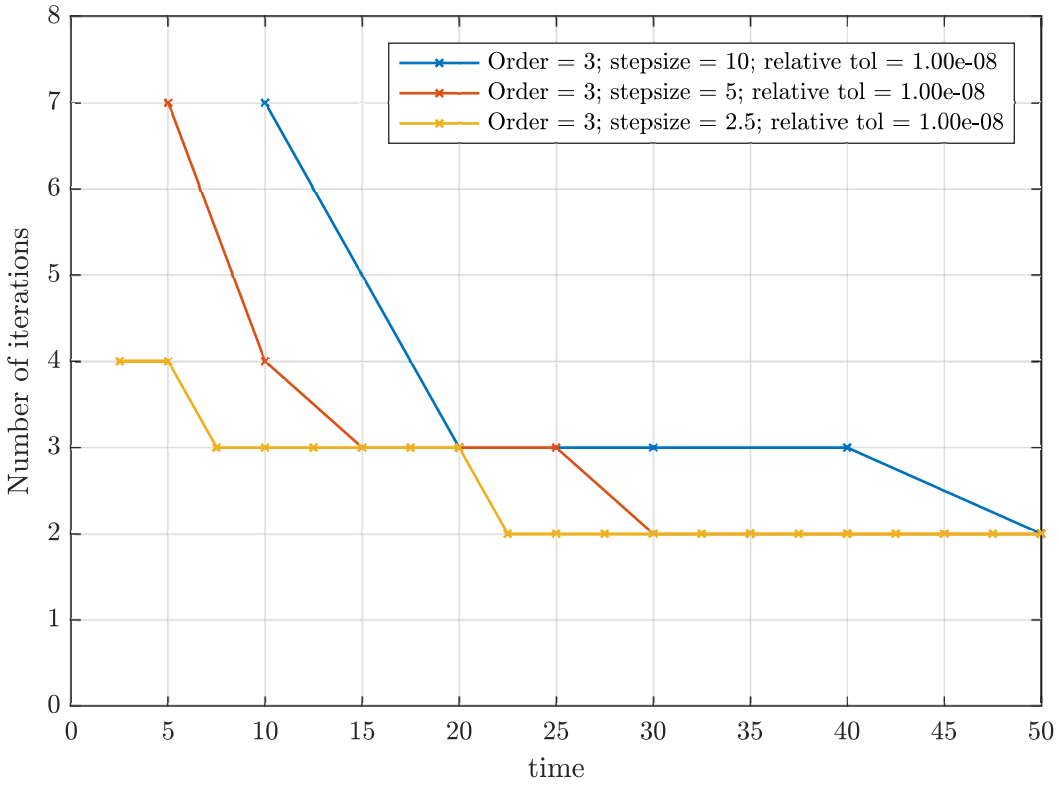

Figure 1.6: Number of iterations for each timestep, with Chebyshev (3^{rd} order) algorithm, for various stepsize values. Absolute tolerance is the product of the relative tolerance and stepsize.

Chapter 2

The BDF2 algorithm

2.1 Increasing the order of convergence

In the previous section, Backward Differentiation Formulas were introduced, taking the form in (1.5), and the BDF formula of order 1 was used to solve a sample initial value problem. In most applications, the rate of convergence of the solution obtained by solving with this scheme may not yield satisfactory results.

It is useful to recall that the order of accuracy of a method expresses the rate of convergence of the solution. A method is p-th order accurate if the global truncation error is proportional to the p-th power of the stepsize [13]. Thus, we can obtain a more accurate solution, at the same stepsize, by using a higher order method.

The second-order Backward Differentiation Formula (*BDF2*) is:

$$y^{n+2} - \frac{4}{3}y^{n+1} + \frac{1}{3}y^n = \frac{2}{3}hf^{n+2}. \tag{2.1}$$

It should be noted that this method is a 2-step method, meaning that in order to compute one step, two previous steps need to be known. When the method is started, only one step is known, that is the initial condition; thus, one more step is required or the method cannot be applied. One way to solve the issue is to start the method with *bootstrapping*.

1. Define a mid-step value $y^{\frac{1}{2}}$
2. Evaluate one step of a first order method, such as BDF1, with half the stepsize used in the main solution, to evaluate $y^{\frac{1}{2}}$

$$y^{\frac{1}{2}} = y^0 + \frac{h}{2}f^{\frac{1}{2}} \tag{2.2}$$

3. Evaluate one step the second order method (BDF2), with half the stepsize used in the main solution, to evaluate y^1

$$y^1 - \frac{4}{5}y^{\frac{1}{2}} + \frac{1}{3}y^0 = \frac{2}{3}hf^1 \tag{2.3}$$

4. Solve the equation with the second order method (BDF2), as in (2.1)

This method can be applied to the sample IVP problem in (2.4). As figure 2.1 shows, the solution obtained with BDF2 is more accurate for the same stepsize; moreover, it converges to the exact solution faster as the stepsize reduces. Indeed, the truncation error of BDF2 is proportional to h^2, while for BDF1 it is proportional to h.

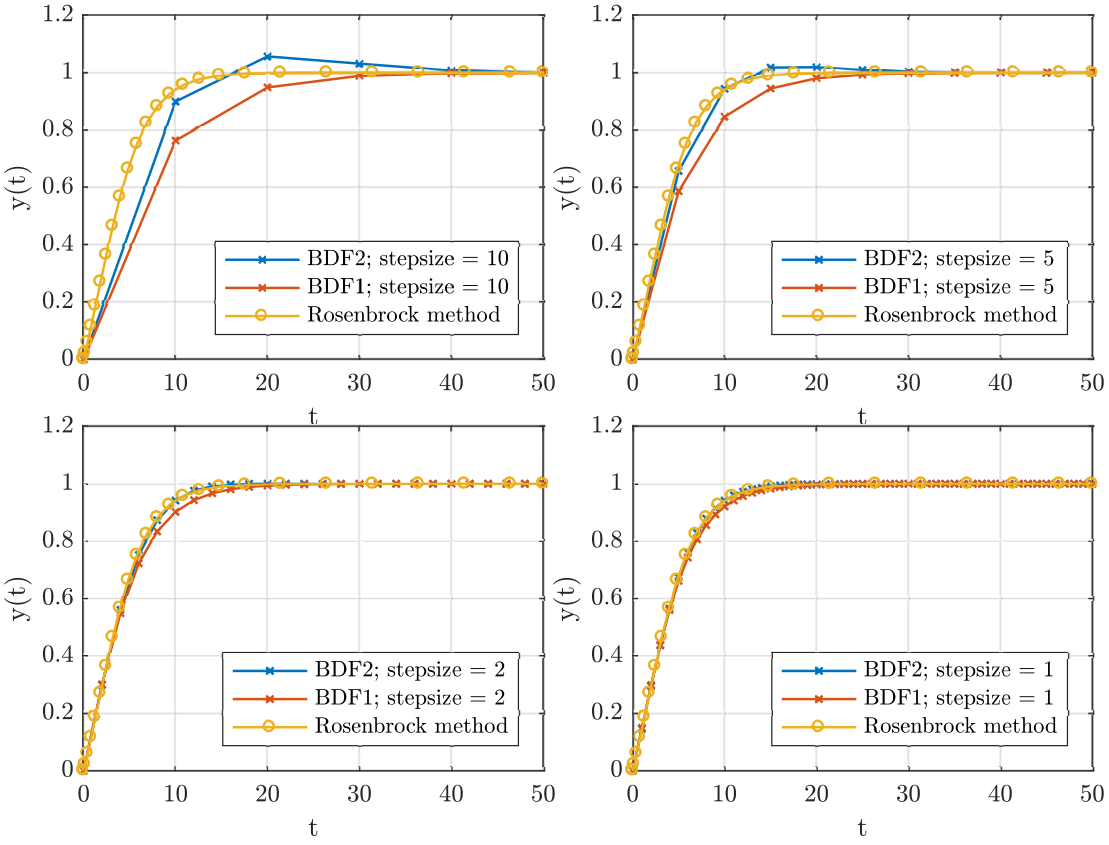

Figure 2.1: Comparison between the solution obtained with BDF1 and BDF2, for different stepsize. It can be observed as the solution obtained with BDF2 is more accurate.

2.2 Increasing the size of the problem

Until now, the solution of a scalar problem was analysed; nonetheless, all of the previous discussions can be extended to the solution of a system of ordinary differential equation.

Combustion processes involve a variety of species whose concentration and rate of reaction change in time, continuously interacting with one another, and the description of the combustion process will therefore involve one differential equation for each species. The number of chemical species involved, and therefore the number of differential equations that compose the initial value problem, can be very high.

Consider the initial value problem

$$\begin{aligned}
&\frac{\mathrm{d}y_1}{\mathrm{d}t} = a_{11}(1-y_1)(1-y_2)\, e^{-\frac{E_1}{y_1+1}},\\
&\frac{\mathrm{d}y_2}{\mathrm{d}t} = a_{21}(1-y_2)\, e^{-\frac{E_2}{y_1+1}} - a_{22}\, y_1\, y_2\, e^{-\frac{E_3}{y_1+1}},\\
&\mathbf{y}(0) = \mathbf{0}.
\end{aligned} \tag{2.4}$$

with parameters $a_{11} = a_{21} = a_{22} = 1$, $E_1 = E_2 = E_3 = 2$.

To solve this set of ordinary differential equations, the same previously discussed schemes can be applied. BDF2 for example simply takes the form

$$\mathbf{y}^{n+2} - \frac{4}{3}\mathbf{y}^{n+1} + \frac{1}{3}\mathbf{y}^{n} = \frac{2}{3}h\mathbf{f}^{n+2}. \tag{2.5}$$

Just as previously discussed, an iteration algorithm is needed to evaluate each step, as the method is implicit. A system generalisation of Newton's or Chebyshev iteration function are needed, to evaluate the root of the function:

$$\mathbf{F}(\mathbf{y}^{n+2}) = \mathbf{y}^{n+2} - \frac{4}{3}\mathbf{y}^{n+1} + \frac{1}{2}\mathbf{y}^{n} - \frac{2}{3}h\mathbf{f}^{n+2}. \tag{2.6}$$

Newton's iteration function requires to solve the system:

$$\mathrm{J}\boldsymbol{\Delta}\mathbf{x} = -\mathbf{F}, \tag{2.7}$$

at each iteration. Then, the approximate solution is incremented as:

$$\mathbf{y}_{k+1}^{n+2} = \mathbf{y}_{k}^{n+2} + \boldsymbol{\Delta}\mathbf{x}, \tag{2.8}$$

and the iteration is repeated until the set tolerance is reached. Here, $\mathbf{F} = \mathbf{F}(\mathbf{y}^{n+2})$, and $\mathrm{J} = \mathrm{J}(\mathbf{F})$ is the Jacobian of $\mathbf{F}$ evaluated in $\mathbf{y}^{n+2}$. Given that $\mathbf{F}$ is a vector, whose size is equal the size of the IVP, its Jacobian J is a square matrix.

The computational cost of the solving each step thus scales with the square of the size of the IVP.

For the system version of the algorithm, we may choose a more refined criterion for the termination of the iteration. As discussed in [11], if the assumption that the initial guess x_0 is sufficiently close to the root, the *relative nonlinear residual* of the system, $\|\mathbf{F}(x)\|/\|\mathbf{F}(x_0)\|$, is strictly related to the size of the error. We may thus impose the relative nonlinear residual to be within a certain tolerance in order to stop the iteration. Moreover, the stopping criteria are generally specified in terms of an absolute and relative tolerance[12]. A relative tolerance correctly relates the tolerance to the size of the solution, where a purely absolute tolerance may be overly permissive or restrictive. However, relative tolerance obviously becomes meaningless whenever the solution has any roots. Thus, the criterion for the termination of the iteration will allow to specify both an absolute and a relative tolerance, τ_a and τ_r.

$$\|\mathbf{F}(x)\| \leq \tau_r \|\mathbf{F}(x_0)\| + \tau_a$$

Algorithm 1 Algorithm for Newton's iteration function in system version

1: Compute $\mathbf{F}(y^*)$
2: Evaluate $r_0 = \|\mathbf{F}(y^*)\|$
3: **while** $\|\mathbf{F}(y^*)\| \leq \tau_r\, r_0 + \tau_a$ **do**
4: Compute J$(y*)$
5: Solve J$\Delta = -\mathbf{F}$
6: $y^* = y^* + \Delta$
7: Compute $\mathbf{F}(y^*)$
8: **end while**

The main integrator algorithm will therefore be able to call an algorithm for advancing the solution at each consecutive timestep. Using Newton's iteration function, the algorithm will be in the form of 1.

