Fast Combustion Simulations

Time-Integration of Chemical Processes in Reactor Networks, using Julia

BSc thesis in Applied Mathematics and Physics

J.S. van der Heide

Delft University of Technology

TU Delft

Fast Combustion Simulations

Time-Integration of Chemical Processes in Reactor Networks, using Julia

by

J.S. van der Heide

to obtain the degrees of
Bachelor of Science in Applied Mathematics
and
Bachelor of Science in Applied Physics
at the Delft University of Technology,

to be defended publicly on the 8th of September 2025.

AI USE STATEMENT
I have used several chatbots to develop the code and to find relevant literature only.

Student number:	4607910	
Project duration:	January, 2024 – May, 2025	
Thesis committee:	Dr. D. J. P. Lahaye,	TU Delft, supervisor
	Prof. dr. ir. C. R. Kleijn,	TU Delft, supervisor
	Dr. A. Palha da Silva Clérigo	TU Delft
	Dr. Bijoy Bera	TU Delft

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

TU Delft

Preface

Beter laat dan nooit...

Like many instances in my academic career this project took longer than planned. If physics is the study of reality and mathematics is the study of truth, then it is ironic that me losing grip on both is what caused these delays. There is a lesson I have learned: thoughts, words, symbols and formulas are never reality—they simulate it. We must never take the model to be more than a model, and if we take the answers of our thinking apparatus as the sure and whole truth then we are certainly delusional.

Many thanks for the patience everyone has had.

J.S. van der Heide
Delft, August 2025

Summary

Is Julia suited for chemical process analyses? Yes.

This thesis investigates solving ordinary differential equations (ODEs) in programming language Julia, which it does firstly through a simple example of algae populations, and secondly through complex chemical simulations. A progressively more complex framework describing chemical combustion and diffusion is established and implemented. Problems of great stiffness and containing 200+ variables have been integrated in less than 20 seconds, showing the promise this new, open-source and accessible code language.

The thermochemical model that was developed could be improved upon: it has the problem of unexpectedly low flame temperatures because of heat lost to dissociative effects, the current solution for preventing variables overshooting into the negative, it is is ugly and there is ever more optimisation possible. What the program does is run, and it approximately simulates the kinetics and thermochemistry of combustion within acceptable runtimes, thus proving the technical feasibility but lacking the complete implementation of graph-based chemical analysis in Julia.

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
0D	0-Dimensional
CRN	Chemical Reactor Network
RM	Reaction Mechanism
ODE	Ordinary Differential Equation
ESDIRK	Explicit first stage Singly DIagonally Implicit Runge-Kutta method
BDF	Backward Differentiation Formula

Symbols

The symbols of chapter 2 have been omitted, for this chapter is purely introductory and contains no relevant physical quantities used in later chapters.

Symbol	Definition	Unit
A	Pre-exponential factor in Arrhenius equation	$[(\mathrm{mol/m^3})^{1-\mathrm{n_j}}/\mathrm{s}]$
$A_{\mathbb{D}}$	Pre-factor in the equation for diffusivity	$[\mathrm{atm\AA^2 cm^2/(K^{3/2}s\sqrt{g/mol})}]$
$\mathcal{A}_i$	Chemical specie i	[-]
c_i	Concentration of species i	$[\mathrm{mol/m^3}]$
$\vec{c}$	Vector of all concentrations	$[(\mathrm{mol/m^3})^{\mathrm{n_s}}]$
$c_{V,v}$	Total volumetric isochoric heat capacity	$[\mathrm{J/K/m^3}]$
$c_{P,v}$	Total volumetric isobaric heat capacity	$[\mathrm{J/K/m^3}]$
$c^o_{P,M}$	Standard state molar isobaric heat capacity	$[\mathrm{J/K/mol}]$
$c_{P,M,i}(T)$	Molar isobaric heat capacity of species i	$[\mathrm{J/K/mol}]$
$\mathbb{D}^i_k$	Diffusivity of specie i in node k	$[\mathrm{m^2/s}]$
d	Effective molecular diameter	[Å] or $[\mathrm{m}]$
E_a	Activation energy of a reaction	$[\mathrm{cal/mol}]$
ΔF	Helmholtz free energy change of a reaction	$[\mathrm{J/mol}]$
f_i	Efficiency of specie i as a third body reactions	[-]
$f_{kk'}$	Weight of edge between k and k'	$[\mathrm{m}]$
ΔG	Gibbs free energy change of a reaction	$[\mathrm{J/mol}]$
$G = (V, E)$	Graph G with nodes V and edges E	[-]
H_v	Enthalpy per volume	$[\mathrm{J/m^3}]$
$\Delta H_{M,j}(T)$	Molar enthalpy change of reaction j	$[\mathrm{J/mol}]$
$H_{M,i}(T)$	Absolute molar enthalpy for species i	$[\mathrm{J/mol}]$
i	Index number for a chemical specie	[-]
j	Index number for a reaction	[-]
$J_{kk'}$	Diffusive transport into k from k'	$[\mathrm{mol/s}]$
k_B	Boltzmann's constant	$[\mathrm{J/K}]$
$k(T)$	Rate constant of a reaction	$[(\mathrm{mol/m^3})^{1-\mathrm{n_j}}/\mathrm{s}]$
k_o	Low pressure rate constant in falloff reactions	$[(\mathrm{mol/m^3})^{1-\mathrm{n_j}}/\mathrm{s}]$
k_∞	High pressure rate constant in falloff reactions	$[(\mathrm{mol/m^3})^{1-\mathrm{n_j}}/\mathrm{s}]$
$K_{eq} = k_{\mathrm{forward}}/k_{\mathrm{reverse}}$	Equilibrium constant	$[-]$
k	index number for node k	[-]
L	Graph Laplacian matrix	[-]
M	Molar mass	$[\mathrm{g/mol}]$

Symbol	Definition	Unit
m	Mass of a molecule	[kg]
N_A	Avogadro's constant	[$1/\mathrm{mol}$]
n_s	Number of chemical species	[-]
n_r	Number of reactions	[-]
n_{ij}	Order of reaction j for species i	[-]
$n_j = \sum_i n_{ij}$	Total order of reaction j	[-]
n_n	Number of nodes	[-]
P	Pressure	[Pa]
Q_k	Total heat in node k	[J]
$q_{kk'}$	Heat flux into k from k'	[W]
$Q_{kk'}$	Volumetric flow rate into k from k'	[m^3/s]
r_j	Rate of reaction j	[$\mathrm{mol}/\mathrm{m}^3/\mathrm{s}$]
$\vec{r}$	Vector of all reaction rates	[$(\mathrm{mol}/\mathrm{m}^3\mathrm{s})^{n_r}$]
R	Universal gas constant	[$\mathrm{J}/\mathrm{mol}/\mathrm{K}$] or [$\mathrm{cal}/\mathrm{mol}/\mathrm{K}$]
S	Stoichiometric $n_s \times n_r$ matrix	[-]
$S_M(T)$	Molar entropy	[$\mathrm{J}/\mathrm{mol}/\mathrm{K}$]
ΔS_M	Entropy change of a reaction	[$\mathrm{J}/\mathrm{mol}/\mathrm{K}$]
T	Temperature	[K]
$T^* = T/(\epsilon/k_B)$	Reduced temperature	[-]
t	Time	[s]
ΔU	Internal energy change of a reaction	[J/mol]
U	Network state matrix	[$(\mathrm{K}, \mathrm{mol}/\mathrm{m}^3, \ldots, \mathrm{mol}/\mathrm{m}^3)^{n_n}$]
V_k	Volume of node k	[m^3]
$\vec{X} = (T, c_1, \ldots, c_{n_s})$	State vector of a reactor	[$(\mathrm{K}, \mathrm{mol}/\mathrm{m}^3, \ldots, \mathrm{mol}/\mathrm{m}^3)$]
$[\mathrm{XYZ}]$	Concentration of molecule XYZ	[$\mathrm{mol}/\mathrm{m}^3$]
ϵ	Well-depth of Lennard-Jones potential	[J]
κ_k	Conductivity in node k	[W/mK]
Λ_k	Container coefficient of node k	[m^3] or [J/K]
ν_{ij}	Coefficient of specie i in reaction j	[-]
Φ	Preserved quantity flux through an edge	[mol/s] or [W]
ϕ_k	Intrinsic variable for a preserved quantity	[$\mathrm{mol}/\mathrm{m}^3$] or [$\mathrm{K}$]
Θ	Extrinsic variable for a preserved quantity	[mol] or [J]
$\Omega(T^*)$	Collision integral	[-]

1 Introduction

Making fire has been a technology since before *homo* was *sapiens* [40]. With the industrial revolution the harnessing of the energy from flames has developed into an integral element of developed society: the use-cases for heat have proven near limitless [8]. The consequences of burning hydrocarbon fossils are catastrophic [36]. Combustion may occur when energy carriers (e.g. CH_4, H_2), oxygen (O_2) and heat are brought together. This oxidation process always has products—some of which are harmless (e.g. H_2O), some of which pollute the environment and cause ecosystem degradation (e.g. CO_2) and some of which can instantly kill a man (e.g. CO).

When designing an industrial chemical process it is important to have some idea of which chemical species are present at each step of the process so a proper risk-assessment can be made. To get this idea a simulation of the combustion process is frequently employed. Computational prediction of chemical species in industrial processes is a well-established field. Traditional process simulation software, such as Aspen Plus and HYSYS, are widely used for modelling entire processes and predicting species distributions [49]. Combustion simulation however remains rather complicated and computation can be demanding. This is a problem because, while designing, a process engineer will usually iterate his design many times and if each iterative step is tied to a long wait before a computer can give predictions on chemical species present in the process, then this will hinder productivity. Apparent is the need to use a programming language that is both intuitive enough to keep complex scientific programming doable (ruling out C) and fast enough not to hinder design workflows (ruling out Python). Julia is an open-source programming language by Bezanson et al. [5], that is said to be as intuitive as Python and as fast as C, and because of this it has emerged as a promising candidate for running these simulations. *This thesis evaluates the possibility and feasibility of simulations of industrial chemical processes in Julia.*

To answer the question of the feasibility of fast combustion process simulation in Julia, code has been developed to study the viability of this approach. This thesis begins in chapter 2 that introduces the integration of unstable differential equations in Julia with the help of an example, namely the simulation of a population of living organisms. To do the analysis of combustion, first a 0-dimensional (0D) model (a perfectly stirred reactor vessel) of the chemistry and thermodynamics at play was developed. After achieving a functional reaction mechanism in chapter 3, secondly a set of these 0D models was combined into a graph over which diffusion was simulated in chapter 4. All this generated differential problems to be solved, resulting in predictions for the evolution of chemical species and heat present in the system. Lastly there is chapter 5 in which the results of the research are presented and discussed.

The graphs in this text are all the result of code written by the author, which can be found at `https://github.com/Jelterminator/Chemical-Combustion`.

2

Numerical integration of a stiff system that describes a population of algae

If simply posing the problem is difficult then it helps if we can easily read what we code, making either Python or Julia a valid choice. If the computations required for solving the problem are numerous then a more optimised language is preferred. Because the chemical analysis of combustion in a reactor network model is problematic in both these respects, we have a compelling case for using Julia. A new coding language is an obstacle and to get acquainted with the syntax the choice was made to first do a warm-up project. Albeit lacking the complexity of combustion simulation and being conceptually completely different, this project which simulates populations of algae is functionally similar in the challenges it poses—it is still a stiff system of ordinary differential equations (ODEs).

Suppose we have a tank of water and in it some species of algae. P is the algae population which is expressed in units of $[\mathrm{g/m^3}]$. What will be the ODE which describes the system? If the size of the population is very small, we may assume growth to be unlimited. The population will grow as cells multiply. If cells multiply every α time units, then every cell generates $1/\alpha$ new cells per time unit and so too the size of the whole population P will increase with $1/\alpha$ of its size per time unit.

$$\frac{dP}{dt} = P/\alpha \implies P(t) = P(0)e^{t/\alpha} \tag{2.1}$$

If we had assumed that growth would occur most rapid in summer and slowest in winter, then the problem would have been different because $\alpha(t)$ would then be time-dependent. Now, let $\beta(t) = 1/\alpha(t)$ so that our ordinary differential equation (ODE) becomes:

$$\frac{dP}{dt} = \beta(t)P \tag{2.2}$$

Which can be analytically solved and has a solution in the form of:

$$P(t) = P(0)e^{\int_0^t \beta(t')dt'} \tag{2.3}$$

This is a problem we can also solve numerically through algorithms that in their most general form look like algorithm 1. The results of this numeric integration using $\beta(t) = (1 + \sin(2\pi t/365))/100$ are shown in figure 2.1.