Chebyshev iteration function requires to solve two systems in order to obtain the increment:

$$\begin{aligned} \mathrm{J}\mathbf{w} &= -\mathbf{F}, \\ \mathrm{J}\boldsymbol{\Delta}\mathbf{x} &= -\mathbf{F} - \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathrm{H}\mathbf{w}. \end{aligned} \tag{2.9}$$

Where H is the Hessian of $\mathbf{F}$ evaluated in $\mathbf{y}^{n+2}$. Since H is a third order tensor, the computational cost of the solving each step thus scales with the square of the size of the IVP.

The computational cost of evaluating the Hessian for every iteration for every step may be prohibitive in practice. However, in some cases this can be avoided, as is the focus of this thesis.

The algorithm for advancing the solution using Chebyshev iteration function will be in the form of 2.

Algorithm 2 Algorithm for Chebyshev iteration function in system version

Compute $\mathbf{F}(y^*)$
Evaluate $r_0 = \|\mathbf{F}(y^*)\|$
while $\|\mathbf{F}(y^*)\| \leq \tau_r\, r_0 + \tau_a$ **do**
    Compute $\mathrm{J}(y^*)$
    Compute $\mathrm{H}(y^*)$
    Solve $\mathrm{J}\mathbf{w} = -\mathbf{F}$
    Solve $\mathrm{J}\Delta = -\mathbf{F} - \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathrm{H}\mathbf{w}$
    $y^* = y^* + \Delta$
    Compute $\mathbf{F}(y^*)$
end while

Algorithms 1 and 2 evaluate the Jacobian and the Hessian of the system inside the iteration loop only. For the combustion mechanism that will be analysed in this work, this is not practical, and it is best to compute $\mathbf{F}(y*)$, J and H at the same time, by a function created for this purpose. The algorithms that were implemented is therefore modified in the form shown in 3 and 4.

Algorithm 3 Modified algortihm for Newton's iteration function

Compute $\mathbf{F}(y^*)$ and $\mathrm{J}(y*)$
Evaluate $r_0 = \|\mathbf{F}(y^*)\|$
while $\|\mathbf{F}(y^*)\| \leq \tau_r\, r_0 + \tau_a$ **do**
    Solve $\mathrm{J}\Delta = -\mathbf{F}$
    $y^* = y^* + \Delta$
    Compute $\mathbf{F}(y^*)$ and $\mathrm{J}(y*)$
end while

Algorithm 4 Modified algorithm for Chebyshev iteration function

1: Compute $\mathbf{F}(y^*)$, $\mathrm{J}(y^*)$ and $\mathrm{H}(y^*)$
2: Evaluate $r_0 = \|\mathbf{F}(y^*)\|$
3: **while** $\|\mathbf{F}(y^*)\| \leq \tau_r\, r_0 + \tau_a$ **do**
4: Solve $\mathrm{J}\mathbf{w} = -\mathbf{F}$
5: Solve $\mathrm{J}\Delta = -\mathbf{F} - \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathrm{H}\mathbf{w}$
6: $y^* = y^* + \Delta$
7: Compute $\mathbf{F}(y^*)$, $\mathrm{J}(y^*)$ and $\mathrm{H}(y^*)$
8: **end while**

The sample problem 2.4 was solved using these algorithms. The solution obtained for a stepsize of 2 is shown in figure 2.2, compared to the one obtained using MATLAB's solver.

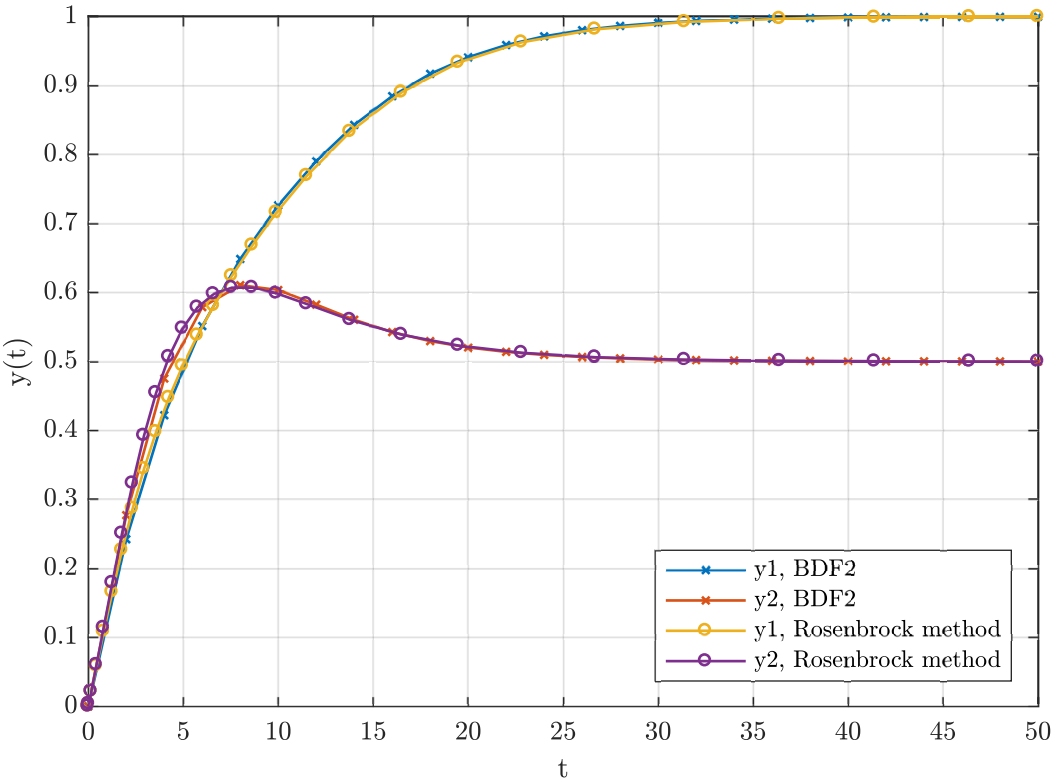

Figure 2.2: Solution of the sample IVP 2.4 with the BDF2 algorithm, compared to MATLAB's Rosenbrock solver.

The number of iterations performed with both iteration functions, for different stepsize values, are compared in figure 2.3. It can be observed once again as the number of iteration performed increases for the steps where the solution changes more rapidly, and that Chebyshev's iteration function generally requires a lower number of iterations. Moreover, it can be seen how the reduced number of iteration becomes increasingly beneficial as the stepsize decreases.

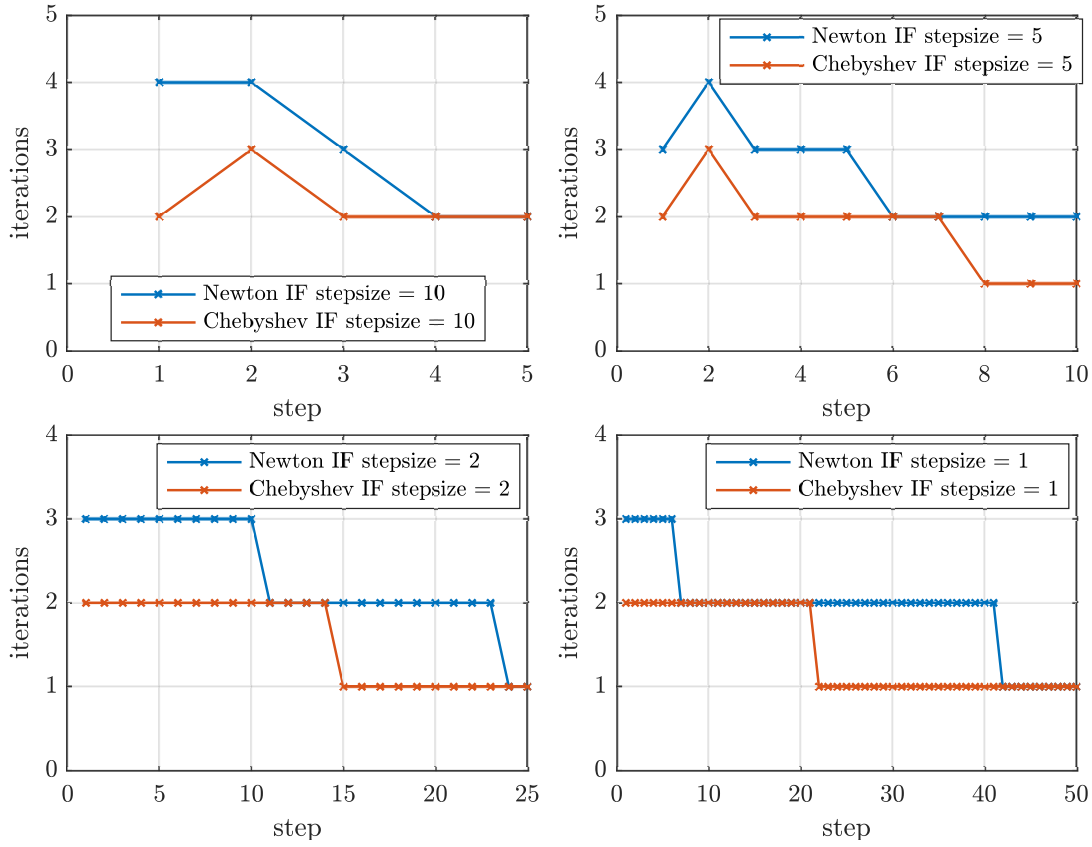

Figure 2.3: Comparison between the number of iterations performed with Newton and Chebyshev iteration function, for different stepsize values. The tolerance parameters are $\tau_r = 1 \times 10^{-7}$ and $\tau_a = 1 \times 10^{-8}$.

Chapter 3

Physical modelling

3.1 Thermodynamical aspects

We now consider the energy conservation equation. For isobaric (constant pressure) processes, it is best to employ the enthalpy equation, that is given by[17]:

$$\rho \frac{\mathrm{D}h}{\mathrm{D}t} = \frac{\mathrm{D}p}{\mathrm{D}t} - \nabla \cdot \mathbf{q} + \mathbf{\Pi}_\nu : \nabla \mathbf{u},$$

where ρ, $\mathbf{u}$, h, p, $\mathbf{q}$ and $\mathbf{\Pi}_\nu$ denote the density, velocity, enthalpy, pressure, heat flux and viscous stress tensor, respectively.

With the previous assumptions, the total enthalpy of the system is conserved.

$$\rho \frac{\mathrm{D}h}{\mathrm{D}t} = 0.$$

Equivalently, in terms of the individual enthalpies of the species, we may write

$$\frac{\partial}{\partial t}\left(\rho \sum_i y_i h_i(T)\right) = 0.$$

For convenience, we can define the total enthalpy of the system as

$$H_0 = \sum_i y_i h_i(T). \tag{3.1}$$

As previously discussed, this quantity is constant in time.

Since only gaseous-phase phenomena is considered in this thesis, the ideal gas law may be applied to rewrite the density, if needed, by

$$p = \rho R^0 T \sum_i \frac{y_i}{w_i},$$

Here, R^0 is the universal gas constant, w_i is the molecular weight of each species, and T is the system's temperature.

We then obtained an equation relating the species' mass fractions and the system's temperature. This equation has to be satisfied at all times in order to respect the thermodynamical constraint that the total enthalpy of the system is conserved.

$$\sum_i y_i h_i(T) = 0.$$

This equation will be appended to a system of ordinary equations corresponding to the rate equations of the chemical species of the considered reaction mechanism. Therefore, we may want to express this relationship as a differential equation as well.

Differentiating with respect to time:

$$\begin{aligned}
\frac{\partial}{\partial t}\left(\sum_i y_i h_i(T)\right) &= 0 \\
\sum_i \left(\frac{\mathrm{d}y_i}{\mathrm{d}t}\, h_i(T) + y_i\, \frac{\partial h_i}{\partial t}\right) &= 0 \\
\sum_i \left(\frac{\mathrm{d}y_i}{\mathrm{d}t}\, h_i(T) + y_i\, \frac{\partial h_i}{\partial T}\, \frac{\partial T}{\partial t}\right) &= 0 \\
\sum_i f_i\, h_i(T) + \frac{\mathrm{d}T}{\mathrm{d}t} \sum_i y_i\, c_{pi} &= 0.
\end{aligned}$$

Thus, introducing an average heat capacity for the whole system: $c_p = \sum_i y_i\, c_{pi}$, we get an equation for the rate of change of temperature:

$$f_T = -\frac{\sum_i f_i\, h_i(T)}{c_p}.$$

This equation will be needed to impose a thermodynamical constraint to the simulation of the combustion mechanism. As it will be discussed in the following sections, the simulation of a reaction mechanism involves the integration of several rate equations, one for each of the chemical species, plus an equation for the rate of change of temperature[15].

This equation is derived from the thermodynamical constraint imposed to the process. In this example, 3.1 simply derives from the conservation of total energy applied to an isobaric process.