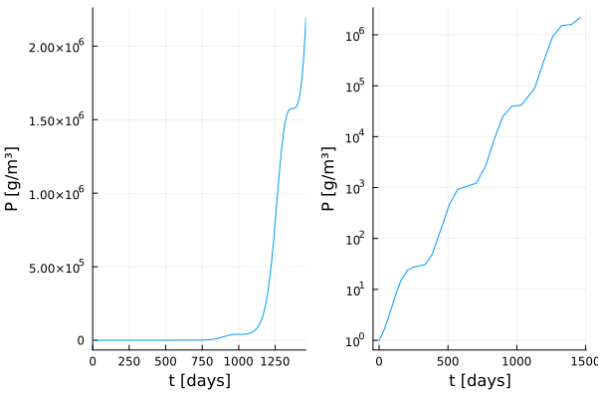

Figure 2.1: Results of the time integration over 4 years or 1460 days of the ODE $\frac{dP}{dt} = \beta(t)P$ with $\beta(t) = (1 + \sin(2\pi t/365))/100$ and $P(0) = 1$, done in Julia.

Algorithm 1 Pseudo-code for a numeric integration problem.

Define an ODE $\dot{X}(t) = F(X)$
Define a time interval $(t_{\text{initial}}, t_{\text{final}})$
Set initial values $X(t_{\text{initial}}) = X_{\text{initial}}$
Define acceptable error limit ε
while $t < t_{\text{final}}$ **do**
    Compute a trial step Δt
    Use numerical method (e.g., Runge-Kutta) to compute tentative solution X_{trial} at $t + \Delta t$
    Estimate local truncation error E
    if $E < \varepsilon$ **then**
        Accept step: $t \leftarrow t + \Delta t$, $X(t) \leftarrow X_{\text{trial}}$
    else
        Reject step and make Δt smaller
    end if
end while
return $X(t)$ over the interval

Unlimited growth is, however, unlikely in any physical case and thus a more realistic model needs to include a second term in its ODE, $-\phi(P)$ which simulates a population decrease. A simple but clever hypothesis is $\phi(P) = \gamma P^2$ [46], which leads to the following ODE:

$$\frac{dP}{dt} = P(\beta(t) - \gamma P) \tag{2.4}$$

For $\beta(t) = \beta$ this equation has a logistic curve as its solution:

$$P(t) = \frac{\beta}{\gamma + (\gamma/P(0) - \beta)e^{-\beta t}} \tag{2.5}$$

If β does depend on time, it is practical to do a numeric rather than analytic analysis, by applying the algorithm to our new ODE, which gives us the results of figure 2.2.

Finally, we can assume that the size of P is not limited by itself but by the presence of some other species C, say for example there is an algae-eating plankton in the tank of water. In this case the

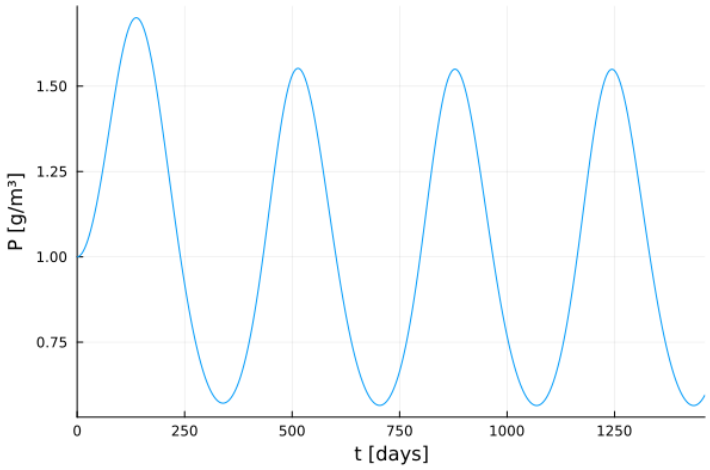

Figure 2.2: Results of the time integration over 4 years or 1460 days of the ODE $\frac{dP}{dt} = P(\beta(t) - \gamma P)$ with $\beta(t) = (1 + \sin(2\pi t/365))/100$, $\gamma = 0.01$ and $P(0) = 1$, done in Julia.

system is described by the Lotka-Volterra equations [27]:

$$\text{ODE} := \begin{cases} \dfrac{dP}{dt} = P(a - bC) \\ \dfrac{dC}{dt} = C(cP - d) \end{cases} \tag{2.6}$$

This ODE has a periodic solutions and admits a non-trivial steady state at $P^* = \frac{d}{c}, C^* = \frac{a}{b}$, as can be seen in figure 2.3. It is an interesting example to see what happens if we again add a time dependency, now of the form $a(t) = 1.1(1 + \sin(2t/365))$ the problem, because here we get dramatically different results for different integrator settings, illustrating how critically dependent on method of integration the solutions of differential systems can be. Stiff problems know a butterfly effect: small differences at one point can have gigantic results at a later point—the way a number is rounded can result in a significant difference down the line when a problem is sufficiently stiff and unstable. In figure 2.4 and figure 2.5, which started with identical initial conditions, we see the the development of the populations for periods of 1 year and 2 years, rendered linearly and logarithmically. In figure 2.4 the standard Julia solver method is employed, which uses a dynamically adapting mix of implicit and explicit methods where at each time-step the most suitable method is picked. In figure 2.5 the Kennedy-Carpenter 4th/7th order implicit-explicit mixed Runge-Kutta method was employed, additionally making the error tolerances very small to enforce precision.

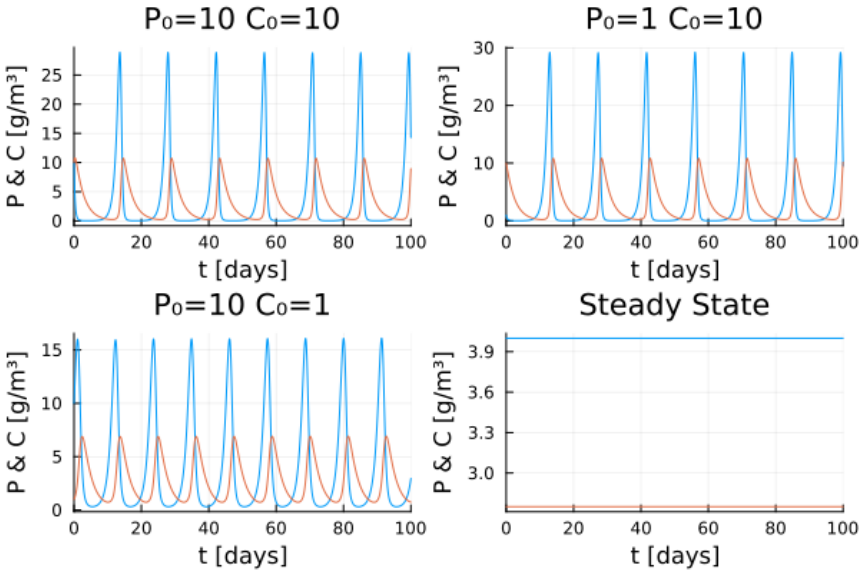

Figure 2.3: The solutions of numeric integration in Julia to the Lotka-Volterra equations (equation 2.6) with P in blue and C in orange, for 4 different initial conditions, the last of which is the steady state. The code results for $P_0 = 10$ and $C_0 = 10$ are identical to Wikipedia solutions [1], and the steady state for initial values indeed result in a steady state.

That the results are wildly different becomes apparent in the light of the weight of a single alga being about $1\mu\text{g}$: any density lower than this means extinction. To be exact, in a vat smaller than a cubic metre, if the population density drops below 10^{-6}g this means the complete extinction of that specie of algae. In the simulation with standard settings seen in figure 2.4, the extinction of C would have happened in the first winter, meanwhile in the high precision simulation shown in figure 2.5, this would have only happened in the second winter.

One must conclude that when we ask a computer to solve our problems for us we must remain arduously aware of what is actually happening and always remain cautious of possible faults the chosen methods may be committing. Another conclusion is that less strict error tolerances do not imply faster solutions, the runtime of the solve call with relaxed settings was 0.015s while the strict solve call ran for 0.0033s.

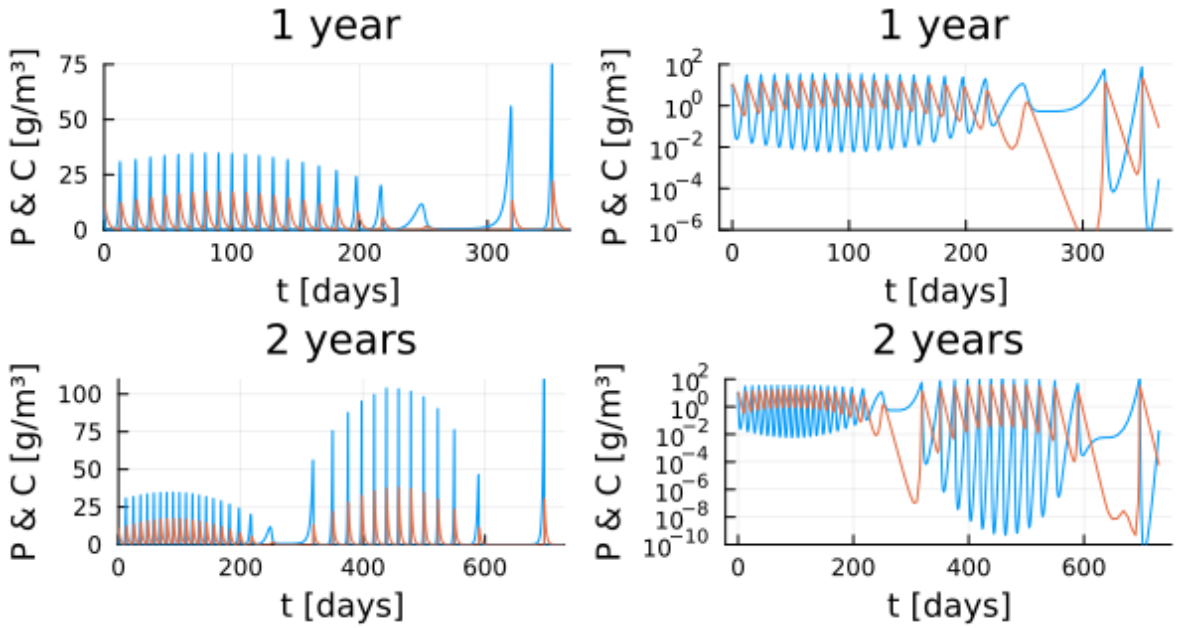

Figure 2.4: The results of integrating a time dependent Lotka-Volterra model in Julia with P in blue and C in orange, for $P_0 = 10$ and $C_0 = 10$, using the default mixed implicit-explicit method. The orange population drops out of frame in the first year and would have gone extinct.

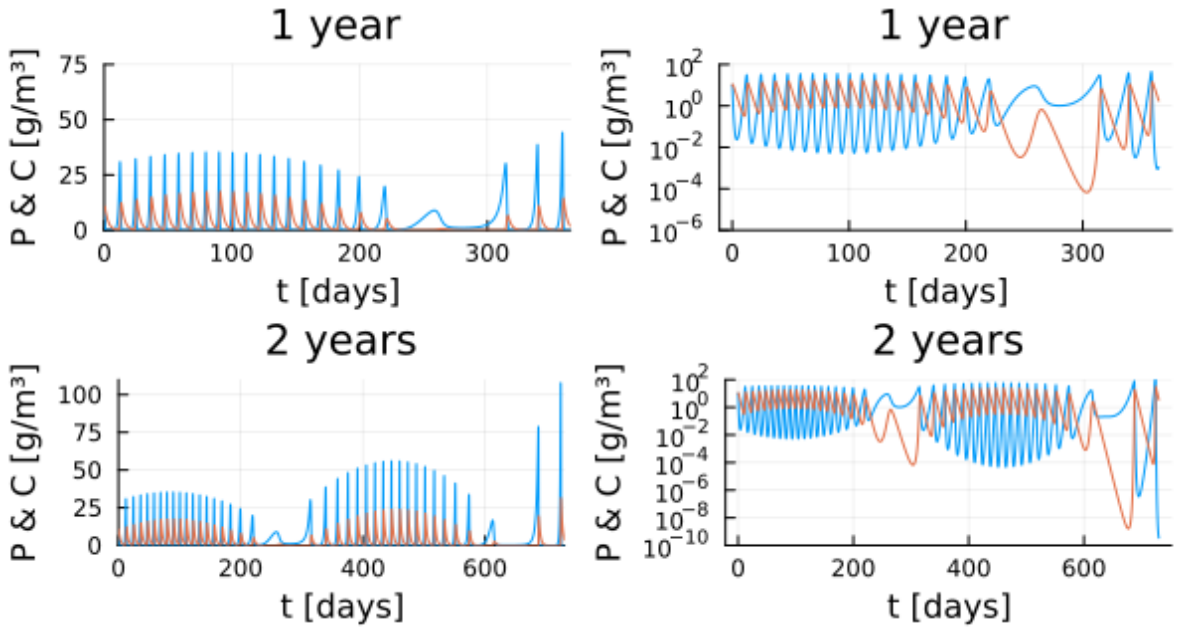

Figure 2.5: The results of integrating a time dependent Lotka-Volterra model in Julia with P in blue and C in orange, for $P_0 = 10$ and $C_0 = 10$, using a Kennedy-Carpenter 4th/7th order implicit-explicit mixed Runge-Kutta method and very strict error tolerances. The orange population survives the first year. Different results, while using the same initial values as in simulations of figure 2.4.