3.2 The rate equations

We now want to express the rate equations describing the chemical kinetics of a combustion process. A chemical reaction is a process that leads to the transformation of some chemical species in others[4]. This process happens at a characteristic reaction rate, which is related to the rate at which the concentration of the species involved change in time.

The reaction rate is proportional to the concentration of the species involved, elevated to their respective partial reaction orders, through a reaction constant.

Also, for a generic non-equilibrium reaction, where reactants and products continuously interact with each other, two reaction rates have to be considered. Usually, the reaction

rate related to the transformation of the reactants into products is called the *forward* reaction rate, while the converse is called the *backward* reaction rate.

The reaction rate is dependent on the molar concentration of the species involved, by the equation:

$$\omega_f = k_f \prod_{i=1}^{N_p} [Y_i]^{x_i},$$
$$\omega_b = k_b \prod_{i=N_p+1}^{N} [Y_i]^{x_i}.$$

Here, k_f and k_b are *rate coeeficients*, $[Y_i]$ is the molar concentration of specie i, and the subscripts $i = 1 \dots N_p$ indicate the reactants, while $i = N_p + 1 \dots N$ indicate the products. Moreover, for elementary reactions, the exponents a_i are equal to the stoichiometric coefficient ν_i.

The reaction rate constant depends on various factors that affect the reaction, the most prominent of which is temperature. In particular, this dependance is often described in textbook chemistry by the *Arrhenius equation*:

$$k(T) = AT^n e^{\frac{E}{RT}}$$

where the reaction rate constants are a function of an activation energy, E, temperature, and a pre-exponential factor A.

In general, we may be interested in solving a system of numerous elementary reactions. Considering a system of I elementary reactions involving K different chemical species, these take the form

$$\sum_{k=1}^{K} \nu'_{ki} Y_k \rightleftharpoons \sum_{k=1}^{K} \nu''_{ki} Y_k$$

where ν'_{ki} is the stoichiometric coefficient of chemical specie Y_k in reaction i where the specie is a reactant, while ν''_{ki} is the stoichiometric coefficient of Y_k as a product[10].

Then, the rate of change of a species concentration is obtained from all the reaction rates involving specie k:

$$\dot{\omega_k} = \frac{\mathrm{d}[Y_k]}{\mathrm{d}t} = \sum_{i=1}^{I} \nu_{ki} \omega_i,$$

where

$$\nu_{ki} = \nu''_{ki} - \nu'_{ki}.$$

Here, the reaction rate of reaction i, ω_i, is given by the difference of the forward and backward reaction rates:

$$\omega_i = k_{f_i} \prod_{k=1}^{K} [Y_k]^{\nu'_{ki}} - k_{b_i} \prod_{k=1}^{K} [Y_i]^{\nu''_{ki}}.$$

In this discussion, $[Y_k]$ indicates the concentration of specie k in terms of its molarity. We are interested in solving for the mole fraction y_k:

$$[Y_k] = \rho \frac{y_i}{w_i},$$

where ρ is the system's density and w_k is the molar mass of specie k. Substituting

$$\frac{\mathrm{d}y_k}{\mathrm{d}t} = \frac{w_k}{\rho} \sum_{i=1}^{I} \nu_{ki}\omega_i$$

Where the expression for the systems density can be obtained from the ideal gas law in equation 3.1.

3.3 The Initial Value Problem

Coupling the equations for the concentration of the species with the thermodynamical constraint obtained, we get a set of $N + 1$ equations:

$$f_i = \frac{\mathrm{d}y_i}{\mathrm{d}t} = \frac{w_i}{\rho} \sum_{l=1}^{L} \nu_{il}\omega_{il},$$
$$f_T = -\frac{\sum_i f_i\, h_i(T)}{c_p}.$$

where I is the number of chemical species involved. This set of ordinary differential equations, together with a set of initial conditions for the species' concentration and temperature, forms the initial value problem to be solved. To solve this IVP with a numerical integrator, the integration algorithm must be able to call a function that computes this set of equations $\mathbf{f}$ for a given set of values of the integration variables $\mathbf{y}$. First, the reaction rates of all the reactions of the mechanism have to be computed, as described in the previous subsections. These will be a function of the concentration of the reactants of that reaction and the reaction rate constant. Thus, each ω_l contains temperature dependent terms and others that depend on the species' concentrations:

$$\omega_l = P_l(y)E_l(T).$$

Then, in order to evaluate the rate equation for each specie i, $\mathbf{f_i}$, the reaction rates that contribute to the production or consumption of that specie are summed or subtracted. Finally, the temperature equation is evaluated, as this is a function of all the $\mathbf{f_i}s$. Code A.1 shows an implementation of this function in MATLAB.

In order to apply the numerical methods described in the previous chapter, the Jacobian matrix and the Hessian tensor of the system may have to be computed several times for each timestep, in order to apply the root-finding algorithm. This may be highly challenging computationally. The purpose of this thesis is to investigate wether the exact Hessian may be substituted with the computation of an inexact Hessian to reduce the overall computational cost. On one hand, introducing an error in the computation of the Hessian is expected to increase the number of iteration steps performed by the root finding algorithm at each timestep. On the other hand, using some inexact Hessian may reduce the computational cost of every evaluation significantly. If the error that is consequently introduced is not too large, the overall computational cost of the root-finding algorithm may be reduced.

In a general application, these derivative terms can be computed by a numerical approximation or directly from their analytical expression. In case it is available, the analytical expression generally allows for faster computation; therefore, in this work, the Jacobian entries are computed as follows:

$$
\begin{aligned}
J_{ij} = \frac{\partial f_i}{\partial y_j} &= \frac{w_i}{\rho} \sum_{l=1}^{L} \frac{\partial \omega_{il}}{\partial y_j} && 1 \le i \le N, \\
&&& 1 \le j \le N \\
= \frac{\partial f_i}{\partial T} &= \frac{w_i}{\rho} \sum_{l=1}^{L} \frac{\partial \omega_{il}}{\partial T} + w_i \frac{R^0 \tilde{w}}{p} \sum_{l=1}^{L} \omega_{il} && 1 \le i \le N, \\
&&& j = N \\
= \frac{\partial f_T}{\partial y_j} &= -\frac{\sum_{i=1}^{N} \frac{\partial f_i}{\partial y_j} h_i}{\tilde{c_p}} = -\frac{\sum_{i=1}^{N} J_{ij} h_i}{\tilde{c_p}} && 1 \le j \le N \\
= \frac{\partial f_T}{\partial y_T} &= -\frac{\sum_{i=1}^{N} \left(F_i \, c_{p_i} + J_{i\,N+1} h_i \right)}{\tilde{c_p}}
\end{aligned}
\tag{3.2}
$$

For the computations of the Jacobian entries, the derivative of the system's density with respect to the individual chemical species concentrations is supposed to be constant. As derived in the previous section, equation 3.1 the system's density is a function of the mean molecular weight $\tilde{w}$:

$$
\rho = \frac{p}{R^0 T \frac{1}{\tilde{w}}} = \frac{p}{R^0 T \sum_i \frac{y_i}{w_i}},
$$

and the mean molecular weight itself is a function of each of the specie's mole fraction. However, while these may change dramatically in the combustion process, the former remains almost constant in time. Thus, the system's density can be considered to be a function of temperature only. This assumption is particularly accurate for constant pressure environments; however, because it strongly simplifies the analytical computation of the Jacobian, it is commonly extended to other conditions[14].

The hessian entries are computed as follows:

$$
\begin{aligned}
H_{ijk} = \frac{\partial^2 f_i}{\partial y_j \partial y_k} &= \frac{w_i}{\rho} \sum_{l=1}^{L} \frac{\partial^2 \omega_{il}}{\partial y_j \partial y_k} && \begin{matrix} 1 \le i \le N, \\ 1 \le j \le N, \\ 1 \le k \le N \end{matrix} \\
= \frac{\partial^2 f_i}{\partial y_j \partial T} &= \frac{w_i}{\rho} \sum_{l=1}^{L} \frac{\partial^2 \omega_{il}}{\partial y_j \partial T} + w_i \frac{R^0 \tilde{w}}{p} \sum_{l=1}^{L} \frac{\partial \omega_{il}}{\partial y_j} && \begin{matrix} 1 \le i \le N, \\ 1 \le j \le N, \end{matrix} \\
= \frac{\partial^2 f_i}{\partial y_j \partial T} &= \frac{\partial^2 f_i}{\partial T \partial y_j} && \begin{matrix} 1 \le i \le N, \\ 1 \le j \le N, \end{matrix} \\
= \frac{\partial^2 f_i}{\partial T^2} &= \frac{w_i}{\rho} \sum_{l=1}^{L} \frac{\partial^2 \omega_{il}}{\partial T^2} + 2\, w_i \frac{R^0 \tilde{w}}{p} \sum_{l=1}^{L} \frac{\partial \omega_{il}}{\partial T} && 1 \le i \le N, \\
= \frac{\partial^2 f_T}{\partial y_j \partial y_k} &= -\frac{\sum_{i=1}^{N} \frac{\partial J_{ij}}{\partial y_k} h_i}{\tilde{c_p}} = -\frac{\sum_{i=1}^{N} H_{ijk} h_i}{\tilde{c_p}} && \begin{matrix} 1 \le j \le N, \\ 1 \le k \le N \end{matrix} \\
= \frac{\partial^2 f_T}{\partial y_j \partial T} &= -\frac{\sum_{i=1}^{N} \left(H_{ijN+1}\, h_i + J_{ij}\, c_{p_i} \right)}{\tilde{c_p}} && 1 \le j \le N, \\
= \frac{\partial^2 f_T}{\partial T \partial y_j} &= \frac{\partial^2 f_T}{\partial y_j \partial T} && 1 \le j \le N, \\
= \frac{\partial^2 f_T}{\partial T^2} &= -\frac{\sum_{i=1}^{N} \left(2 J_{iN+1}\, c_{p_i} + H_{iN+1N+1}\, h_i \right)}{\tilde{c_p}} &&
\end{aligned}
\tag{3.3}
$$

In order to solve the IVP by the BDF2 algorithms presented in chapter 2, the integration algorithm must be able to call a function that computes the Jacobian and Hessian of the system of ODEs, for a given set of values of the integration variables $\mathbf{y}$. For this work, a fully analytical computation of J and H was implemented. Code A.2 shows an implementation in MATLAB of a function that evaluates them, together with the functions $\mathbf{f}$.

Chapter 4

A model for Hydrogen Combustion

In order to demonstrate the application of the possibilities discussed in the previous chapter, a simplistic combustion model is introduced in this chapter. The reaction mechanism is based on the H_2 / O_2 reaction mechanism developed by [5]. It is a detailed mechanism where the oxydation of hydrogen is broken down in 19 elementary reaction involving 10 chemical species, including termolecular reactions.

H_2 / O_2 chain reactions

$$H + O_2 \rightleftarrows O + OH \tag{4.1}$$
$$O + H_2 \rightleftarrows H + OH \tag{4.2}$$
$$OH + H_2 \rightleftarrows H + H_2O \tag{4.3}$$
$$O + H_2O \rightleftarrows OH + OH \tag{4.4}$$

H_2 / O_2 dissociation/recombination

$$H_2 + M \rightleftarrows H + H + M \tag{4.5}$$
$$O + O + M \rightleftarrows O_2 + M \tag{4.6}$$
$$O + H + M \rightleftarrows OH + M \tag{4.7}$$
$$O + OH + M \rightleftarrows H_2O + M \tag{4.8}$$

Formation and consumption of HO_2

$$H + O_2 + M \rightleftarrows HO_2 + M \tag{4.9}$$
$$H + O_2 \rightleftarrows HO_2 \tag{4.10}$$
$$HO_2 + H \rightleftarrows H_2 + O_2 \tag{4.11}$$
$$HO_2 + H \rightleftarrows OH + OH \tag{4.12}$$
$$HO_2 + O \rightleftarrows OH + O_2 \tag{4.13}$$
$$HO_2 + OH \rightleftarrows H_2O + O_2 \tag{4.14}$$

Formation and consumption of H_2O_2

$$HO_2 + HO_2 \rightleftarrows H_2O_2 + O_2 \tag{4.15}$$
$$H_2O_2 + M \rightleftarrows OH + OH + M \tag{4.16}$$
$$H_2O_2 + H \rightleftarrows H_2O + OH \tag{4.17}$$
$$H_2O_2 + H \rightleftarrows H_2 + HO_2 \tag{4.18}$$
$$H_2O_2 + O \rightleftarrows OH + HO_2 \tag{4.19}$$
$$H_2O_2 + OH \rightleftarrows H_2O + HO_2 \tag{4.20}$$

Each of the chemical specie is characterised by its own molecular weight, specific heat c_p and enthalpy h. Given that the process only involves gaseous phase elements, these two are simple functions of temperature. For simplicity, the temperature dependence of the specific heat was ignored. Enthalpy was simply evaluated as

$$h_i(T) = \Delta_f h_i^0 + \int_{298K}^{T} c_{pi} \mathrm{d}T,$$

where $\Delta_f h^0$ is the standard enthalpy of formation.

The data for the chemical species is reported in table 4.1.

A more detailed modelling would involve the temperature dependance of the specific heat. The most commonly used technique for computer simulation is to refer to a formulation of the thermodynamic properties properties of each species, both c_p and h, by polynomials that allow their direct calculation for any temperature[8][3]. These polynomials are formulated with a number of coefficients, for several temperature ranges.

Specie	$w_i \left(\frac{\mathrm{g}}{\mathrm{mol}}\right)$	$c_p \left(\frac{\mathrm{cal}}{\mathrm{mol\,K}}\right)$	$\Delta_f h^0 \left(\frac{\mathrm{kcal}}{\mathrm{mol}}\right)$
H	1.008	4.97	52.10
H_2	2.016	6.89	0.00
O	15.999	5.24	59.55
O_2	31.998	7.02	0.00
OH	17.007	7.14	9.40
H_2O	18.015	8.03	-57.80
N_2	28.014	6.96	0.00
HO_2	33.006	8.34	3.00
H_2O_2	34.014	10.13	-32.48
Ar	39.948	4.97	0.00

Table 4.1: Chemical species data

Given that the main focus of this work is computational efficiency, rather than modelling accuracy, the mechanism was modified to introduce several simplifications, in order to reduce the complexity of the code. In the three body reactions, the effective concentration of the third body is simply evaluated the sum of the concentrations of all the other species,

ignoring collision efficiencies. For some of the reactions, the reaction rate is also dependant on pressure. In this model, this is considered by using the *Troe* formulation. For the purpose of this thesis, the pressure dependance of the rate coefficients is also not taken into account, ignoring the Troe parametrisation of the original mechanism.

The rate coefficients are evaluated with the Arrhenius extended equation:

$$k = AT^n \exp(-E_a/RT).$$

The parameters, obtained from [5], are reported in table 4.2

	Forward			Backward		
Reaction	A	n	E_a	A	n	E_a
1	1.915E+14	0.00	1.644E+04	5.481E+11	0.39	-2.930E+02
2	5.080E+04	2.67	6.292E+03	2.667E+04	2.65	4.880E+03
3	2.160E+08	1.51	3.430E+03	2.298E+09	1.40	1.832E+04
4	2.970E+06	2.02	1.340E+04	1.465E+05	2.11	-2.904E+03
5	4.577E+19	-1.40	1.044E+05	1.146E+20	-1.68	8.200E+02
6	4.515E+17	-0.64	1.189E+05	6.165E+15	-0.50	0.000E+00
7	9.880E+17	-0.74	1.021E+05	4.714E+18	-1.00	0.000E+00
8	1.912E+23	-1.83	1.185E+05	4.500E+22	-2.00	0.000E+00
9	1.475E+12	0.60	0.000E+00	3.090E+12	0.53	4.887E+04
10	1.660E+13	0.00	8.230E+02	3.164E+12	0.35	5.551E+04
11	7.079E+13	0.00	2.950E+02	2.027E+10	0.72	3.684E+04
12	3.250E+13	0.00	0.000E+00	3.252E+12	0.33	5.328E+04
13	2.890E+13	0.00	-4.970E+02	5.861E+13	0.24	6.908E+04
14	4.634E+16	-0.35	5.067E+04	4.200E+14	0.00	1.198E+04
15	2.951E+14	0.00	4.843E+04	3.656E+08	1.14	-2.584E+03
16	2.410E+13	0.00	3.970E+03	1.269E+08	1.31	7.141E+04
17	6.025E+13	0.00	7.950E+03	6.025E+13	0.00	7.950E+03
18	9.550E+06	2.00	3.970E+03	8.660E+03	2.68	1.856E+04
19	1.000E+12	0.00	0.000E+00	1.838E+10	0.59	3.089E+04

Table 4.2: Reaction rate data

4.1 Simulation using MATLAB solver

In order to set a reference for the simulation of the kinetic mechanism, the MATLAB ode solver was used. The main issue is to construct a set of functions corresponding to the kinetic system, describing the rate of change of concentration of all the chemical species, completed with an equation for the rate of change of temperature.

The rate of change of temperature is given by the constraints imposed to the process. In this example, and adiabatic, constant pressure combustion process was considered. Therefore, as discussed in section 3, the total enhalpy of the system must remain constant.

The system is therefore composed of 10 equations of the form:

$$f_i = \frac{\mathrm{d}y_i}{\mathrm{d}t} = \frac{w_i}{\rho} \sum_l \nu_{il} \dot{\omega}_{il},$$

and one of the form:

$$f_T = -\frac{\sum_i f_i \, h_i(T)}{c_p}.$$

The solution of the IVP requires 11 initial conditions, that is the initial concentration of the chemical species and the initial temperature. The initial composition of the mixture was set to be 50% hydrogen and 50% air, while the initial temperature was set to 1200 K. Table 4.3 shows the exact set of initial conditions.

Specie	y_0	Specie	y_0
H	0	H_2O	0
H_2	0.5	N_2	0.105
O	0	HO_2	0
O_2	0.39	H_2O_2	0
OH	0	Ar	0.005

Table 4.3: Initial concentrations for the chemical species. The initial temperature was set to 1200 K

4.2 Comparison between the iteration functions

The BDF2 algorithm described in chapter 2 can now be used to solve the system. Figure 4.2 compares the number of iterations performed using Newton and Chebyshev iteration function, for different stepsize. It can be observed that there a few occurrences where Chebyshev's IF requires more iterations than Newton's. This may be due to the assumptions made for computing the entries of the Hessian, where the density is considered to be constant with respect to the specie's concentration. Nontheless, it generally requires less iterations overall.

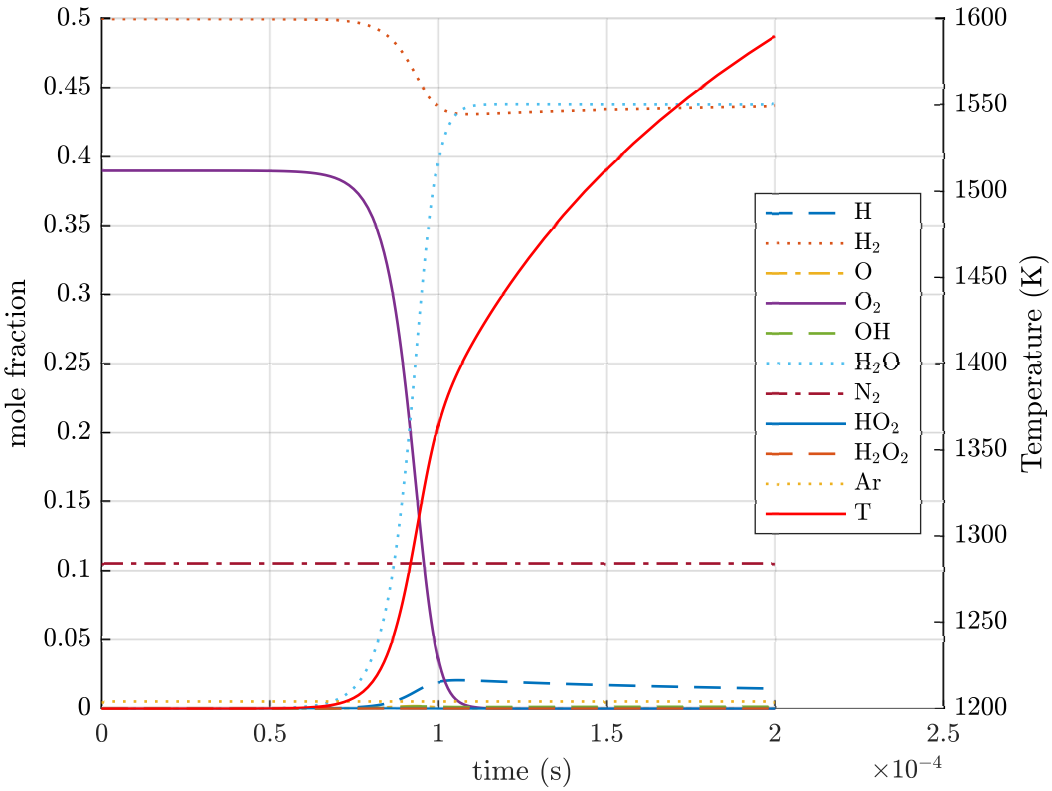

Figure 4.1: The solution of the proposed problem.

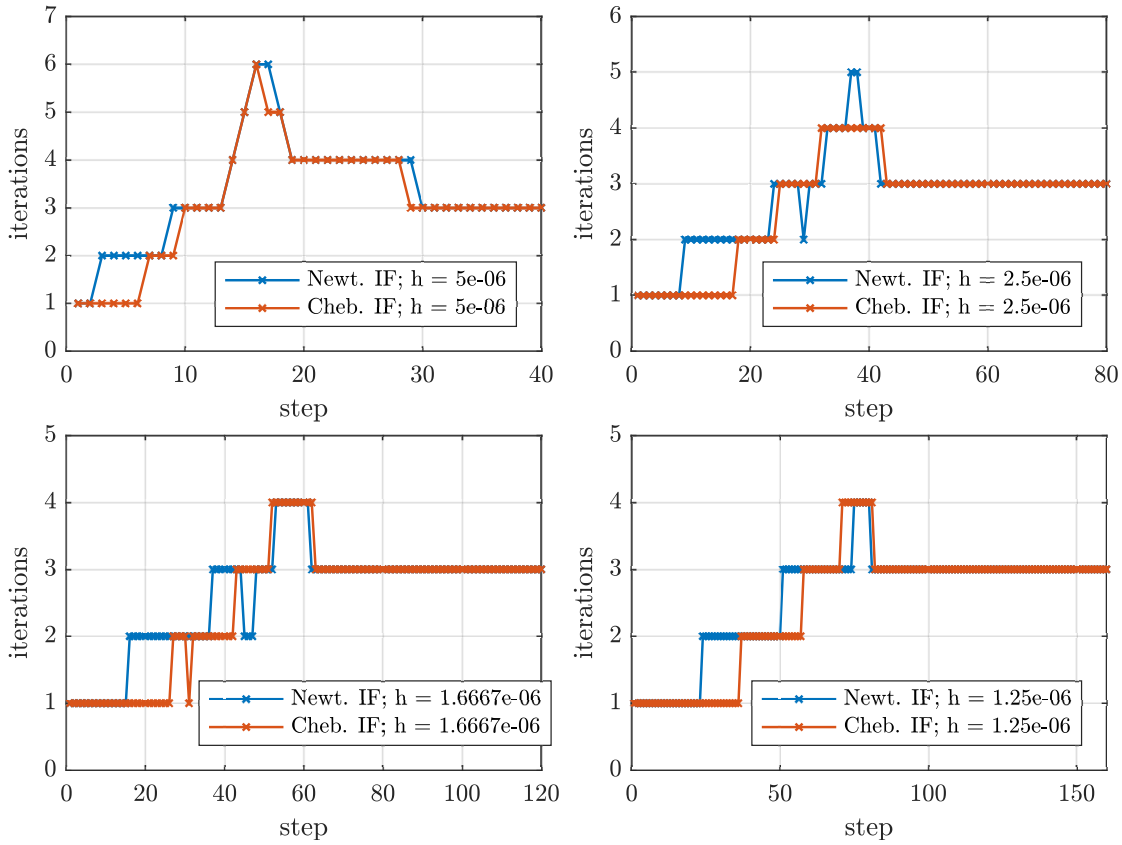

Figure 4.2: Number of iterations performed with Newton and Chebyshev iteration function.

Chapter 5

Using an inexact Hessian

In the previous section, the effect of using a higher rate of convergence iteration function for the solution of a combustion mechanism was presented. As expected, the implementation of a third-order method lead to a reduced number of iterations with respect to a second-order method. However, the total running time may not decrease; in fact, because the third-order method also requires the computations of the Hessian of the system of ODEs, the running time for this specific problem is more than doubled.

Therefore, the evaluation of rigorously exact Hessian matrix is too expensive for the method to be efficient with Chebyshev's iteration function. However, a full exact Hessian may not be required to reduce the number of iterations. For example, since the system's temperature doesn't change by a great extent during the iteration, some entries may be computed only once for every timestep.