3

Zero-Dimensional Models, Kinetics & Thermodynamics

The general Chemical Reactor Network (CRN) model to be developed is complex because it must work for complex reaction mechanisms in any number of reactors with any number of flows between and beyond these reactors—rather complex problems indeed! For a start, however, the simplest case is one reactor with no in- or outflow. If one assumes a single perfectly stirred reactor they obtain a zero-dimensional (0D) problem, and this makes for the perfect playground to develop a model for chemical analysis. Atoms combine into arbitrarily complex molecules through near infinite amounts of reaction mechanism, yet computation is limited so there is a choice in the complexity of the Reaction Mechanism (RM).

3.1. The Arrhenius Equation

Suppose the model is a 0D chemical reactor with n_s species and n_r reactions, then the simulation is determined by an evolving state vector $\vec{X} = (T, \vec{c})$ consisting of a temperature T in Kelvin [K] and a vector of species concentrations $\vec{c} = (c_1, c_2, \ldots, c_i, \ldots, c_{n_s})$, with the c_i in moles per cubic metre [$\mathrm{mol/m^3}$]. This system changes as a set of reactions takes place, which can be described by a vector of reaction rates $\vec{r} = (r_1, r_2, \ldots, r_j, \ldots, r_{n_r})$, where the r_j are in moles per cubic metre per second [$\mathrm{mol/m^3/s}$], which, in the simplest case, are determined by:

$$r_j = k(T) \prod_{\text{species } i \text{ is a reactant of reaction } j} c_i^{n_{ij}} \tag{3.1}$$

in which n_{ij} is the order of reaction j for species i, and $k(T)$ is described by the Arrhenius equation [2]:

$$k(T) = Ae^{-\frac{E_a}{RT}} \tag{3.2}$$

where:

- A is a reaction-specific pre-exponential factor that may depend on T.
- E_a is the activation energy of the reaction.
- R is the universal gas constant.

If ν_{ij} is the coefficient in front of species i in the equation of reaction j, then a $n_s \times n_r$ stoichiometric matrix S is defined as:

$$S_{ij} = \begin{cases} -\nu_{ij} & \text{if } i \text{ is a reactant in } j \\ +\nu_{ij} & \text{if } i \text{ is a product in } j \end{cases} \tag{3.3}$$

then the following system of equations describes the evolution of the species concentrations:

$$\frac{d\vec{c}}{dt} = S\vec{r} \tag{3.4}$$

3.2. A Simple Reaction Mechanism

Perhaps it helps to clarify things with a simple example. If T is treated constant then the above is all there is to a RM. A very basic description would simply be: $2O_2$ + CH_4 $\implies$ $2H_2O$ + CO_2, with a reaction mechanism consisting of $n_s = 5$ species; N_2, O_2, CH_4, H_2O and CO_2, and $n_r = 1$ reaction. Nitrogen could have been left out, but it was not.

Then, the state equation $\dot{X} = f(X)$ is:

$$\frac{d}{dt}\begin{bmatrix}[\mathrm{N_2}]\\ [\mathrm{O_2}]\\ [\mathrm{CH_4}]\\ [\mathrm{H_2O}]\\ [\mathrm{CO_2}]\end{bmatrix} = \begin{bmatrix}0\\ -2\\ -1\\ 2\\ 1\end{bmatrix} k(T)[\mathrm{CH_4}]^{n_{\mathrm{CH_4}}}[\mathrm{O_2}]^{n_{\mathrm{O_2}}} \tag{3.5}$$

with $n_{\mathrm{O_2}} = 2$ and $n_{\mathrm{CH_4}} = 1$. This is readily translated to Julia (see appendix A), using the appropriate parameters for the Arrhenius equation [48].

The question of appropriate initial values comes up. Although not (yet) a variable, a choice for T has to be made. The minimum temperature to ignite methane would be $600\mathrm{K}$ [39], but the reaction rate is extremely low at this temperature. In reality the combustion generates heat which speeds up the process, and so a higher $T = 1200\mathrm{K}$, closer to the average over the whole process, was chosen. With high T comes high P, or else the reaction rates would be very low in this unphysically thin gas. For the species concentrations the choice was made for a stoichiometric mixture of air and CH_4 at 10bar, meaning a mixture containing exactly so much air that complete combustion can occur. The reaction demands $\mathrm{CH_4} : \mathrm{O_2} = 1 : 2$, now because air is 20.9% O_2, $\mathrm{CH_4} : \mathrm{air} = 1 : 2/0.209 = 1 : 9.57$. From the ideal gas law:

$$c = \frac{n}{V} = \frac{P}{RT} \tag{3.6}$$

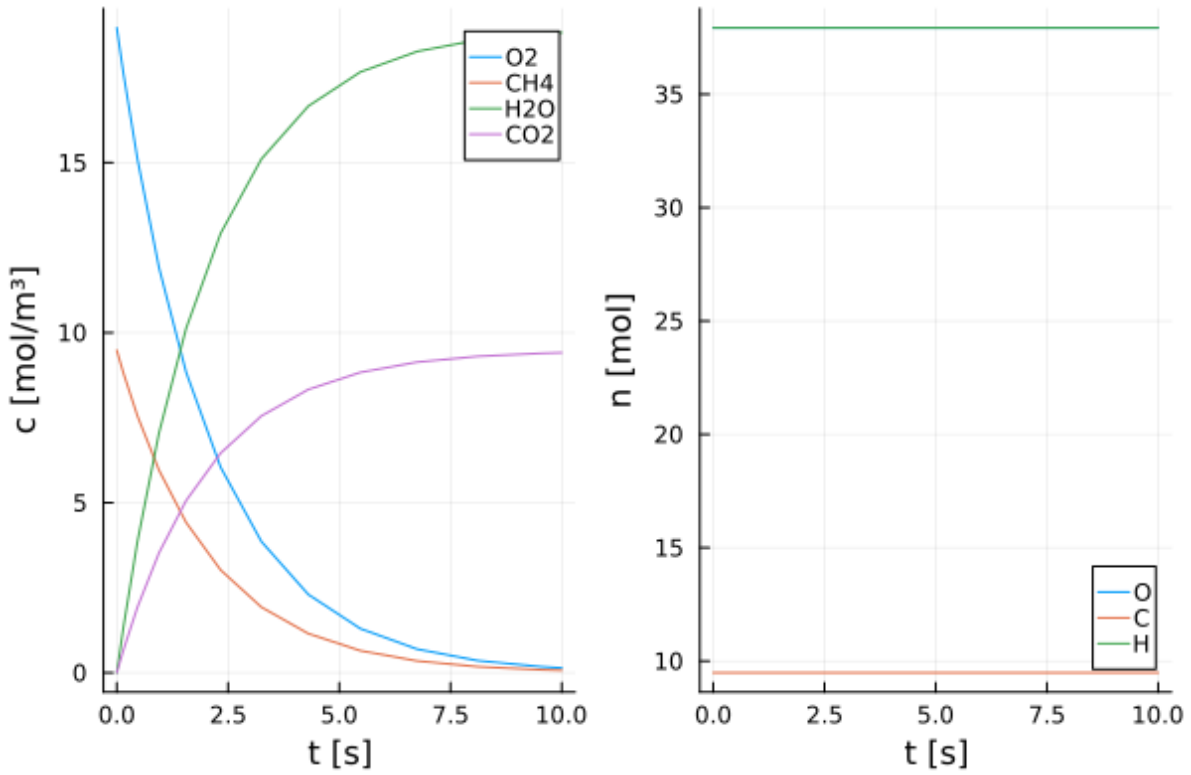

Figure 3.1: Output of the Julia script for a one-step mechanism with temperature T fixed at 1200 for an stoichiometric methane-air mixture. A second plot shows that atoms are being exactly preserved throughout the simulation.

we find the total concentration and thus each partial concentration can be computed. After some initial runs of the program an appropriate time interval for integration was found.

3.3. Thermodynamics with NASA-polynomials

It is however entirely unlikely that there is no temperature change in a combustion process. A slightly less implausible scenario would be the model of a perfectly isolated reactor vat. In this vat the release of energy from reactions causes a rise of temperature, which in turn increases the reaction speed, leading to a different prediction entirely. The change in temperature is governed by:

$$\frac{dT}{dt} = \frac{1}{c_{P,v}(T)} \frac{dH_v(T)}{dt} \tag{3.7}$$

where dH_v is the infinitesimal change in volumetric enthalpy. Here $c_{P,v}(T)$ is defined as the total volumetric heat capacity, which is the sum of the individual molar heat capacities of the species multiplied by their concentrations:

$$c_{P,v}(T) = \sum_i c_{P,M,i}(T) c_i \tag{3.8}$$

Now, the change in enthalpy is determined by the enthalpy change of the reactions. A chemical reaction j, in general, looks like:

$$\sum_{i \in \text{Reactants of } j} \nu_{ij} \mathcal{A}_i \Leftrightarrow \sum_{i \in \text{Products of } j} \nu_{ij} \mathcal{A}_i \tag{3.9}$$

where $\mathcal{A}_i$ signifies specie i and ν_{ij} the coefficient in front of that specie. Using this notation, and given the absolute molar enthalpy for species i as $H_{m,i}$, then the molar enthalpy change of reaction j is:

$$\Delta H_{M,j}(T) = \sum_{i \in \text{Products of } j} \nu_{ij} H_{M,i}(T) - \sum_{i \in \text{Reactants of } j} \nu_{ij} H_{M,i}(T) \tag{3.10}$$

and when multiplied by the speed of the reactions r_j, the contribution of reaction j to change in enthalpy per volume is found. Together:

$$\frac{dH_v}{dt} = \sum_{j \in \text{reactions}} \Delta H_{M,j} r_j \tag{3.11}$$

It should be noted that if $\vec{H}_m$ is the vector of all species their absolute molar enthalpies and $\Delta\vec{H}_m$ the vector of all reactions their enthalpy change, then the equations can be vectorized into $\Delta\vec{H}_m = \vec{H}_m^T S$. The whole ODE of the form $\dot{X} = f(X)$ can now be described as:

$$\text{RM} := \begin{cases} \frac{d\vec{c}}{dt} = S\vec{r} \\ \frac{dT}{dt} = \frac{1}{c_P} \vec{H}^T S\vec{r} \end{cases} \tag{3.12}$$

so that all which remains is to find the enthalpies $H_{m,i}(T)$ and the specific heats $c_{P,M,i}(T)$.

3.3.1. When V is Constant

When V is constant instead of P, two things change:

- $c_{V,v} = \sum_i [c_{P,M,i}(T) - R] c_i$ is used instead of c_P.
- Internal energy $U = H - RT$ is used instead of enthalpy.

The reason for the second change is that in both cases, the desired quantity is Q, which, from the first law of thermodynamics, is:

$$Q = \Delta U - W \tag{3.13}$$

When pressure is constant, work is done in the form of expansion of the gas: $W = P\Delta V$ and $Q = \Delta H$. When volume is constant, no work is done: $W = 0$ and $Q = \Delta U$.

3.3.2. NASA-Polynomials

There is a method [18] and there is data [23] that we can use to find the required thermodynamic variables. The method is NASA7 and it consists of, for each molecule in the analysis, a set of coefficients that can be used to compute thermodynamic variables, namely molar heat capacity $c^{\circ}_{P,M}$, absolute molar enthalpy H°_{M} and molar entropy S°_{M}, through the so called NASA-polynomials. The computation for $c^{o}_{P,M}$ is as follows.

$$\frac{c^{\circ}_{P,M}}{R} = a_1 + a_2 T + a_3 T^2 + a_4 T^3 + a_5 T^4 \tag{3.14}$$

The superscript "$\circ$" refers to the standard-state, which is to say that these variables are relative to a certain state $\circ$ that has pressure P°. For perfect gases, however, the heat capacities are independent of pressure, so there is only temperature to consider and we can drop the "$\circ$".