5.1 Freezing the temperature terms

As shown in chapter 3, the chemical kinetics of the reaction mechanism is described by a set of coupled ODEs describing the rate of change of the chemical species concentration plus one describing the rate of change of temperature. This last equation is usually derived from physical constraint imposed on the system, for example the conservation of energy; therefore, it depends on the thermodynamical properties of all the chemical species involved. For this reason, a relatively large computational effort is required to compute its derivatives is required. In particular, equations 3.3 shows how the Hessian entries relative to the temperature equation can be complex even for a simple mechanism where isobaric conditions were imposed.

Given that the temperature equation depends on all of the chemical species' enthalpies, its value is only relatively affected by a change in one of the species' concentration. Moreover, as previously mentioned, the temperature does not vary much during the course of each iteration. Consequently, the enthalpy of each specie doesn't change significantly.

As a result, the Hessian entries relative to this equation do not vary significantly during the course of an iteration. A way to speed up the algorithm is to freeze these entries, computing them only once at the beginning of each iteration. The algorithm for the iteration is therefore modified as in 5.

When applied to the combustion mechanism considered in this work, this algorithm requires exactly the same number of iterations for a wide range of stepsize and tolerance values, thus proving that the Hessian entries related to the temperature equation do not change significantly in an iteration.

Algorithm 5 Modified algorithm for Chebyshev iteration function, freezing the temperature terms of the Hessian. N is the number of chemical species.

1: Compute $\mathbf{F}(y^*)$, $\mathrm{J}(y^*)$ and $\mathrm{H}_{\mathrm{ijk}}(y^*)$, for $1 \leq i \leq N+1$
2: Evaluate $r_0 = \|\mathbf{F}(y^*)\|$
3: **while** $\|\mathbf{F}(y^*)\| \leq \tau_r\, r_0 + \tau_a$ **do**
4: Solve $\mathrm{J}\mathbf{w} = -\mathbf{F}$
5: Solve $\mathrm{J}\Delta = -\mathbf{F} - \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathrm{H}\mathbf{w}$
6: $y^* = y^* + \Delta$
7: Compute $\mathbf{F}(y^*)$, $\mathrm{J}(y^*)$ and $\mathrm{H}_{\mathrm{ijk}}(y^*)$, for $1 \leq i \leq N$
8: **end while**

5.2 Freezing the full Hessian

We saw how freezing the temperature terms of the Hessian for every timestep doesn't affect significantly the number of iterations, thus allowing a small decrease in computation time.

Indeed, we may push this even further and freeze the full Hessian for every timestep. While it is true that the concentration of the chemical species can change steeply at some point of the integration, the Hessian terms depend on derivatives of the second order of several reaction rates, with respect to two reactants. Thus, steep changes in the Hessian terms from one iteration to the other are unlikely to happen.

Freezing the full Hessian at every timestep allow to fully take its out of the iteration loop, as shown in algorithm 6. Provided that this does not increase significantly the number of iterations performed, this produces an obvious improvement of the computation time.

As figure 5.1 shows, this algorithm once again does not increase the number of iterations performed with respect to the exact algorithm where the Hessian is re-evaluated for every iteration. Thus, a strong reduction in computation time is obtained.

Algorithm 6 Modified algorithm for Chebyshev iteration function, freezing the full Hessian.

1: Compute $\mathbf{F}(y^*)$, $\mathrm{J}(y^*)$ and $\mathrm{H}(y^*)$
2: Evaluate $r_0 = \|\mathbf{F}(y^*)\|$
3: **while** $\|\mathbf{F}(y^*)\| \leq \tau_r\, r_0 + \tau_a$ **do**
4: Solve $\mathrm{J}\mathbf{w} = -\mathbf{F}$
5: Solve $\mathrm{J}\Delta = -\mathbf{F} - \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathrm{H}\mathbf{w}$
6: $y^* = y^* + \Delta$
7: Compute $\mathbf{F}(y^*)$ and $\mathrm{J}(y^*)$.
8: **end while**

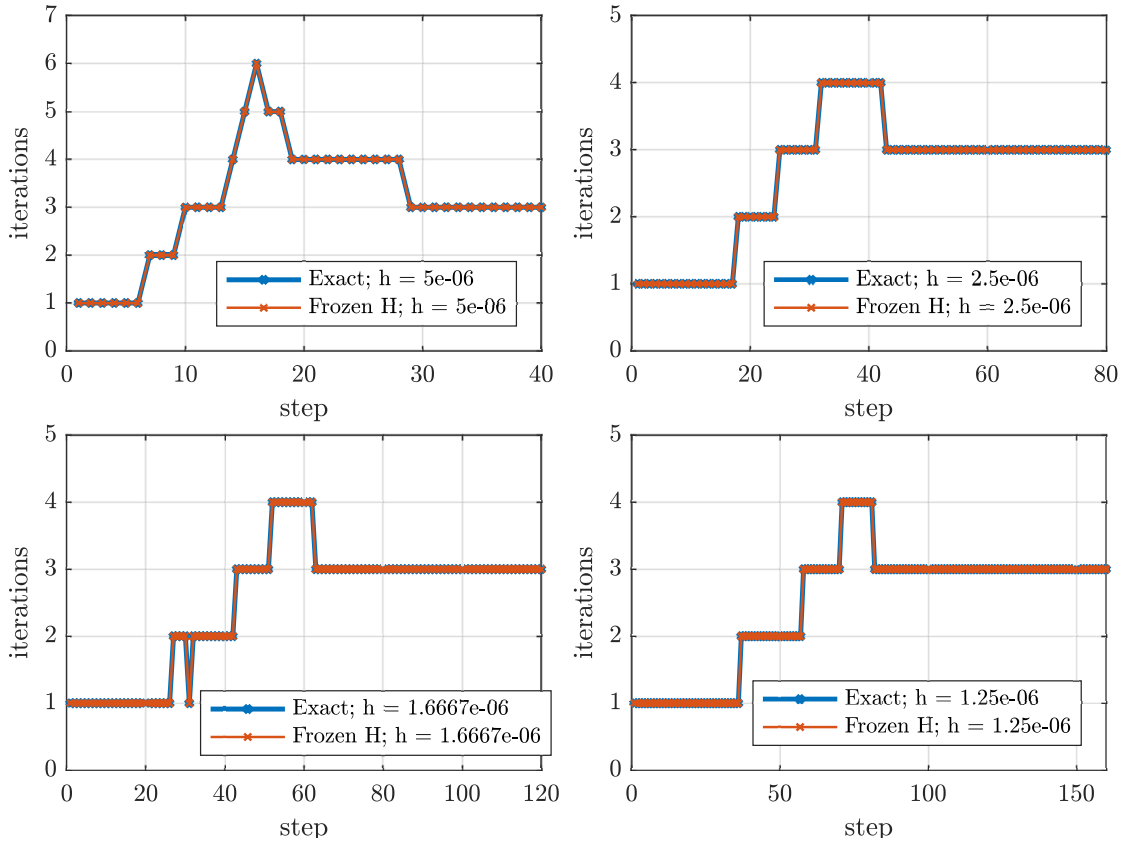

Figure 5.1: Number of iterations performed with the exact Hessian reevaluated at every iteration and with an frozen Hessian, computed only once per timestep at the beginning of the iteration.

5.3 Freezing the temperature

Until now, we tried to avoid recomputing some or all the terms of the Hessian during the course of an iteration, performing the computation at the beginning of the iteration only. An even stronger assumption may be made if we look closely at how the temperature changes from one iteration to the next. Even for small timesteps, this changes are relatively small, due to the fact that the temperature profile, as can be seen in figure 4.2, is quite smooth after the ignition delay.

For this reason, although we certainly cannot assume that the effect of temperature on the Hessian terms is negligible, we may suppose that the error that is introduced does not have a big impact on the iteration method.

If we introduce this assumption, many entries of the Hessian may be left null.

$$H_{ijk} = 0 \qquad \text{for } i = N \vee j = N \vee k = N.$$

These terms, whose analytical expression is reported in table 3.3, are quite expensive to compute. However, if implementing this solution for the integration of the combustion mechanism analysed in this work, the number of iterations performed by the iteration function is almost unchanged, as figure 5.2 shows.

Overall, the amount of computations saved produces a strong reduction in computation time, because the number of iterations performed remains almost the same. It remains to investigate for which combustion mechanisms this remains true.

Finally, another effect of not computing many entries is that the Hessian tensor has far less non zero elements. In other words, the tensor will be *sparser*. As discussed in the next chapter, the *sparsity* of the tensor can be taken advantage of, if specific storage structures and algorithms are implemented, to reduce memory requirements and computation times.

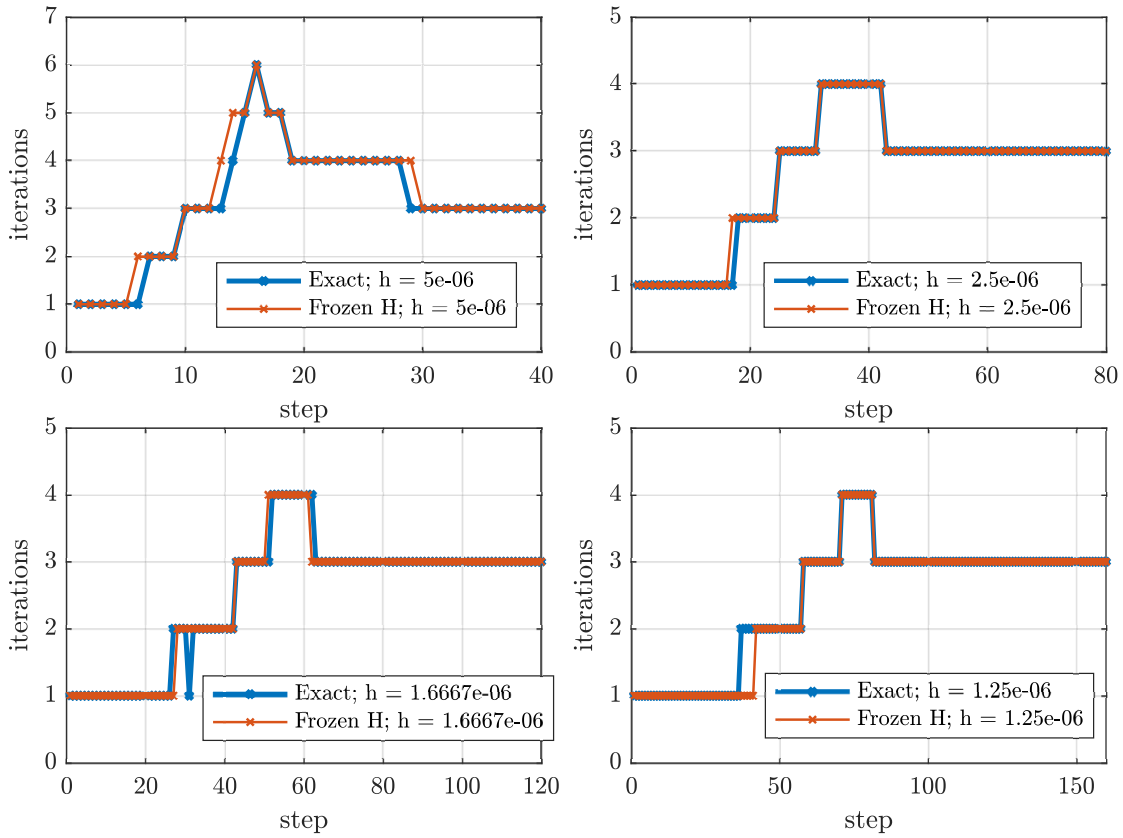

Figure 5.2: Number of iterations performed with the exact Hessian reevaluated at every iteration and with an incomplete Hessian, freezing the temperature at every timestep and computed only once per timestep.

Chapter 6

Sparse Storage

In the previous sections, a requirement for storing large amounts of data in matrix and tensor structures for the application of implicit methods for the numerical integration of ODEs emerged. These structures contain the derivatives of the functions to integrate with respect to each of the integration variables; for the applications discussed in this work, these are the species concentrations and the system's temperature. As the complexity of the combustion mechanism increases, so does the size of the IVP and therefore the size of these data structures.

However, not all of the entries of these structures, that is the Jacobian and the Hessian of the system, are relevant. In particular, the terms relative to derivatives of the rate of change of the species concentration may often be null. Consider the expression for the reaction rates 3.2. Considering elementary reactions, no more than three different reactants and three different products are usually involved[14]. Thus, the stoichiometric coefficients ν_{ki} are mostly null, and the more the null elements, the more loosely coupled the system of equations is. Therefore, these Jacobian and Hessian of the chemical kinetics of a combustion mechanism are typically *sparse*.

A *sparse* structure is a structure where many or most of the elements are null; its sparsity is related to how loosely coupled the system of equations is. By contrast, a structure where most of the elements are not null is called *dense*. The sparsity of these structures may be taken advantage of to reduce both the computer memory required for their storage and the computation time required for some operations[9].