The molar enthalpy is now found by the integration of the heat capacity over temperature. This could be done all the way from $0\mathrm{K}$ upwards and it would be rather challenging. Lucky for us, any other reference point will do.

$$H_M(T) = \int_{T_0}^{T} c_{P,M} dT + H(T_0) \tag{3.15}$$

You could find it disturbing that we will compute the heat released by a reaction with some arbitrarily referenced variable—which has absolute in the name. And yet, because reactants and products are all referenced from the same point the arbitrariness cancels out; every absolute enthalpy in NASA7 is energy it takes to make that compound, starting from bare elements at $T_0 = 298\mathrm{K}$; the null temperature.

After integrating and then dividing by RT, and then choosing the sixth polynomial coefficient exactly right, we obtain:

$$\frac{H_M}{RT} = a_1 + \frac{a_2}{2} T + \frac{a_3}{3} T^2 + \frac{a_4}{4} T^3 + \frac{a_5}{5} T^4 + \frac{a_6}{T} \tag{3.16}$$

The molar entropy is found by the integration of the heat capacity divided by temperature:

$$S_M(T) = \int_{T_0}^{T} \frac{c_{P,M}}{T} dT + S(T_0) \tag{3.17}$$

Through a similar process as for enthalpy we get:

$$\frac{S_M}{RT} = a_1 \log T + a_2 T + \frac{a_3}{2} T^2 + \frac{a_4}{3} T^3 + \frac{a_5}{4} T^4 + a_7 \tag{3.18}$$

3.3.3. A Note on the Units of R

A trend among datasets for reaction mechanisms is to use a chemist's units for kinetic computations: $[\mathrm{mol}; \mathrm{cm}; \mathrm{K}; \mathrm{cal}; \mathrm{s}]$, and SI units for thermodynamic calculations: $[\mathrm{mol}; \mathrm{m}; \mathrm{K}; \mathrm{J}; \mathrm{s}]$—or so it would seem. There is in principle no need for a unit of energy at all in the thermodynamic determinations of dT and $d\vec{c}$. It is because the NASA-polynomials provide unitless quantities that the units of energy are only introduced when we multiply by R. This holds for all thermodynamic quantities: $c_{P,M}$, H_m, S_m, and for ΔG_m as well because it is calculated from H and S. Hence all energy units cancel out again when used in the ODE. In computing any variable of the ODE, R is factored out: for K_{eq} we divide ΔG by RT, and for dT we divide the energy unit of dH by the energy unit of c_P. So, it would have been all the same to pick units of energy such that $R = 1$.

However, in other places R does need to be defined: the activation energy E_a is given in units of $\mathrm{cal}/\mathrm{mol}$ so here we need $R_{cal} = 1.987\mathrm{cal}/\mathrm{molK}$, and meanwhile, in the ideal gas law used to find the total initial concentration, we need $R_J = 8.314\mathrm{J}/\mathrm{molK}$—if we do not want to use unusual units for pressure at least. This is not altogether contradictory, but R needs to be handled with care. The author stuck

with the concentrations in $[\mathrm{mol}/\mathrm{m}^3]$ that came out of the ideal gas law, which then made the Arrhenius equations rather unwieldy formulas and, in hind-sight, converting the initial concentrations and using the system $[\mathrm{mol}; \mathrm{cm}; \mathrm{K}; \mathrm{cal}; \mathrm{s}]$ everywhere else would have been a better way.

3.4. Thermodynamic Models

In what follows the results are shown of applying the theory of the previous section to simple reaction mechanisms, like the model we saw in section 3.2 which is elaborated, and like the slightly more complex four-step experiment which comes after.

3.4.1. A One-Step Mechanism with Thermochemistry

The example toy RM from earlier can be expanded upon. Suppose still $n_s = 5$ and $n_r = 1$ but now suppose there is extra variable $T(t)$. From the discussion above it follows that:

$$\frac{d}{dt}\begin{bmatrix}[\mathrm{N_2}]\\ [\mathrm{O_2}]\\ [\mathrm{CH_4}]\\ [\mathrm{H_2O}]\\ [\mathrm{CO_2}]\end{bmatrix} = \begin{bmatrix}0\\ -2\\ -1\\ 2\\ 1\end{bmatrix} k(T)[\mathrm{CH_4}]^{n_{\mathrm{CH_4}}}[\mathrm{O_2}]^{n_{\mathrm{O_2}}} \tag{3.19}$$

$$\frac{dT}{dt} = \frac{k(T)[\mathrm{CH_4}]^{n_{\mathrm{CH_4}}}[\mathrm{O_2}]^{n_{\mathrm{O_2}}}}{c_P(T)}(-2H_{\mathrm{O_2}}(T) - H_{\mathrm{CH_4}}(T) + 2H_{\mathrm{H_2O}}(T) + H_{\mathrm{CO_2}}(T)) \tag{3.20}$$

Which, after some programming—both for loading the database containing NASA-polynomials, and for computing the thermodynamic variables (see appendix B) and for integrating the ODE (see appendix C)—results in the evolution of species and temperature seen in figure 3.2. Although, in terms of what chemicals are present at the end, the results are the same as before, what becomes apparent is that

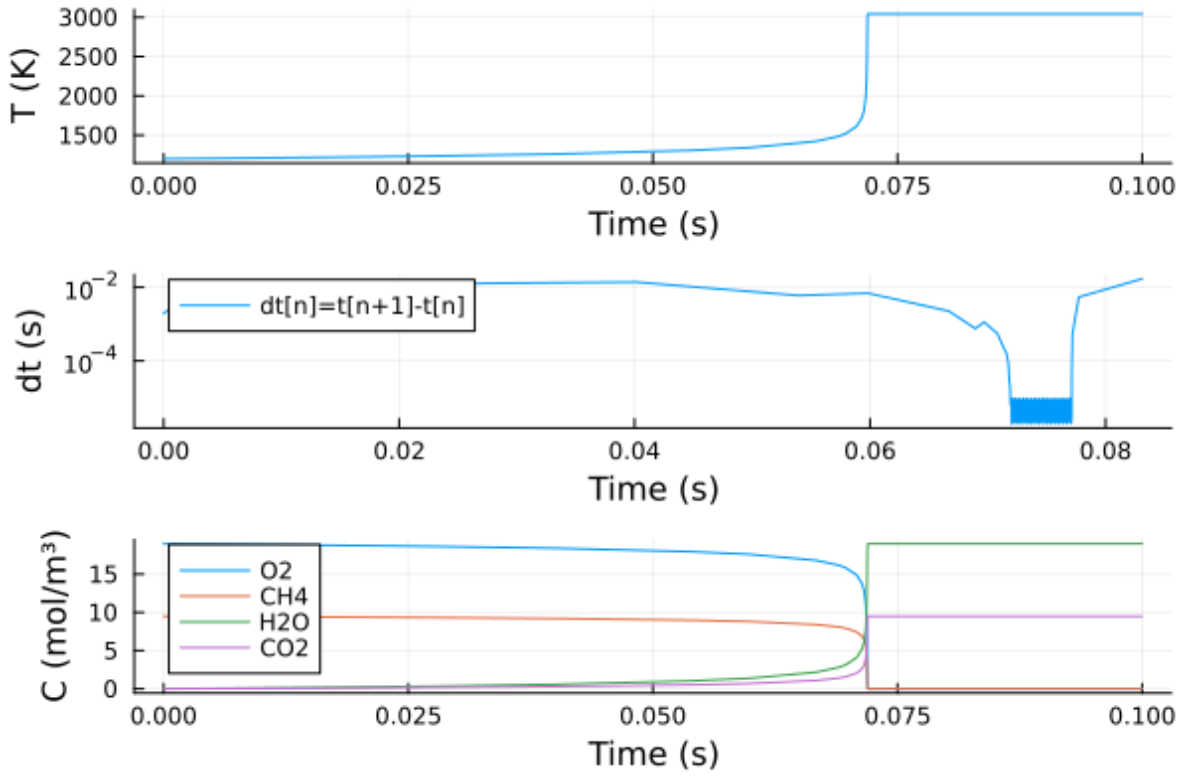

Figure 3.2: The explosive results of integrating the 0D one-step constant-P ODE problem with thermodynamics in Julia. Here $T(0) = 1200\mathrm{K}$, $P(0) = 10\mathrm{bar}$ and the initial gas was an stoichiometric methane-air mixture.

treating T as a constant severely misrepresents reality, where combustion is not a gradual process but rather an explosion. A self-reinforcing feedback loop of increasing T and r leads to a near instantaneous moment where all reactants burn up: the curve for T has an exponential rise and an abrupt discontinuous flattening end of combustion, when there are no more reactants.

It can be observed, in figure 3.2, that the final temperature of $3044\mathrm{K}$ is rather large. This can be explained by two factors: firstly, that in this model the reaction perfectly and completely combusts methane into product molecules of small formation enthalpy, this is unphysical. We will see later, in more realistic models, that several other species remain among the final products each having a higher formation enthalpy, which is where the heat should have went that made T so high in this simulation. That reduced mechanisms which only allow for complete combustion can have (a lot) higher final temperatures was found by others too [29].

Secondly, the initial temperature of $1200\mathrm{K}$ is extremely large. The large initial temperature is indeed the cause: in the simulation shown in figure 3.3, better agreement with reality is seen. Here initial conditions were chosen such that they exactly match available data from measurements (e.g. $298.15\mathrm{K}$ and $1\mathrm{bar}$] and the resulting flame temperature is accurate: literature states $2236\mathrm{K}$ [45], and the program finds $2319\mathrm{K}$. Another source that also used a heavily reduced mechanism and simulated unphysically complete combustion [29], agrees even better at $2326\mathrm{K}$. There is however no such thing as methane burning at room temperature, so to achieve combustion at this temperature a waiting period of $10^{24}\mathrm{s}$ was simulated, in which reactions slowly took place and temperature slowly rose.

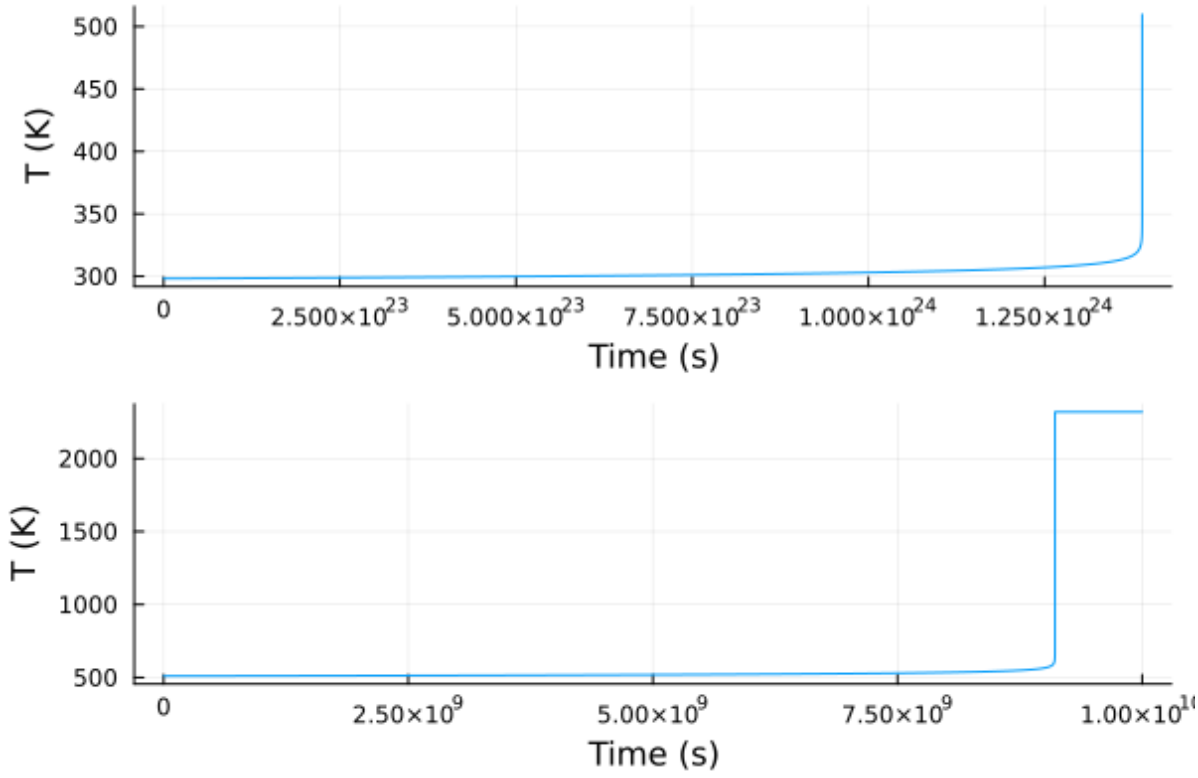

Figure 3.3: The results of a second integration of the 0D one-step constant-P ODE problem with thermodynamics in Julia. Now, $T(0) = 298.15\mathrm{K}$, $P(0) = 1\mathrm{bar}$ and the initial gas was an stoichiometric methane-air mixture. Because combustion started slow and ended fast, making the problem extraordinarily stiff, the integration was split in two.