If we consider a matrix $M \in \mathbb{R}^{m \times n}$, storing it as a dense matrix requires a storage space proportional to mn, while storing it as a sparse tensor requires a space proportional to the number of non zero elements. Moreover, any operation performed on a sparse matrix stored as a dense structure means that a lot of time is wasted on unnecessary operations, such as zero-adds; in general, the computational complexity of basic operations is proportional to mn[9], while a specific algorithms designed for sparse structures generally require a reduced computational complexity, proportional to the number of nonzeros in the best cases.

The same considerations can be applied to tensors: for example, a dense third order tensor $T \in \mathbb{R}^{m \times n \times p}$ requires a storage space proportional to mnp. If we consider complex reaction mechanism, where hundreds of elementary reactions can be involved, it is evident that the amount of memory required may become extremely burdening for many workstations. For these reasons, it may be convenient to resort to sparse store storage for

the Jacobian and Hessian of the system for the application of Newton's or Chebyshev's iteration method.

6.1 Implementation

In order to take advantage of the sparsity of the Jacobian and Hessian of the system, the computation of their entries must be performed with specifically designed algorithms, since direct access to a sparse structure entries by their indexes is computationally inefficient, because of the way they are stored.

Several methods exist to implement sparse tensor storage. The software used for this work, the *MATLAB Tensor Toolbox*[2], resorts to a *coordinate storage format*. Given a tensor with nnz nonzeros, the nonzero values stored in an array of length nnz, and the corresponding indexes in the rows of a matrix. Entries with duplicate subscripts are summed when the tensor is assembled.

Preallocating an empty sparse structure and then modifying its entries by direct indexing would be extremely costly. The Jacobian and Hessian entries therefore have to be computed with loops that create new entries. Therefore, a specific algorithm has to be implemented.

The Jacobian and Hessian entries are created in coordinate format as shown in the appendix A.4.

Then, the iteration function was applied using the `ttv` function of the *MATLAB Tensor Toolbox*, that implements tensor *n-mode multiplication* for sparse tensors[1]. Formally, the n-mode product of a tensor $\mathrm{T} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_n}$ times a vector $\mathbf{v} \in \mathbb{R}^{I_n}$ is a tensor of size $I_1 \times \cdots \times I_{n-1} \times I_{n+1} \times \cdots \times I_N$, whose entries are given by:

$$(\mathrm{T} \bar{\times}_n \mathbf{v})_{i_1 \ldots i_{n-1} i_{n+1} \ldots i_N} = \sum_{i_n=1}^{I_n} \mathrm{T}_{i_1 \ldots i_N} \, v_{i_n}.$$

Thus the product $\mathbf{w}^{\mathrm{T}}\mathrm{H}\mathbf{w}$ corresponds to:

$$(\mathrm{T} \bar{\times}_2 \mathbf{w}) \, \bar{\times}_3 \mathbf{w}.$$

The algorithm thus implemented may reduce the computation time of the integration, provided that the Hessian and Jacobian are sparse enough. For the combustion mechanism examined in this work, sparse storage did not produce significant improvements in terms of execution time, although it allows to reduce the memory storage requirement.

However, the sparsity of these structures scales as the size of the problem increases. It is expected that, for large reaction mechanisms with thousands of reaction equations, the benefits of using algorithms specifically designed for sparse structures may become sensible.

Chapter 7

Conclusions

Several issues regarding the numerical integration of the rate equations of a combustion mechanism were presented, and a stiff integrator, based on the implicit integration scheme BDF2 was introduced using Newton's iteration method, which requires the computation of the Jacobian matrix of the system of differential equations.

Then, an iteration method with a higher rate of convergence was introduced, and the resulting number of iterations were compared. As expected, while this method produces less iterations, the necessity to evaluate the Hessian as well as the Jacobian of the system of ODEs causes the total computation time to increase. Moreover, Chebyshev's iteration function requires to solve twice as many linear system solutions with respect to Newton's iteration function. Therefore, the former is only advantageous if the high rate of convergence can be exploited.

Several simplifications in the algorithm were therefore introduced. First, the computation of the Hessian was taken out of the iteration loop, and it is observed that this does not affect the convergence of the method significantly. Then, the temperature derivatives of the Hessian, that are computationally costly to compute, were excluded and once again the convergence of the method is practically unchanged.

Finally, sparse storage for the Jacobian and the Hessian was introduced, but the sparsity of the structures for the combustion mechanism examined in this work is not sufficient to compensate for the increase linear solutions.

A more sophisticated iteration function may be needed to exploit the higher convergence rate. Even then, the benefits may become evident only for particular conditions; for example, large time steps would produce the largest difference in the number of iterations performed between the two iteration functions.

Further development of this work would include an implementation of the techniques presented here to a large scale reaction mechanism, were it is expected that the reduction of number of iterations given by Chebyshev's iteration function becomes far larger and the sparsity of the Jacobian and Hessian far more prominent.

Appendix A

MATLAB code

In this thesis, techniques for the integration of coupled systems of ODEs, specifically for the solution of the rate equations of the chemical species involved in a combustion mechanism, were described. To solve the IVP, formed by the rate equations, the temperature equation and a set of initial conditions, any numerical integrator requires a function evaluating the ODEs for a given set of values of the integration variables $\mathbf{y}$ and time. The code A.1 is a function that does this. As described in chapter 3, first it evaluates the reaction rates of all the reactions of the mechanism; then, the rate equations for each specie are evaluated by summing the reaction rates that contribute to the consumption or production of that specie. Finally, the temperature equation is evaluated.

Listing A.1: The function evaluating the ODEs for the species concentration for the hydrogen combustion mechanism.

```
function F = odefun(t, y, reaction, specie)
% Evaluates f (the system of equations to be integrated), for a
% given y (values of the integration variables).

%% Set thermodynamics constants, initial concentrations and temperature and
    evaluate enthalpies
p = 1e5; %Pa
Rgas = 8.3144598; % Js / K mol
nspecies = size(specie,2)-1;
nT = nspecies+1;
T = y(end);
y = y(1:end-1);

h = zeros(nspecies,1);
for i=1:nspecies
    h(i) = enth(specie(i), T);
end

meanwght = 0;
for i=1:nspecies
```

```
    meanwght = meanwght + y(i)/specie(i).w;
end

y = [y; specie(end).w*meanwght];

rho = p/(Rgas*T*meanwght)*1e-6; % g/cm3

% Evaluate reaction rates
nreactions = size(reaction,2);
omega = zeros(nreactions,1);
for i = 1:nreactions
    cprod(1:3) = 1;
    for ir = 1:reaction(i).nreact
        isa = reaction(i).reactants(ir);
        cprod(ir) = y(isa) * rho/specie(isa).w; %(mol/cm3)^2 or (mol/cm3)^3
    end
    c = reaction(i).A * T^reaction(i).n * exp(-reaction(i).Ta/T);
    omega(i) = c * cprod(1) * cprod(2) * cprod(3); %(mol/cm3)
end

% Sum reaction rates
Fs = zeros(nT,1);
for i = 1:nreactions
    for ir=1:reaction(i).nreact
        isa = reaction(i).reactants(ir);
        if isa ~= 11
            Fs(isa) = Fs(isa) - omega(i);
        end
    end
    for ir=1:reaction(i).nprod
        isa = reaction(i).products(ir);
        if isa ~= 11
            Fs(isa) = Fs(isa) + omega(i);
        end
    end
end

% Evaluate rate equations and temperature equation
F = zeros(nT,1);
meancp = 0;
for i=1:nspecies
    meancp = meancp + y(i) * specie(i).cp;
    F(i) = Fs(i) * specie(i).w / rho;
    F(end) = F(end) - F(i)*h(i);
end
F(end) = F(end) / meancp;
```

```
66
67 end
68
69 function enth = enth(specie, T) % Js/kg
70 % evaluate enthalpy of a specie as a function of temperature
71     enth = specie.hf + (specie.cp)*(T-298);
72 end
```

The code A.2 is the BDF2 algorithm used to integrate the rate equations for the combustion mechanism presented in chapter 4. It requires several inputs:

- ...

Listing A.2: The BDF2 algorithm used to integrate the rate equations of the combustion mechanism for Hydrogen.

```
1  function [t, y] = BDF2(odefun, reaction, specie, order, y0, nsteps, tol,
       tend)
2  %% [t, y] = BDF2(odefun, reaction, specie, order, y0, nsteps, tol, tend)
3  % *odefun* is a function returning f (the system of equatins to be
       integrated),
4  % Jf (the Jacobian of the system) and Hf (the Hessian of the system),
5  % for a given set of values of the integration variables. If order is set
       to 2
6  % (Newton's IF), Hf is not required.
7  %
8  % *reaction* is a structure containing information about the
9  % reactants, the products and the Arrhenius equation parameters for the
10 % reactions of the mechanism
11 %
12 % *specie* is a structure containing information thermochemical data of the
13 % species involved in the mechanism, as well as initial concentrations
14 % *y0* is the initial condition
15 %
16 % *nsteps* is the number of steps
17 % *tol = [tr, ta]* , where *tr* and *ta* are the relative and absolute
       error
18 % tolerances
19 % *tend* is the simulation end time
20
21 %% Set end time, inital conditions, number of time steps, tolerance
22 if order ~= 2 && order ~= 3
23     error('Order must be either 2 or 3')
24 end
25
26 systemDimension = size(y0,2);
27
28 y=zeros(systemDimension,nsteps);
```

```
y(:,1) = y0;
h = tend/nsteps;
t = 0:h:tend;

tr = tol(1);
ta = tol(2); %tolerance for convergence
imax = 100; %number of maximum iterations

%% Bootstrapping
% BWD1 atep to evaluate intermediate-step temporary value
Y = y(:,1); %first guess

% Evaluate F(Y)
[f, Jf, Hf] = odefun(Y, reaction, specie);
F = Y - y(:,1) - h/2 * f;
r0 = norm(F);

for k=1:imax
    % Evaluate the Jacobian J(Y)
    J = eye(systemDimension) - h/2 * Jf;
    H = -h/2*Hf;

    % Advance Y
    w = linsolve(J,-F);
    wHw = zeros(systemDimension,1);
    for hi = 1:systemDimension
        for hj = 1:systemDimension
            for hk = 1:systemDimension
                wHw(hi,1) = wHw(hi,1) + H(hi,hj,hk)*w(hj)*w(hk);
            end
        end
    end
    D = linsolve(J, -F-1/2*wHw);
    Y = Y+D;

    % Re-evaluate F(Y) for tolerance criterion check and possible next
    % iteration
    [f, Jf, Hf] = odefun(Y, reaction, specie);
    F = Y - y(:,1) - h/2 * f;
    % Tolerance check
    if norm(F) < tr*r0 + ta
        iterations(2) = k;
        break
    end
    if k==imax
        warning('maximum number of iterations reached')
```

```
    end
end

ytemp = Y; % intermediate-step value, y^(1/2)

% BDF2 step to evaluate first step of the solution
% Evaluate F(Y)
[f, Jf, Hf] = odefun(Y, reaction, specie);
F = Y - 4/3*ytemp + 1/3*y(:,1) - 2/3*h/2 * f;
r0 = norm(F);

for k=1:imax
    % Evaluate the Jacobian J(Y)
    J = eye(systemDimension) -2/3*h/2 * Jf;
    H = -2/3*h/2*Hf;

    % Advance Y
    w = linsolve(J,-F);
    wHw = zeros(systemDimension,1);
    for hi = 1:systemDimension
        for hj = 1:systemDimension
            for hk = 1:systemDimension
                wHw(hi,1) = wHw(hi,1) + H(hi,hj,hk)*w(hj)*w(hk);
            end
        end
    end
    D = linsolve(J, -F-1/2*wHw);
    Y = Y+D;

    % Re-evaluate F(Y) for tolerance criterion check and possible next
    % iteration
    [f, Jf, Hf] = odefun(Y, reaction, specie);
    F = Y - 4/3*ytemp + 1/3*y(:,1) - 2/3*h/2 * f;
    % Tolerance check
    if norm(F) < tr*r0 + ta
        iterations(2) = k;
        break
    end
    if k==imax
        warning('maximum number of iterations reached')
    end
end
y(:,2) = Y;

%% Main run with BDF2
for n=1:(nsteps-1)
```

```
121     % Evaluate F(Y)
122     [f, Jf, Hf] = odefun(Y, reaction, specie);
123     F = Y - 4/3*y(:,n+1) + 1/3*y(:,n) - 2/3*h * f;
124     r0 = norm(F);
125 
126     for k=1:imax
127         % Evaluate the Jacobian J(Y)
128         J = eye(systemDimension) - 2/3*h*Jf;
129         H = -2/3*h*Hf;
130 
131         % Advance Y
132         w = linsolve(J,-F);
133         if order == 3
134             wHw = zeros(systemDimension,1);
135             for hi = 1:systemDimension
136                 for hj = 1:systemDimension
137                     for hk = 1:systemDimension
138                         wHw(hi,1) = wHw(hi,1) + H(hi,hj,hk)*w(hj)*w(hk);
139                     end
140                 end
141             end
142         end
143         D = linsolve(J, -F-1/2*wHw);
144         Y = Y+D;
145 
146         % Re-evaluate F(Y) for tolerance criterion check and possible next
147         % iteration
148         [f, Jf, Hf] = odefun(Y, reaction, specie);
149         F = Y - 4/3*y(:,n+1) + 1/3*y(:,n) - 2/3*h * f;
150         % Tolerance check
151         if norm(F) < tr*r0 + ta
152             iterations(n+2) = k;
153             break
154         end
155         if k==imax
156             warning('maximum number of iterations reached')
157         end
158     end
159     y(:,n+2) = Y;
160 end
161 
162 end
```

Code A.1 was used for integrating using the library MATLAB integrators. The BDF2 algorithm developed in this work also requires the Jacobian and Hessian of the system, depending on the iteration function used. Codes A.3 and A.4 show the functions that do

this, in dense and sparse format.