3.4.2. A Four-Step Mechanism with Thermochemistry

Further elaboration of the reaction mechanism is possible to a so-called four-step mechanism, which has a number of reactions $n_r = 6$. Not 4, because 2 reactions in the mechanism are reversible, and the frontwards and backwards reactions are modelled separately—for now. Suppose also that $n_s = 7$, more exactly suppose the mechanism knows these species: $\mathrm{N_2}$, $\mathrm{O_2}$, $\mathrm{CH_4}$, $\mathrm{H_2O}$, $\mathrm{CO_2}$, CO and $\mathrm{H_2}$.

These species then act in the following reactions:

$$\begin{aligned} 0.5O_2 + CH_4 &\implies CO + 2H_2 \\ CH_4 + H_2O &\implies CO + H_2 \\ 0.5O_2 + H_2 &\iff H_2O \\ H_2O + CO &\iff CO_2 + H_2 \end{aligned}$$

Suppose this, and what follows is already a quite complex system of ODEs. It is not stated here but can be distilled from the code in appendix D. What is still readable is the expression for the stoichiometric matrix S and the concentration vector $\vec{c}$:

$$S = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ -0.5 & 0.0 & -0.5 & 0.5 & 0.0 & 0.0 \\ -1.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & -1.0 & 1.0 & -1.0 & -1.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & -1.0 \\ 1.0 & 1.0 & 0.0 & 0.0 & -1.0 & 1.0 \\ 2.0 & 3.0 & -1.0 & 1.0 & 1.0 & -1.0 \end{bmatrix} \quad \vec{c} = \begin{bmatrix} [N_2] \\ [O_2] \\ [CH_4] \\ [H_2O] \\ [CO_2] \\ [CO] \\ [H_2] \end{bmatrix} \tag{3.21}$$

Experiments have been done to determine the Arrhenius coefficients of these 6 reaction mechanisms [22, 20], which can be used to write code that runs this problem (appendix D). The results are shown in figure 3.4, and they would seem valid. Firstly, because the amount of each atom is preserved (as can be read in the print statements for initial and final amount of atoms present, which is included in appendix D). Secondly, because when the results of the four-step simulation are compared to figures 3.1 and 3.2 it shows that in all simulations the final states are the same—when we look at the final value for T and the final species concentrations they are the same. This validates that something is correct

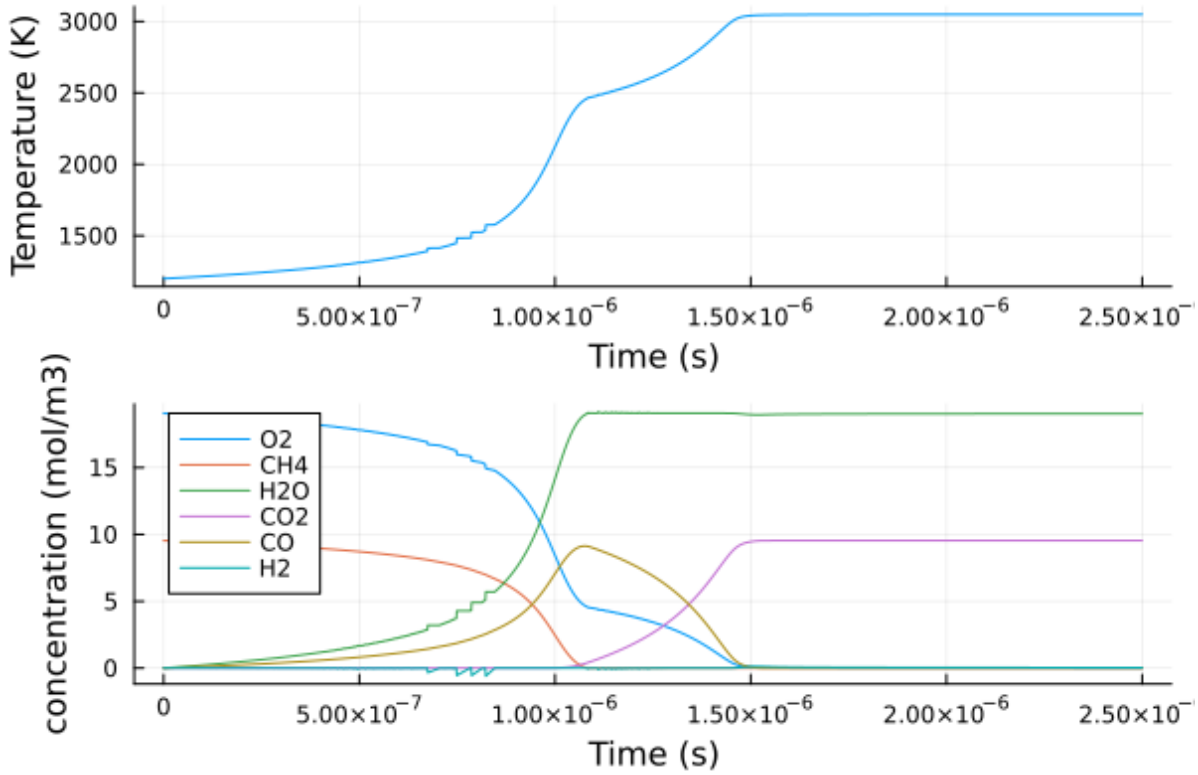

Figure 3.4: The results of integrating a 0D four-step constant-P model with thermodynamics. Initial conditions where $1200\mathrm{K}$ and $10\mathrm{bar}$ for a stoichiometric methane air mixture. That it looks odd is most likely because it is a 'Frankenstein model' composed of several non-calibrated theories about the Arrhenius parameters of different reactions.

about it, but the four-step mechanism here is quite bad. In fact, the incorrectness of the model is so severe that this hinders the integration in Julia: the solve took a shocking 54 seconds of runtime. The most probable cause is that the ODE is nearly too unstable to integrate, which is the result of mixing two separate kinetic theories for different subsets of reactions without calibration. Beyond this, the same strangely high final T as before is observed, which has the same explanation as it did for the one-step thermodynamic model.

3.5. Complex Chemical Kinetics

Things are rarely simple and so too are chemical reactions often more complex than the simple Arrhenius equation. The model we are working towards will soon be too sophisticated for only equation 3.1 to suffice. Two reaction types, three-body reactions and falloff reactions, which are part of most chemical models, are discussed first. After this the reversibility of reactions is treated.

3.5.1. Third-Body Reactions

Some reactions require a medium, which is itself unchanged, but facilitates the reaction. In these reactions the medium M is to be treated as an extra reactant in equation 3.1, which has a concentration of

$$[M] = \sum f_i[X_i] \tag{3.22}$$

which is simply the sum of the concentrations of all species $[X_i]$, times their efficiencies f_i which are 1 unless specified otherwise.

3.5.2. Falloff Reactions

Falloff reactions are reactions that have different rates at low pressure (k_o) and high pressures (k_∞). They transition in the Lindemann equation:

$$k = \frac{k_\infty}{1 + \frac{k_\infty}{k_o[M]}} F \tag{3.23}$$

where $F = 1$ or is determined by a calculation requiring the Troe parameters a, b, c, d, they are used to find F_{cent}:

$$F_{cent} = (1 - a)e^{-T/b} + ae^{-T/c} + e^{-d/T} \tag{3.24}$$

which, combined with reduced pressure $P_r = k_o[M]/k_\infty$ gives us F via

$$\log F = \frac{\log F_{cent}}{1 + [(\log P_r + C)/(N - 0.14(\log P_r + C))]^2} \tag{3.25}$$

with $N = 0.75 - 1.27 \log F_{cent}$ and $C = -0.4 - 0.67 \log F_{cent}$.

The way the computation of rate constants k for these reactions is done is very much a blunt engineering approach. The formulas are specific to the types of datasets that will be used and can be found online [43].

3.5.3. Reversible Reactions

All that happens in the physical world is reversible, some things are just not that likely to reverse. So too can chemical reactions happen in both directions, and there are often cases where the reversible reaction is probable enough to require accounting for in simulation. In these cases the forward rate constant k_f, which is computed as usual, and the reverse rate constant k_r will obey a ratio K_{eq} that can be known. From

$$K_{eq} = \frac{k_f}{k_r} \tag{3.26}$$

The k_f is computed through equation 3.1 and thus, using K_{eq}, the reverse reaction rate is obtained. What then, is K_{eq}? The formula for the equilibrium constant can be derived through a thermodynamic theory which is rather deep [16]: at equilibrium the “available energy” G of a system is minimal and this determines the equilibrium ratio between products and reactants, namely the ratio that carries the

lowest value of G. The difference in G between products and reactants determines how fast the reaction works to move concentrations towards equilibrium. Long story short:

$$K_{eq,P} = e^{-\Delta G/RT} \quad \text{where} \quad \Delta G = \Delta H - T\Delta S \tag{3.27}$$

Here ΔG is the standard Gibbs free energy change of the reaction for which, like we saw in section 3.3.2, usual notation involves a superscript $\circ$ on the thermodynamic variables G, H and S to signify a standard state—it is omitted here too because ideal gases have no pressure dependence in any of these variables.

Equation 3.27 is for constant pressure, at constant volume we use the standard Helmholtz free energy change of the reaction:

$$K_{eq,V} = e^{-\Delta F/RT} \quad \text{where} \quad \Delta F = \Delta U - T\Delta S \tag{3.28}$$

3.6. A Sophisticated Zero-Dimensional Model

Here we will use a complete reaction mechanism; 'complete' meaning of sufficient complexity to accurately approximate the chemical evolution in the system for risk analysis in process design. It is a reaction mechanism still, but with a large number of species, and an equally large number of variables thus. It is a very complex reaction mechanism, especially when a very large number of reactions is under consideration. These reaction mechanisms can however all be treated with the mathematics developed before. Although from a certain size onwards the chemists do not count the amount of steps and simply name the whole a skeletal mechanism.

3.6.1. Skeletal Mechanisms

The idea that chemical reactions may happen in chains originates with Max Bodenstein who, in 1913, conjectured that the products of primary reactions may react again in the so called secondary reactions [6]. Over the last century the theory matured with most notably the work of Nikolay Semyonov and Cyril Norman Hinshelwood who shared a Nobel prize for their work on the subject in 1956 [35]. Nowadays these models are commonly referred to as skeletal mechanisms but perhaps tree mechanisms or forest mechanisms would be their more appropriate name, because their structure has roots in reactants that through initiation reactions result in intermediaries which through propagation reactions branch of into even more intermediary species or eventually through termination reactions end the mechanism by creating products.

Any and all molecule was the result of some chemical reaction and in a way the universe is a big chain reaction system: it is obvious that a skeletal mechanism can be made more complex ad nauseam. There is balance to be found between complexity, accuracy and runtime. A question for the chemist: which species will appear and how; what can and cannot be omitted from a reaction mechanism? This question has been answered [51].

3.6.2. Databases and YAML-files

So far, the most complex systems shown have contained less than 10 species and reactions. It is not in itself a mathematical challenge to make our state parameter bigger. However, as was shown by the failed experiment of section 3.4.2, mixing multiple theoretic models requires prudence. The whole of the reaction mechanism must be properly calibrated and optimised, which many people have done and their databases containing RMs are available on the internet. For the purpose of chemical analysis, a great many institutions maintain files that contain all the needed properties of all the needed species and reactions to simulate some process, whichever relevant to their research. The database GRI-Mech 3.0 was developed for hydrocarbon combustion processes by fitting all kinetic and thermodynamic parameters of a reaction mechanism containing 53 species and 325 reactions to a plethora of experimental data [42], and the database is available online for free. There are others, a lot of which can be found on `https://chemistry.cerfacs.fr/`, and some of them are far more complex than GRI-Mech 3.0. Curran's detailed mechanism for gasoline kinetics, for example, contains 1034 species and 8453 reversible reactions [10].

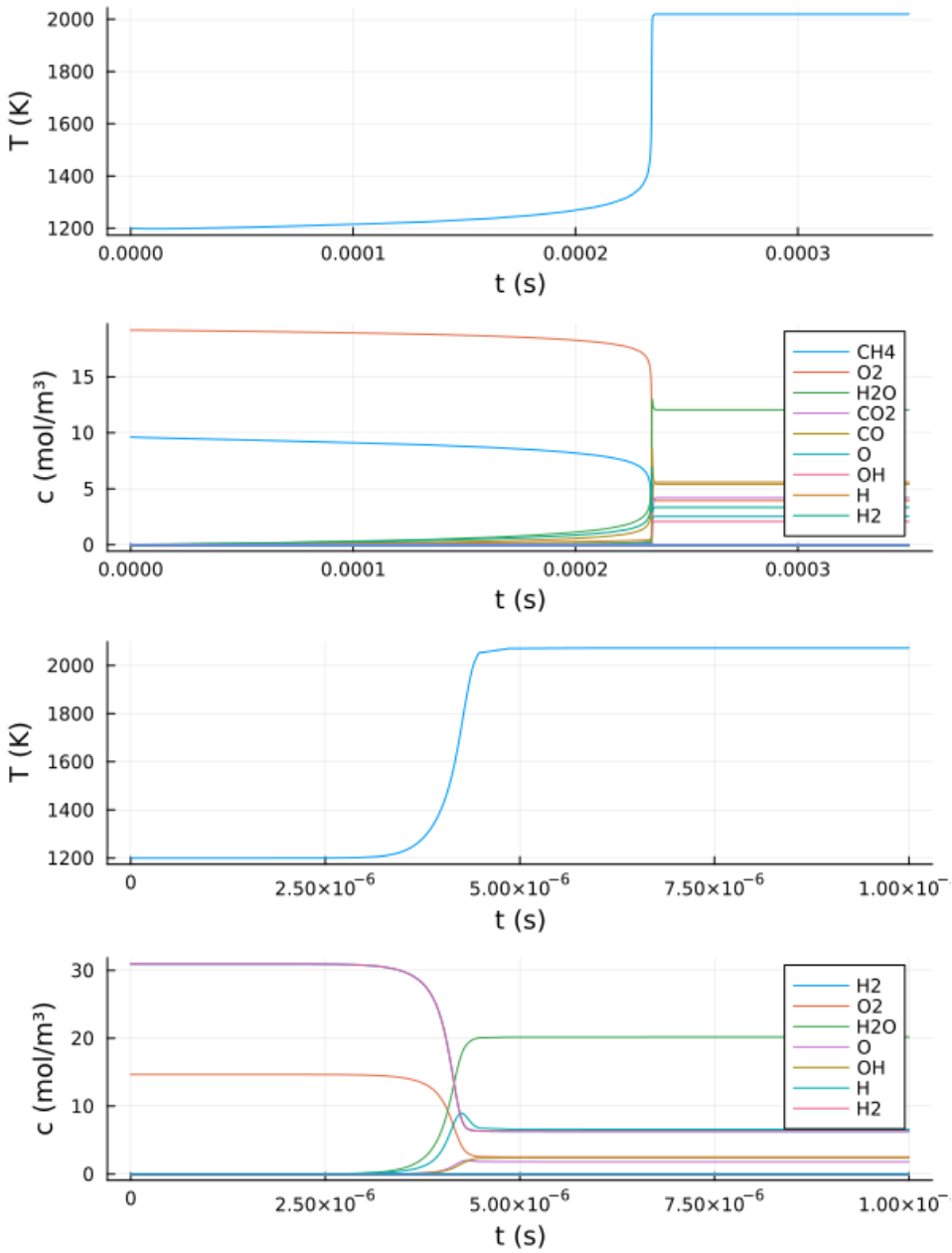

Figure 3.5: The results of integrating a 0D constant-V models starting at $T(0) = 1200\text{K}$ and $P(0) = 10\text{bar}$, with above a simulation using GRI-Mech 3.0 with a stoichiometric air-methane mixture and below a simulation employing Laurent's mechanism with a stoichiometric hydrogen-air mixture.

Ideally all of the models discussed in this report would have been run on GRI-Mech 3.0, but there were significant hardware limitations as the laptop of the author is quite old. Concessions had to be made as model complexity increased. There are four reaction mechanisms which have been used in the programs of this thesis:

- GRI-Mech 3.0 was developed with the intent of it having the minimum of complexity required for the combustion of hydrocarbon fuels, this minimum is 53 species and 325 reactions [42].
- Tianfeng Lu managed to reduce the mechanism of GRI-Mech to 30 species and 184 reactions and his mechanism is specifically designed for methane combustion [28].
- Charlelie Laurent made a similar effort which resulted in methane specific mechanism containing 17 species and 82 reactions [26].

- The 2-step mechanism has 6 species and was developed Franzelli et al. and although a smaller mechanism exists, this is the smallest mechanism of which its creators do not explicitly discourage its use [15]. It is an exception to the rule that reaction mechanism datasets use [$\mathrm{cal}/\mathrm{mol}$] for E_a, here activation energy is in units of [J/mol].

Modern reaction mechanism databases usually come in .YAML format—human-readable data serialization files designed to represent structured data [3]. Over the last decennium or so, .YAML has become the de facto standard for defining chemical simulation inputs, because it was specifically made for expressing the complex, hierarchical information, such as a chemical system, in a way that is both accessible to coders and readily parsed by computers. This is exemplified by popular frameworks such as Cantera [17] and ChemKED [47], which employ .YAML to store thermochemical properties, reaction mechanisms, and solver parameters. To model well based on such inputs, it is necessary to write code that can efficiently read .YAML files, store their contents in properly so the data is available for physical and chemical computations.

3.6.3. A Complete Model and Simulation

The figure at the end of this chapter (figure 3.5) is the result of code found in appendix E, and what is happening should not be exciting—in terms of new physics or chemistry, at least—to those who read the earlier parts of this chapter. All is still ruled by equation 3.12, initial concentrations are still chosen as a stoichiometric methane-air mixture and still $1200\mathrm{K}$ and $10\mathrm{bar}$. Nothing remarkable about it but the fact that it runs, which was exciting to the author. The program works universally, meaning that it runs with several different .YAML files and initial conditions. While it is conceptually still wholly similar to the one-step and four-step models of section 3.4, this sophisticated model has as many steps and species as the databases the user provides.

A considerable amount of work went into adapting the previous code so it would run for an arbitrary reaction mechanism put in as a .YAML file. A Julia file was made to contain all the extracting, parsing, parameter constructing and property calculating functions needed to create the ODE that can be fed to a solving algorithm. This program has become quite elaborate (>1000 lines) and therefore is not included in the appendices, but it can be found as ”Chemistry.jl” on the GitHub page `https://github.com/Jelterminator/Chemical-Combustion`, along with some code-documents that exemplify its use.

To achieve stable integration, the problem of overshoot into the negative domain had to be dealt with: certain computations for reaction rates break down on negative concentrations. To prevent such overshoot there exists a method [41], implemented in the DiffEqCallbacks.jl module—and it still was insufficient for implicit methods that may reach the negative in intermediate calculations within one timestep. Even a clamp on zero itself still was not robust enough and so the maximum of concentration values and the arbitrarily small 10^{-100} was used in computations, creating a numeric artifact which heats to simulation when the number of iterations becomes very large. Another Julia module that cannot be left unmentioned is, of course, DifferentialEquations.jl [38]. The readily available solver algorithms provided in their code made short work of even the complete GRI-Mech 3.0 mechanism, specifically an A-L stable stiffly-accurate 5th order eight-stage ESDIRK method, which is called “ESDIRK54I8L2SA”, proved useful.

Correctness of the Skeletal Models

The Chemistry.jl package was remodelled for constant volume, both in regard of the closed spaces that will be modelled next chapter, and in regard of the temperature which is lower than theory typically predicts. The transition changed very little about flame temperatures seen in this research, remarkably they got even lower. The change in final T was from $2050\mathrm{K}$ for isobaric combustion to $2016\mathrm{K}$ for isochoric combustion of stoichiometric methane-air. Meanwhile theory predicts a change from roughly $2200\mathrm{K}$ for constant P to $2700\mathrm{K}$ for constant V [33]—because of the extreme initial conditions used there is but little literature that states the adiabatic flame for the right types of initial conditions, it is guesswork. What is not guesswork is that T ought to have been higher after the transition from isobaric to isochoric was made, that it was not indicates a problem in the model.

As a general rule of thumb, simple models yield adiabatic flame temperatures which are higher than what theory predicts, while complex models yield flame temperatures which are lower than the theoretical values. A remark many theorists make when determining an adiabatic flame temperature is that

just about anything will lower the final value. Yet it is strange, because unlike in life, on a computer it is very easy to create perfect adiabatic conditions. Heat cannot escape if the code does not let it, which is reason to expect final temperatures higher than literature values. The energy cannot leave the system, so where does it go and why does it not turn to heat?

The answer: it goes into dissociation. The energy cannot escape out of the system, but it can escape into the chemistry. GRI-Mech 3.0 certainly allows for complex enough processes that dissociation; the breaking apart of stable molecules into parts with a higher formation enthalpy, that it can happen. There is double confirmation of this hypothesis for the missing energy: firstly, when a reaction mechanism without the complexity needed for dissociation is employed the final $T = 2784\mathrm{K}$ which appears very correct for an isochoric adiabatic flame temperature starting from $1200\mathrm{K}$. Secondly, when all energy in the system is tracked it clearly shows there is no bug and energy is preserved, it all goes into the formation of dissociated radicals, as can be seen in figure 3.6.

What is the cause for this model overestimating the dissociation activity to the point where it has capped how high T can go? It remains the subject of speculation.

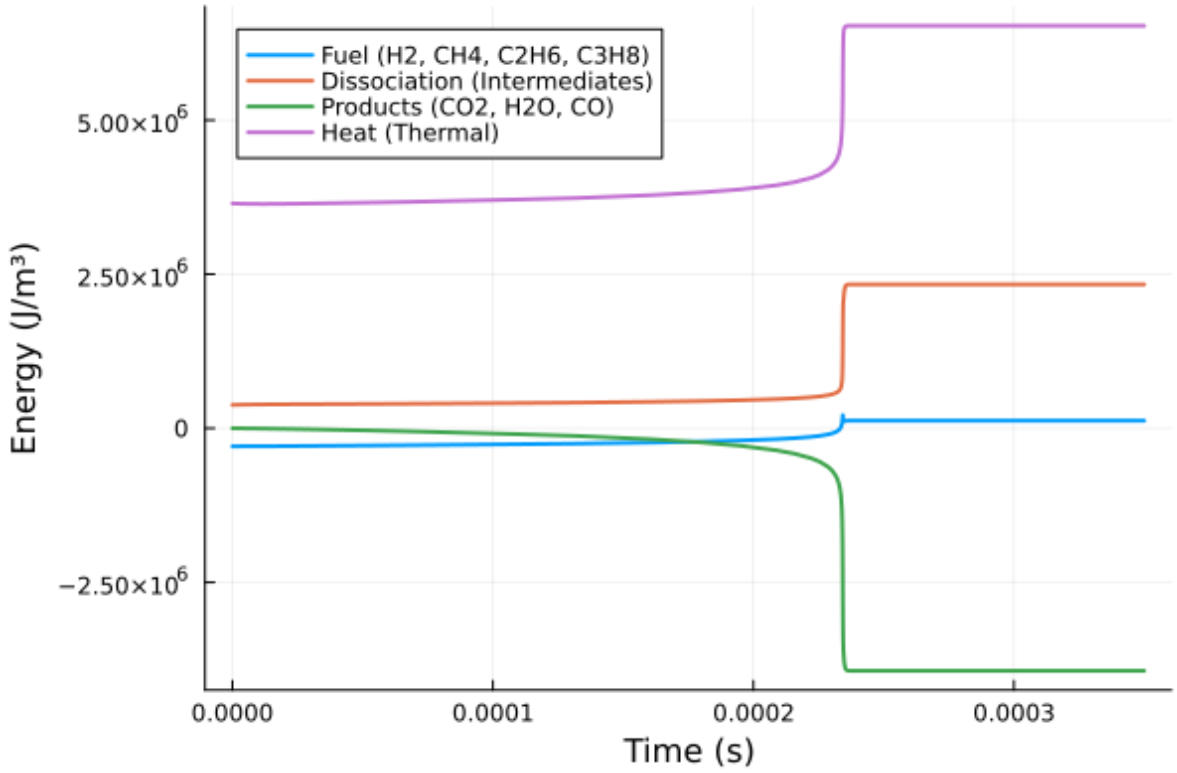

Figure 3.6: The methane simulation from figure 3.5 is here portrayed again but from a different angle: it shows all energy present in the system, and in what form it is present. The orange line of the dissociated starts rising later and sharper than the purple line of heat, and it clearly consumes a substantial part of the energy released in the exothermic combustion product creation.