Listing A.3: Function evaluating the ODEs, as well as the Jacobian and Hessian.

```
function [F,J,H] = F_cheb(y, reaction, specie)
% Evaluates f (the system of equations to be integrated),
% J (the Jacobian of the system) and H (the Hessian of the system), for a
% given y (values of the integration variables).

%% Set thermodynamics constants, initial concentrations and temperature and
    evaluate enthalpies
p = 1e5; %Pa
Rgas = 8.3144598; % Js / K mol
nspecies = size(specie,2)-1;
nT = nspecies+1;
T = y(end);
y = y(1:end-1);

h = zeros(nspecies,1);
for i=1:nspecies
    h(i) = enth(specie(i), T);
end

meanwght = 0;
for i=1:nspecies
    meanwght = meanwght + y(i)/specie(i).w;
end

y = [y; specie(end).w*meanwght];

rho = p/(Rgas*T*meanwght)*1e-6; % g/cm3

%% Evaluate reaction rates and their derivatives
nreactions = size(reaction,2);
omega = zeros(nreactions,1);
omegad = zeros(nreactions,3);
omegadd = zeros(nreactions,3,3);
for i = 1:nreactions
    cprod(1:3) = 1;
    coeff(1:3) = 1;
    for ir = 1:reaction(i).nreact
        isa = reaction(i).reactants(ir);
        cprod(ir) = y(isa) * rho/specie(isa).w; %(mol/cm3)^2 or (mol/cm3)^3
        coeff(ir) = rho/specie(isa).w;
    end
    c = reaction(i).A * T^reaction(i).n * exp(-reaction(i).Ta/T);
    cd = ( reaction(i).n + reaction(i).Ta/T ) / T;
```

```
43      omega(i) = c * cprod(1) * cprod(2) * cprod(3); %(mol/cm3)
44      omegad(i,1) = c * coeff(1) * cprod(2) * cprod(3);
45      omegad(i,2) = c * cprod(1) * coeff(2) * cprod(3);
46      omegad(i,3) = omega(i) * ( cd - reaction(i).nreact / T );
47      omegadd(i,1,1) = 0;
48      omegadd(i,1,2) = c * coeff(1) * coeff(2) * cprod(3);
49      omegadd(i,1,3) = omegad(i,1) * ( cd - reaction(i).nreact / T );
50      omegadd(i,2,1) = omegadd(i,1,2);
51      omegadd(i,2,2) = 0;
52      omegadd(i,2,3) = omegad(i,2) * ( cd - reaction(i).nreact / T );
53      omegadd(i,3,1) = omegadd(i,1,3);
54      omegadd(i,3,2) = omegadd(i,2,3);
55      omegadd(i,3,3) = omegad(i,3) * ( cd - reaction(i).nreact / T ) + omega(
            i) * ( reaction(i).nreact / T^2 );
56  end
57
58  %% Sum reaction rates and their derivatives
59  Fs = zeros(nT,1);
60  Js = zeros(nT,nT);
61  Hs = zeros(nT,nT,nT);
62  for i = 1:nreactions
63      for ir=1:reaction(i).nreact
64          isa = reaction(i).reactants(ir);
65          if isa ~= 11
66              Fs(isa) = Fs(isa) - omega(i);
67              isb = reaction(i).reactants(1);
68              if isb ~= 11
69                  Js(isa,isb) = Js(isa,isb) - omegad(i,1);
70                  isc = reaction(i).reactants(1);
71                  if isc ~= 11
72                      Hs(isa,isb,isc) = Hs(isa,isb,isc) - omegadd(i,1,1);
73                  end
74                  isc = reaction(i).reactants(2);
75                  if isc ~= 11
76                      Hs(isa,isb,isc) = Hs(isa,isb,isc) - omegadd(i,1,2);
77                  end
78                  Hs(isa,isb,end) = Hs(isa,isb,end) - omegadd(i,1,3);
79              end
80              isb = reaction(i).reactants(2);
81              if isb ~= 11
82                  Js(isa,isb) = Js(isa,isb) - omegad(i,2);
83                  isc = reaction(i).reactants(1);
84                  if isc ~= 11
85                      Hs(isa,isb,isc) = Hs(isa,isb,isc) - omegadd(i,2,1);
86                  end
87                  isc = reaction(i).reactants(2);
```

```
88              if isc ~= 11
89                  Hs(isa,isb,isc) = Hs(isa,isb,isc) - omegadd(i,2,2);
90              end
91              Hs(isa,isb,end) = Hs(isa,isb,end) - omegadd(i,2,3);
92          end
93          Js(isa,end) = Js(isa,end) - omegad(i,3); %dFsi/dT
94          isc = reaction(i).reactants(1);
95              if isc ~= 11
96                  Hs(isa,end,isc) = Hs(isa,end,isc) - omegadd(i,3,1);
97              end
98              isc = reaction(i).reactants(2);
99              if isc ~= 11
100                 Hs(isa,end,isc) = Hs(isa,end,isc) - omegadd(i,3,2);
101             end
102             Hs(isa,end,end) = Hs(isa,end,end) - omegadd(i,3,3);
103     end
104 end
105 for ir=1:reaction(i).nprod
106     isa = reaction(i).products(ir);
107     if isa ~= 11
108         Fs(isa) = Fs(isa) + omega(i);
109         isb = reaction(i).reactants(1);
110         if isb ~= 11
111             Js(isa,isb) = Js(isa,isb) + omegad(i,1);
112             isc = reaction(i).reactants(1);
113             if isc ~= 11
114                 Hs(isa,isb,isc) = Hs(isa,isb,isc) + omegadd(i,1,1);
115             end
116             isc = reaction(i).reactants(2);
117             if isc ~= 11
118                 Hs(isa,isb,isc) = Hs(isa,isb,isc) + omegadd(i,1,2);
119             end
120             Hs(isa,isb,end) = Hs(isa,isb,end) + omegadd(i,1,3);
121         end
122         isb = reaction(i).reactants(2);
123         if isb ~= 11
124             Js(isa,isb) = Js(isa,isb) + omegad(i,2);
125             isc = reaction(i).reactants(1);
126             if isc ~= 11
127                 Hs(isa,isb,isc) = Hs(isa,isb,isc) + omegadd(i,2,1);
128             end
129             isc = reaction(i).reactants(2);
130             if isc ~= 11
131                 Hs(isa,isb,isc) = Hs(isa,isb,isc) + omegadd(i,2,2);
132             end
133             Hs(isa,isb,end) = Hs(isa,isb,end) + omegadd(i,2,3);
```

```
            end
            Js(isa,end) = Js(isa,end) + omegad(i,3); %dFsi/dT
            isc = reaction(i).reactants(1);
                if isc ~= 11
                    Hs(isa,end,isc) = Hs(isa,end,isc) + omegadd(i,3,1);
                end
                isc = reaction(i).reactants(2);
                if isc ~= 11
                    Hs(isa,end,isc) = Hs(isa,end,isc) + omegadd(i,3,2);
                end
                Hs(isa,end,end) = Hs(isa,end,end) + omegadd(i,3,3);
        end
    end
end

%% Evaluate rate equations, temperature equation and their derivatives
rhod = Rgas*meanwght/p;
F = zeros(nT,1);
J = zeros(nT,nT);
H = zeros(nT,nT,nT);
meancp = 0;
for i=1:nspecies
    meancp = meancp + y(i) * specie(i).cp;
    % Fi, FT
    F(i) = Fs(i) * specie(i).w / rho;
    F(end) = F(end) - F(i)*h(i);
    % Jij
    for j = 1:nspecies
        J(i,j) = Js(i,j) * specie(i).w / rho;
        % Hijk
        for k = 1:nspecies
            H(i,j,k) = specie(i).w / rho * Hs(i,j,k);
        end
        % HijT, HiTj
        H(i,j,end) = specie(i).w / rho * Hs(i,j,end) + specie(i).w * rhod *
             Js(i,j);
        H(i,end,j) = H(i,j,end);
    end
    % JiT, JTT, HiTT
    J(i,end) = Js(i,end) * specie(i).w / rho + specie(i).w * rhod * Fs(i);
    J(end,end) = J(end,end) - ( F(i) * specie(i).cp + J(i,end) * h(i) );
    H(i,end,end) = specie(i).w / rho * Hs(i,end,end) + 2 * specie(i).w *
        rhod * Js(i,end);
end
% FT, JTT
F(end) = F(end) / meancp;
```

```
J(end,end) = J(end,end) / meancp;

for j = 1:nspecies
    % JTj, HTjT, HTTj
    for i = 1:nspecies
        J(end,j) = J(end,j) - J(i,j) * h(i);
        H(end,j,end) = H(end,j,end) - H(i,j,end) * h(i) - J(i,j) * specie(i
            ).cp;
    end
    J(end,j) = J(end,j)/meancp;
    H(end,j,end) = H(end,j,end) / meancp;
    H(end,end,j) = H(end,j,end);
    % HTjk
    for k = 1:nspecies
        for i = 1:nspecies
            H(end,j,k) = H(end,j,k) - H(i,j,k) * h(i);
        end
        H(end,j,k) = H(end,j,k) / meancp;
    end
end

for i = 1:nspecies
    H(end,end,end) = H(end,end,end) - H(i,end,end) * h(i) - 2 * J(i,end) *
        specie(i).cp;
end
H(end,end,end) = H(end,end,end) / meancp;

end

function enth = enth(specie, T) % J/kg
% evaluate enthalpy of a specie as a function of temperature
    enth = specie.hf + (specie.cp)*(T-298);
end
```