4

Networks of Reactors

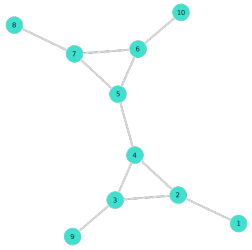

Figure 4.1: This is a graph. It was not used in this thesis: loose ends where a node connects to the graph by one edge alone tend to create numerical instabilities.

This chapter starts with an argument for the use of graph theoretical models, because why a graph? Because a continuous model would be enormously computationally expensive, is why. Especially memory becomes a problem: a 0-dimensional problem may involve hundreds of chemical species (each a variable of the problem) and thousands of reactions, so to transition into a spatial model where a grid of these cellular problems is used—and where each point gets its own hundreds of variables—would be to ask the computer to integrate a stiff system of differential equations involving millions of variables, which is a significant computational task. Because every added node brings a lot of extra variables, every node must be selected with care so its purpose is not outweighed by its computational burden. The minimal model to describe a combustion process in space, it is a graph.

4.1. Graph Theoretic Models for Reactor Networks

A graph $G = (V, E)$ is set containing a number n_n of points known as nodes v_k that make a set V, which are connected by a number n_e of lines known as edges e_l that make a set E. Each edge e_x connects exactly 2 nodes, $e_x = (v_y, v_z)$ for some nodes v_y, v_z. Instead of the formal v_k, a node can also be referred to simply as k.

All of the descriptions of the state of a system up to now have only contained one node, and for each extra node we will need a complete extra state vector to describe the new reactor, so that for a whole graph we get a set of state vectors $\vec{X}_k(t)$ which we will together refer to as the network state matrix U.

$$U(t) := [\vec{X}_1(t)\vec{X}_2(t)\dots\vec{X}_{n_n}(t)]^T \tag{4.1}$$

The archetypical ODE ruling the system is most easily expressed by individual equations for each node. The state equation for each node always contains an intrinsic term representing the reactions taking place (what was developed thus far), and a sum of transport terms; one for each edge connecting the node.

$$\frac{d\vec{X}_k}{dt} = F_{\text{intrinsic}}(\vec{X}_k) + \sum_{k':(k',k)\in E} F_{\text{transport}}\left(\vec{X}_k, \vec{X}_{k'}, (k,k')\right) \tag{4.2}$$

The purpose of the current section is to develop an expression for $F_{\text{transport}}(\vec{X}_k, \vec{X}_{k'}, (k,k')) := F_{kk'}$. It will be a multi-valued function: $F_{kk'}$ takes the states $\vec{X}_k$ and $\vec{X}_{k'}$, and the properties of their connecting

edge (k, k'). The function returns the change per unit-time to the state vector of node k caused by transport over edge (k, k').

4.2. Diffusion & Convection

The basic assumption of this thesis is that a node is a perfectly stirred and perfectly thermally isolated reactor, which can by described completely by a state vector containing the specie concentrations and temperature. These variables evolve because of internal production or consumption of species and heat in reactions—the subject of chapter 3—and because heat or molecules are being exchanged with other nodes. In reality the exchange of these quantities between systems can happen through convection, diffusion and conduction. We will ignore convection until section 4.3, leaving us with diffusion of species and conduction of heat. Fourier's Law of Heat Conduction [14], and Fick's Law of Diffusion [13], would typically dictate these processes—but not in our case, because these laws rule over reality which is not a graph, but a continuous space:

$$q = -\kappa \nabla T \quad ; \quad J = -\mathbb{D} \nabla c \tag{4.3}$$

The dimensional reduction of such equations has been written about in great detail (and at excruciating levels of abstraction, I might add) in many papers [31, 19, 4], so the brief and incomplete justification of the next section will suffice for this research.

4.2.1. Graph Laplacian Operators

When we realise that:

$$\frac{dT}{dt} = -\frac{1}{c_{P,v}} \nabla q \quad \Longrightarrow \quad \frac{dT}{dt} = \frac{1}{c_{P,v}} \nabla(\kappa \nabla T) \tag{4.4}$$

$$\frac{dc}{dt} = -\nabla J \quad \Longrightarrow \quad \frac{dc}{dt} = \nabla(\mathbb{D} \nabla c) \tag{4.5}$$

we recognise that we are dealing with Laplacians, which most generally express laws of diffusion for certain preserved quantities with distributions ϕ. With a variable parameter $D(\phi, \mathbf{r})$ called the diffusivity and $\mathbf{r}$ the general equation is:

$$\frac{\partial \phi(\mathbf{r}, t)}{\partial t} = \nabla \cdot \big[D(\phi, \mathbf{r}) \, \nabla \phi(\mathbf{r}, t) \big] = \Delta_D \phi(\mathbf{r}, t) \tag{4.6}$$

The discrete Laplace operator is the graph theoretical version of the Laplace operator and it is obtained by applying a finite difference method to a Laplacian equation as will follow. In this derivation, $\phi(k, t) := \phi_k$ is a finite number resembling the concentration of an unspecified preserved quantity at node k.

Suppose there are two points named x_1 and x_2, separated by a distance L_{12}. Now, let each point represent both a node in graph-space and a volume in actual-space, and suppose these volumes are connected by some edge in graph-space; that their volumes are touching over a surface A_{12} in space. If the transport is ruled by an equation such as 4.6, then of interest is the quantity $\partial \phi(x_{1.5}, t)/\partial t$ at the surface where the two volumes meet. This can be approached with discrete differential schemes which is completely accurate if the slope between x_1 and x_2 is linear.

First a central difference method at $r = (x_1 + x_2)/2 := x_{3/2}$ is applied, and next a forward and backward difference scheme replace the nabla operators at respectively $r = x_1$ and $r = x_2$:

$$\frac{\partial \phi(x_{3/2}, t)}{\partial t} = \nabla \cdot \big[D(\phi, \mathbf{r}) \nabla \phi(\mathbf{r}, t) \big] \Big|_{r = x_{3/2}} \approx \frac{1}{L_{12}} \big[D_2 \nabla \phi \Big|_{r = x_2} - D_1 \nabla \phi \Big|_{r = x_1} \big] \tag{4.7}$$

$$\approx \frac{1}{L_{12}^2} \Big[D_2 \big(\phi_2 - \phi_1 \big) - D_1 \big(\phi_1 - \phi_2 \big) \Big] = \frac{D_1 + D_2}{L_{12}^2} \left[\phi_2 - \phi_1 \right] \tag{4.8}$$

Since this is the transport per surface area, the connecting surface A_{12} should be factored in to obtain the complete transport through the edge between x_1 and x_2; the flux Φ_{12}:

$$\Phi_{12} \approx A_{12} \frac{D_1 + D_2}{L_{12}^2} \left[\phi_2 - \phi_1 \right] = w_{12} \left[\phi_2 - \phi_1 \right] \tag{4.9}$$

So far we considered intrinsic quantity ϕ, yet flux through an edge adds extrinsically to the content of a node—let Θ be the extrinsic quantity such that it is related to its intrinsic counterpart by a node-specific container constant Λ_k: $\Lambda\phi = \Theta$.[1]

$$\Lambda_k \frac{d}{dt}\phi_k = \frac{d}{dt}\Theta_k = \sum_{k':(k,k')\in E} \Phi_{kk'} \tag{4.10}$$

The total transport is the sum of transports over all edges, which yields the graph Laplacian equation:

$$(\Delta_D \phi)(k,t) = \frac{1}{\Lambda_k} \sum_{k':(k,k')\in E} w_{kk'} \left[\phi_{k'} - \phi_k\right] \quad \text{with} \quad w_{kk'} = A_{kk'} \frac{D_k + D_{k'}}{L_{kk'}^2} \tag{4.11}$$

Furthermore:

- $\Delta_D \phi$ is the Laplace operator for graphs on the vector ϕ of all nodes' ϕ_k. An element $(\Delta_D \phi)(k,t)$ tells the total change in ϕ_k and has units of $[\mathrm{mol/m^3/s}]$ or $[\mathrm{K/s}]$.
- ϕ_k is an intrinsic variable representing some preserved quantity in node k, with units of either $[\mathrm{mol/m^3}]$ or $[\mathrm{K}]$.
- $A_{kk'}$ and $L_{kk'}$ are the connecting surface area at interface (in $[\mathrm{m^2}]$) and the distance between centres (in $[\mathrm{m}]$) of node k and k'.
- $\Lambda_k := \Theta_k/\phi_k$ is the container coefficient of node k: it is either a volume in $[\mathrm{m^3}]$ or a heat capacity in $[\mathrm{J/K}]$.
- D_k is the thermal/mass diffusivity in k, a variable which has units of $[\mathrm{m^2/s}]$ for mass diffusion or $[\mathrm{W/mK}]$ for thermal diffusion.
- $w_{kk'}$ is the form factor of the edge, which has the same units as the diffusivity which is a factor in it. It is the weight of the edge.

4.2.2. Vectorization of Discrete Laplace Equations

Given a simple graph $G = (V, E)$ describing a network, with n_n vertices $v_1, \ldots, v_{n_n}$, its $n_n \times n_n$ Laplacian matrix L completely captures the edge dynamics. If the diffusivities and form factors are all 1, then it can be defined element-wise as

$$L_{i,j} := \begin{cases} \deg(v_i) & \text{if } i = j \\ -1 & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise,} \end{cases} \tag{4.12}$$

or equivalently, L can be defined as

$$L := D - A \tag{4.13}$$

where

- A is the weighted adjacency matrix: $A_{ij} := w_{ij}$ if nodes i and j are connected ($A_{ij} = 0$ otherwise)
- D is the degree matrix: $D_{ii} := \sum_j w_{ij}$ (diagonal matrix with row-sums on diagonal)

The second definition of L works too when the edges have weights w_{ij}. L encodes transport:

$$\frac{d\phi_k}{dt} = \sum_{k'} w_{kk'} \left[\phi_{k'} - \phi_k\right] = \left[L \left(\phi_1 \phi_2 \ldots \phi_{n_n}\right)^T\right]_k \tag{4.14}$$

In this equation $[*]_k$ is to be read as the k'th row of $*$.

Given the matrix L, we can rephrase the discrete Laplace equation 4.11 as:

$$\dot{U} = LU \tag{4.15}$$

[1] Think of $\Lambda\phi$ as an abstraction of either volume times concentration or heat capacity times temperature.

That U returns in our discussion is no surprise: one column of U describes a preserved quantity in each node, exactly what the whole set $\{\phi(i,t) : i = 1 : n_n\}$ does. Because ϕ was arbitrary and there are many preserved quantities, we can stack an instance of equation 4.14 for each of the quantities to obtain equation 4.15 for the network state matrix.

4.2.3. Fourier's Law of Heat Conduction

The heat flux into node k from node k' is:

$$q_{kk'} = w_{kk'}(T_{k'} - T_k) \quad \text{with} \quad w_{kk'} = A_{kk'}\frac{\kappa_k + \kappa_{k'}}{L^2_{kk'}} \tag{4.16}$$

where κ_x is the thermal conductance at node x and $w_{kk'}$ is the form factor of edge (k, k'). If the change in heat content of node k is $dQ_k = V_k c_{V,k} \cdot dT_k$, then:

$$\frac{dQ_k}{dt} = \sum_{k':(k',k)\in E} w_{kk'}(T_{k'} - T_k) \tag{4.17}$$

becomes:

$$\frac{dT_k}{dt} = \frac{1}{V_k c_{V,k}} \sum_{k':(k',k)\in E} w_{kk'}(T_{k'} - T_k) \tag{4.18}$$

which can be vectorized if we define $\vec{T} = [T_1 T_2 \dots T_{n_n}]^T$ as the vector of temperatures, and $(Vc_P)^{-1} = \mathrm{diag}\{(V_1 c_{P,1})^{-1}, (V_2 c_{P,2})^{-1}, \dots, (V_{n_n} c_{P,n_n})^{-1}\}$ as the diagonal matrix of inverted volume-specific heat products. We combine with L_κ; the weighted Laplacian matrix, to obtain:

$$\frac{d\vec{T}}{dt} = (Vc_P)^{-1} L\vec{T} \tag{4.19}$$

Calculating Thermal Conductivity κ

A theory for the determination of thermal conductivity was developed separately by Sydney Chapman and David Enskog in 1916 and 1917 respectively [9, 12], however, the equation used here is from a website [44]. The thermal conductivity of a pure ideal gas is computed as:

$$\kappa = \frac{f}{3d^2}\sqrt{\frac{k_B^3 T}{\pi^3 m}} \tag{4.20}$$

where all units are SI and:

- f is degrees of freedom of the gas molecules
- d is effective molecular diameter
- m is the mass of one molecule

Although there are methods for the conductivities of gas mixtures [11], the choice was made to simply compute the conductivity of a pure nitrogen gas to simplify calculations and enhance computational performance.