Listing A.4: Function evaluating the ODEs, as well as the Jacobian and Hessian, in sparse format.

```
function [F,J,H] = F_cheb(y, reaction, specie)
% Evaluates f (the system of equations to be integrated),
% J (the Jacobian of the system) and H (the Hessian of the system), for a
% given y (values of the integration variables). J and H are stored in
% sparse format

%% Set thermodynamics constants, initial concentrations and temperature and
     evaluate enthalpies
p = 1e5; %Pa
```

```
Rgas = 8.3144598; % J / K mol
nspecies = size(specie,2)-1;
nT = nspecies+1;
T = y(end);
y = y(1:end-1);

h = zeros(nspecies,1);
for i=1:nspecies
    h(i) = enth(specie(i), T);
end

meanwght = 0;
for i=1:nspecies
    meanwght = meanwght + y(i)/specie(i).w;
end

y = [y; specie(end).w*meanwght];

rho = p/(Rgas*T*meanwght)*1e-6; % g/cm3

%% Evaluate reaction rates and their derivatives
nreactions = size(reaction,2);
omega = zeros(nreactions,1);
omegad = zeros(nreactions,3);
omegadd = zeros(nreactions,3,3);
for i = 1:nreactions
    cprod(1:3) = 1;
    coeff(1:3) = 1;
    for ir = 1:reaction(i).nreact
        isa = reaction(i).reactants(ir);
        cprod(ir) = y(isa) * rho/specie(isa).w; %(mol/cm3)^2 or (mol/cm3)^3
        coeff(ir) = rho/specie(isa).w;
    end
    c = reaction(i).A * T^reaction(i).n * exp(-reaction(i).Ta/T);
    cd = ( reaction(i).n + reaction(i).Ta/T ) / T;
    omega(i) = c * cprod(1) * cprod(2) * cprod(3); %(mol/cm3)
    omegad(i,1) = c * coeff(1) * cprod(2) * cprod(3);
    omegad(i,2) = c * cprod(1) * coeff(2) * cprod(3);
    omegad(i,3) = omega(i) * ( cd - reaction(i).nreact / T );
    omegadd(i,1,1) = 0;
    omegadd(i,1,2) = c * coeff(1) * coeff(2) * cprod(3);
    omegadd(i,1,3) = omegad(i,1) * ( cd - reaction(i).nreact / T );
    omegadd(i,2,1) = omegadd(i,1,2);
    omegadd(i,2,2) = 0;
    omegadd(i,2,3) = omegad(i,2) * ( cd - reaction(i).nreact / T );
    omegadd(i,3,1) = omegadd(i,1,3);
```

```
    omegadd(i,3,2) = omegadd(i,2,3);
    omegadd(i,3,3) = omegad(i,3) * ( cd - reaction(i).nreact / T ) + ...
        + omega(i) * ( reaction(i).nreact / T^2 );
end

%% Sum reaction rates and their derivatives
F = zeros(nT,1);
Jspos = zeros(80,2);
Jsval = zeros(80,1);
Hspos = zeros(400,3);
Hsval = zeros(400,1);
posJ = 1;
posH = 1;

for i = 1:nreactions
    for ir=1:reaction(i).nreact
        isa = reaction(i).reactants(ir);
        if isa ~= 11
            F(isa) = F(isa) - omega(i);
            isb = reaction(i).reactants(1);
            if isb ~= 11
                Jspos(posJ,:) = [isa isb];
                Jsval(posJ) = - omegad(i,1);
                posJ = posJ+1;
                isc = reaction(i).reactants(1);
                if isc ~= 11
                    Hspos(posH,:) = [isa isb isc];
                    Hsval(posH) = - omegadd(i,1,1);
                    posH = posH+1;
                end
                isc = reaction(i).reactants(2);
                if isc ~= 11
                    Hspos(posH,:) = [isa isb isc];
                    Hsval(posH) = - omegadd(i,1,2);
                    posH = posH+1;
                end
                Hspos(posH,:) = [isa isb nT];
                Hsval(posH) = - omegadd(i,1,3);
                posH = posH+1;
            end
            isb = reaction(i).reactants(2);
            if isb ~= 11
                Jspos(posJ,:) = [isa isb];
                Jsval(posJ) = - omegad(i,2);
                posJ = posJ+1;
                isc = reaction(i).reactants(1);
```

```
101                 if isc ~= 11
102                     Hspos(posH,:) = [isa isb isc];
103                     Hsval(posH) = - omegadd(i,2,1);
104                     posH = posH+1;
105                 end
106                 isc = reaction(i).reactants(2);
107                 if isc ~= 11
108                     Hspos(posH,:) = [isa isb isc];
109                     Hsval(posH) = - omegadd(i,2,2);
110                     posH = posH+1;
111                 end
112                 Hspos(posH,:) = [isa isb nT];
113                 Hsval(posH) = - omegadd(i,2,3);
114                 posH = posH+1;
115             end
116             Jspos(posJ,:) = [isa nT];
117             Jsval(posJ) = - omegad(i,3);
118             posJ = posJ+1; %dFi/dT
119             isc = reaction(i).reactants(1);
120                 if isc ~= 11
121                     Hspos(posH,:) = [isa nT isc];
122                     Hsval(posH) = - omegadd(i,3,1);
123                     posH = posH+1;
124                 end
125                 isc = reaction(i).reactants(2);
126                 if isc ~= 11
127                     Hspos(posH,:) = [isa nT isc];
128                     Hsval(posH) = - omegadd(i,3,2);
129                     posH = posH+1;
130                 end
131                 Hspos(posH,:) = [isa nT nT];
132                 Hsval(posH) = - omegadd(i,3,3);
133                 posH = posH+1;
134         end
135     end
136     for ir=1:reaction(i).nprod
137         isa = reaction(i).products(ir);
138         if isa ~= 11
139             F(isa) = F(isa) + omega(i);
140             isb = reaction(i).reactants(1);
141             if isb ~= 11
142                 Jspos(posJ,:) = [isa isb];
143                 Jsval(posJ) = omegad(i,1);
144                 posJ = posJ+1;
145                 isc = reaction(i).reactants(1);
146                 if isc ~= 11
```

```
147                     Hspos(posH,:) = [isa isb isc];
148                     Hsval(posH) = + omegadd(i,1,1);
149                     posH = posH+1;
150                 end
151                 isc = reaction(i).reactants(2);
152                 if isc ~= 11
153                     Hspos(posH,:) = [isa isb isc];
154                     Hsval(posH) = + omegadd(i,1,2);
155                     posH = posH+1;
156                 end
157                 Hspos(posH,:) = [isa isb nT];
158                 Hsval(posH) = + omegadd(i,1,3);
159                 posH = posH+1;
160             end
161             isb = reaction(i).reactants(2);
162             if isb ~= 11
163                 Jspos(posJ,:) = [isa isb];
164                 Jsval(posJ) = omegad(i,2);
165                 posJ = posJ+1;
166                 isc = reaction(i).reactants(1);
167                 if isc ~= 11
168                     Hspos(posH,:) = [isa isb isc];
169                     Hsval(posH) = + omegadd(i,2,1);
170                     posH = posH+1;
171                 end
172                 isc = reaction(i).reactants(2);
173                 if isc ~= 11
174                     Hspos(posH,:) = [isa isb isc];
175                     Hsval(posH) = + omegadd(i,2,2);
176                     posH = posH+1;
177                 end
178                 Hspos(posH,:) = [isa isb nT];
179                 Hsval(posH) = + omegadd(i,2,3);
180                 posH = posH+1;
181             end
182             Jspos(posJ,:) = [isa nT];
183             Jsval(posJ) = omegad(i,3);
184             posJ = posJ+1;
185             isc = reaction(i).reactants(1);
186                 if isc ~= 11
187                     Hspos(posH,:) = [isa nT isc];
188                     Hsval(posH) = + omegadd(i,3,1);
189                     posH = posH+1;
190                 end
191                 isc = reaction(i).reactants(2);
192                 if isc ~= 11
```

```
                    Hspos(posH,:) = [isa nT isc];
                    Hsval(posH) = + omegadd(i,3,2);
                    posH = posH+1;
                end
                Hspos(posH,:) = [isa nT nT];
                Hsval(posH) = + omegadd(i,3,3);
                posH = posH+1;
        end
    end
end

Jspos(~any(Jsval,2),:) = [];
Jsval(~any(Jsval,2)) = []; %cuts off zeros
Hspos(~any(Hsval,2),:) = [];
Hsval(~any(Hsval,2)) = [];

[Jspos, Jsval] = compress(Jspos, Jsval);
[Hspos, Hsval] = compress(Hspos, Hsval);

%% Evaluate rate equations, temperature equation and their derivatives
posH = 1;
posJ = 1;
Jpos = zeros(120,2);
Jval = zeros(120,1);
Hpos = zeros(700,3);
Hval = zeros(700,1);

rhod = Rgas*meanwght/p;
meancp = 0; summ = 0;
for i=1:nspecies
    meancp = meancp + y(i) * specie(i).cp;
end

for i=1:nspecies
    %JiT
    Jpos(posJ,:) = [i nT];
    Jval(posJ) = F(i) * specie(i).w * rhod;
    posJ = posJ + 1;
    % Fi
    F(i) = F(i) * specie(i).w/rho;
    % FT
    summ = summ + F(i) * h(i);
    % JTT
    Jpos(posJ,:) = [nT nT];
    Jval(posJ) = F(i) * (-specie(i).cp/meancp);
    posJ = posJ + 1;
```

```
239 end
240 F(end) = - summ/meancp;
241
242 for n = 1:length(Jsval)
243     i = Jspos(n,1);
244     j = Jspos(n,2);
245     % J_ij, J_iT
246     Jpos(posJ,:) = [i j];
247     Jval(posJ) = Jsval(n) * specie(i).w / rho;
248     % J_Tj, J_TT
249     Jpos(posJ+1,:) = [nT j];
250     Jval(posJ+1) = Jval(posJ) * (-h(i)/meancp);
251     posJ = posJ + 2;
252     % H_ijT, H_iTj
253     Hpos(posH,:) = [i j nT];
254     Hval(posH) = Jsval(n) * specie(i).w * rhod;
255     Hpos(posH+1,:) = [i nT j];
256     Hval(posH+1) = Hval(posH);
257     posH = posH + 2;
258     % H_TjT, H_TTj, H_TTT
259     Hpos(posH,:) = [nT j nT];
260     Hval(posH) = Jval(posJ-2) * (-specie(i).cp) / meancp;
261     if j == nT
262         Hval(posH) = Hval(posH) * 2;
263     end
264     posH = posH + 1;
265 end
266
267 for n = 1:length(Hsval)
268     i = Hspos(n,1);
269     j = Hspos(n,2);
270     k = Hspos(n,3);
271     % H_ijk, H_ijT, H_iTj
272     Hpos(posH,:) = [i j k];
273     Hval(posH) = Hsval(n) * specie(i).w / rho;
274     % H_Tjk
275     Hpos(posH+1,:) = [nT j k];
276     Hval(posH+1) = Hval(posH) * (-h(i) / meancp);
277     posH = posH + 2;
278 end
279
280 Jpos(~any(Jval,2),:) = [];
281 Jval(~any(Jval,2)) = []; %cuts off zeros
282 Hpos(~any(Hval,2),:) = [];
283 Hval(~any(Hval,2)) = []; %cuts off zeros
284
```

```
285 J = sparse(Jpos(:,1), Jpos(:,2), Jval, nT, nT);
286 H = sptensor(Hpos, Hval, [nT nT nT]);
287 end
288
289 function enth = enth(specie, T) % J/kg
290 % evaluate enthalpy of a specie as a function of temperature
291     enth = specie.hf + (specie.cp)*(T-298);
292 end
293
294 function [X, Y] = compress(pos, val)
295 % compress sparse triplet by summing terms with equal index
296         [X,~,loc] = unique(pos,'rows');
297         Y = accumarray(loc, val,[size(X,1) 1]);
298 end
```

Bibliography

[1] Brett W. Bader and Tamara G. Kolda. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, December 2007.

[2] Brett W. Bader, Tamara G. Kolda, et al. Matlab tensor toolbox version 3.0-dev. Available online, 2017.

[3] Alexander Burcat, Branko Ruscic, et al. Third millenium ideal gas and condensed phase thermochemical database for combustion (with update from active thermochemical tables). Technical report, Argonne National Laboratory (ANL), 2005.

[4] Compiled by A. D. McNaught and A. Wilkinson. *IUPAC. Compendium of Chemical Terminology, 2nd ed. (the "Gold Book").* Blackwell Scientific Publications, Oxford, 1997.

[5] Marcus Ó Conaire, Henry J. Curran, John M. Simmie, William J. Pitz, and Charles K. Westbrook. A comprehensive modeling study of hydrogen oxidation. *International Journal of Chemical Kinetics*, 36(11):603–622.

[6] François Dubeau and Calvin Gnang. Fixed point and newton's methods for solving a nonlinear equation: From linear to high-order convergence. *SIAM Review*, 56(4):691–708, 2014.

[7] Lawrence F. Shampine and Mark Reichelt. The matlab ode suite. 18, 05 1997.

[8] William Cecil Gardiner and Alexander Burcat. *Combustion chemistry.* Springer, 1984.

[9] John R Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in matlab: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[10] Robert J Kee, Fran M Rupley, Ellen Meeks, and James A Miller. Chemkin-iii: A fortran chemical kinetics package for the analysis of gas-phase chemical and plasma kinetics. Technical report, Sandia National Labs., Livermore, CA (United States), 1996.

[11] CT Kelley. Iterative methods for linear and nonlinear equations, siam, philadelphia, 1995. *MR 96d*, 65002.

[12] H.J. Lee and W.E. Schiesser. *Ordinary and Partial Differential Equation Routines in C, C++, Fortran, Java, Maple, and MATLAB.* CRC Press, 2003.

[13] Elaine S. Oran and Jay P. Boris. *Numerical Simulation of Reactive Flows.* Cambridge Univeristy Press, 2nd edition, 2001.

[14] Federico Perini, Emanuele Galligani, and Rolf D Reitz. An analytical jacobian approach to sparse reaction kinetics for computationally efficient combustion modeling with large reaction mechanisms. *Energy & Fuels*, 26(8):4804–4822, 2012.

[15] Tamás Turányi and Alison S Tomlin. *Analysis of kinetic reaction mechanisms*. Springer, 2016.

[16] MR Whitbeck. Numerical modeling of chemical reaction mechanisms. *Tetrahedron Computer Methodology*, 3(6):497–505, 1990.

[17] Hiroshi Yamaguchi. *Engineering fluid mechanics*, volume 85. Springer Science & Business Media, 2008.