4.2.4. Fick's Law of Diffusion

Similarly, the transport rate of specie i to node k from k' because of diffusion is:

$$J^i_{kk'} = w^i_{kk'}(c^i_{k'} - c^i_k) \quad \text{with} \quad w^i_{kk'} = A_{kk'}\frac{\mathbb{D}^i_k + \mathbb{D}^i_{k'}}{L^2_{kk'}} \tag{4.21}$$

where again $w^i_{kk'}$ is an edge specific weight. In this instance $\mathbb{D}^i_k$ is the (possibly specie and node dependent) mass diffusivity. J quantifies an amount of molecules per time unit,[2] and as such cannot be added to a concentration; it is the change in the amount of specie i in the reactor. $Jdt = dn =$

[2] Although typically J is used to signify diffusion flux *per surface* here the surface has been factored in already through w.

$d(V_k c_k) = V_k dc_k$, with V_k the volume of reactor k. Thus, the rate of change in the concentration of specie i at node k by diffusion is:

$$\frac{dc_k^i}{dt} = \frac{1}{V_k} \sum_{k':(k',k)\in E} w_{kk'}^i (c_{k'}^i - c_k^i) \tag{4.22}$$

which can be vectorized if we define $\vec{c}^i = [c_1^i c_2^i \dots c_{n_n}^i]^T$ as the diffusivity weighted vector of concentrations, and $V^{-1} = \mathrm{diag}(1/V_1, 1/V_2, \dots, 1/V_{n_n})$ as the diagonal matrix of inverted volumes. We combine with $L_{\mathbb{D}}$; the weighted Laplacian matrix, to obtain:

$$\frac{d\vec{c}^i}{dt} = V^{-1} L_{\mathbb{D}} \vec{c}^i \tag{4.23}$$

Calculating Diffusivity $\mathbb{D}$
Again Chapman-Enskog theory is used and again the formula is from a website [50], resulting in the following equation to compute the diffusivity of specie A in a gas of specie B:

$$\mathbb{D} = \frac{A_{\mathbb{D}} T^{\frac{3}{2}}}{p d_{AB}^2 \Omega(T^*)} \sqrt{\frac{1}{M_A} + \frac{1}{M_B}} \tag{4.24}$$

$$A_{\mathbb{D}} = \frac{3}{8} k_b^{\frac{3}{2}} \sqrt{\frac{N_A}{2\pi}} \tag{4.25}$$

In this equation

- $d_{AB} = (d_A + d_B)/2$ is the averaged molecular diameter of
- M_x is the molar mass of molecule x
- $A_{\mathbb{D}} \approx 1.859 \times 10^{-3} \frac{\mathrm{atm}\cdot\text{Å}^2\cdot\mathrm{cm}^2}{\mathrm{K}^{3/2}\cdot\mathrm{s}\sqrt{\frac{\mathrm{g}}{\mathrm{mol}}}}$ (with Boltzmann constant k_b and Avogadro constant N_A)
- Ω is a temperature-dependent collision integral (dimensionless).

In this equation appears $\Omega(T^*)$, which depends on reduced temperature T^*:

$$T^* = \frac{T}{(\epsilon/k_B)_{AB}} \quad ; \quad \left(\frac{\epsilon}{k_B}\right)_{AB} = \sqrt{\left(\frac{\epsilon}{k_B}\right)_A \cdot \left(\frac{\epsilon}{k_B}\right)_B} \tag{4.26}$$

To find the collision integral an approximation by Neufeld et al. is used [34]:

$$\Omega(T^*) = \frac{A}{(T^*)^B} + \frac{C}{\exp(DT^*)} + \frac{E}{\exp(FT^*)} + \frac{G}{\exp(HT^*)} \tag{4.27}$$

In which:

$$\begin{aligned} A &= 1.06036, & B &= 0.15610, \\ C &= 0.19300, & D &= 0.47635, \\ E &= 1.03587, & F &= 1.52996, \\ G &= 1.76474, & H &= 3.89411 \end{aligned}$$

Again, for easy of computation an approximation was used where every specie's diffusivity is computed as if it diffused through pure nitrogen.

4.3. Convection

Convection is transport, of either some specie or of heat, caused by a displacement of the medium in which the transported is contained. Therefore, to discus what are the changes to our quantities of interest we must first know what is the movement of gas within our system, which is not at all a trivial question. One might suspect, because pressure in each node is known, that engineering approximations like the Weymouth or the Panhandle equation could tell us what will be flow rates through the

edges—but these engineering equations work for pipes, and the edges of our model do not have to be pipes.

The transport problem is so thoroughly difficult that in the analysis of chemical processes as reactor networks it is common practice to have two simulations. In these analyses, a first simulation of the fluid dynamics of the process solves the Navier-Stokes equations numerically, providing an idea of the gas flows in the system. A second simulation, our graph-based chemical analysis, takes the flow rates through the edges as a given input for its computations. The first simulation and the translation process which obtains, from a continuous description of flow, the set of volumetric flow rates for all edges $\{Q_l : e_l \in E\}$ will not be discussed. We will simply assume $Q_{kk'}$: the volume of gas flowing to node k from node k', to always be known *a priori*.

4.3.1. Convective Transport of Heat

Because temperature is a global variable, there is some translating to do before a volumetric flow can tell us how much T will change. The corresponding intrinsic variable is enthalpy H. It is intrinsic because it has units of energy per volume: [J/m^3]. The amount of heat moved into node k will be

$$\frac{dQ_k}{dt} = \sum_{k':(k',k)\in E} Q_{kk'} H_{k'} \tag{4.28}$$

so that, similarly to convection's computation in section 4.2.3, the temperature change becomes

$$\frac{dT_k}{dt} = \frac{1}{c_{P,k} V_k} \sum_{k':(k',k)\in E} Q_{kk'} H_{k'} \tag{4.29}$$

were it should be noted that depending on the flow direction $Q_{kk'}$ can be negative or positive. If gas flows into node k and out of node k' then $Q_{kk'} > 0$ and vice versa, and always $Q_{ab} = -Q_{ba}$.

4.3.2. Convective Transport of Species

The change in concentration of specie i in reactor k because of convection is a very similar expression, as we will see. The amount of particles of specie i entering node k is the volume being injected times the concentration of specie i in this injecting flow, which is the concentration at the origin of the flow. Summing over all edges, we arrive at

$$\frac{dn_k^i}{dt} = \sum_{k':(k',k)\in E} Q_{kk'} c_{k'}^i \tag{4.30}$$

Dividing by the volume of k we arrive at the differential equation for change in concentration because of convection

$$\frac{dc_k^i}{dt} = \frac{1}{V_k} \sum_{k':(k',k)\in E} Q_{kk'} c_{k'}^i \tag{4.31}$$

4.4. Incorporation into a Network Model

The full reactor network dynamics combine reaction, diffusion, conduction, and convection. All is summarised in the equation below:

$$\frac{d\vec{X}_k}{dt} = \underbrace{F_{\text{reaction}}(\vec{X}_k)}_{\text{§3}} + \underbrace{\sum_{k':(k',k)\in E} F_{\text{conduction}}(\vec{X}_k, \vec{X}_{k'})}_{\text{§4.2.3}} + \underbrace{\sum_{k':(k',k)\in E} F_{\text{diffusion}}(\vec{X}_k, \vec{X}_{k'})}_{\text{§4.2.4}} + \underbrace{\sum_{k':(k',k)\in E} F_{\text{convection}}(\vec{X}_{k'})}_{\text{§4.3}}$$

4.5. Simulations of the 12-Node Network

Because of the work of Bowen Zu [52], and Jochem van der Meer [30], obtaining code to simulate a network was trivial: the new thermodynamic functions could simply be applied to previously existing code, with but little modification required. Sections from the program that resulted in the figures of this section are included in appendix F. The simulations all describe closed systems without any convection, but these can easily be added to the model if desired. As a first proof-of-concept a 12-node graph was created, which is visualised in figure 4.2. In this graph, all nodes were attributed an equal volume of $1\mathrm{m}^3$.

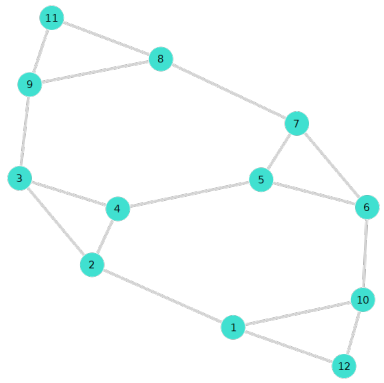

Figure 4.2: The graph used in the 12-node system simulation. All edges have weight 1. Earlier prototypes were discarded because single 'twigs' on a graph; vertices connected by just one edge, these can generate numerical instabilities. Thus a graph was made that has none, the 12-node graph.

This was the point were optimisation of the code and smart solver methods became a necessity. All of the data-types in the programs had to be reworked to become compatible with automatic differentiation methods, which largely succeeded: using the intelligent auto-differentiation of Enzyme.jl [32], does not prompt errors and the author apologizes that this is all he can say about it. This auto-differentiation does allow for the determination of a Jacobian which makes enables many implicit integration methods. Implicit methods such as the Explicit first stage Singly DIagonally implicit Runge-Kutta methods (ESDIRKs) of which Kristensen et al. [25], recommend the use of higher order variants for chemical kinetics ODEs where error tolerances are strict, which is our case because the solvers tend to crash on higher tolerance settings. Multistep methods are typically seen as the go-to powerhouse integrators for stiff problems but discontinuities reset the step sizes to small which makes the predictive magic of the multistep just very computationally expensive. Although they are robust, both me, Kristensen et al. and Rackauckas and Nie—the last of which get their knowledge from of a Julia-specific benchmarking investigation [37]—find that single step methods are better for combustion. Multistep methods did find an application when smooth solutions were expected in the diffusion-lead end-phase of a skeletal mechanism simulation, of which the results are seen in figure 4.3. The methods developed by Kennedy & Carpenter [24], proved especially effective.

To create an ODE that would truly test the capabilities of the solver methods, the a 12-node setup using Laurent's mechanism [26], was given an initial state of $1200\mathrm{K}$ and $10\mathrm{bar}$ in all nodes, with only air present everywhere except for nodes 4, 7, 9 and 12 which contained an stoichiometric air-methane mixture and nodes 1, 3 and 5 which contained an stoichiometric hydrogen-air mixture. This setup created such a stiff problem because the two explosions happen at wildly different timescales, and diffusion proceeds at a different timescale still. The results are shown in figure 4.3.

That this one simulation requires a range of images to display is because of the way the stiffness was overcome. All algorithms are on the spectrum from robust to efficient, and therefore the choice was made to divide the simulation timespan up into parts so a switch of algorithm could be made in the middle of the simulation. From experiments with the single node skeletal modal some ideas were obtained on when the explosions where expected to happen: in figure 3.5 we can see that methane ignites within 0.5 milliseconds and hydrogen within 5 microseconds, so naturally this gives us periods for which the necessity of a slow-but-steady, stiffness capable solver was known. The first part of the integration was done with an automatically method switching algorithm, with an A-L stable stiffly-accurate 4th order seven-stage ESDIRK method with splitting ('KenCarp47'), and Verner's "Most Efficient" 7/6 Runge-Kutta method ('Vern7') as the choices for the automatic switch. After the hydrogen had burned a new strategy was applied: the A-L stable stiffly-accurate 3rd order ESDIRK method with splitting, known as 'KenCarp3' was used in combination with again 'Vern7'. After the methane had burned too and no more discontinuities in the solution were expected, then a multistep method became appropriate. A fixed-leading coefficient adaptive-order adaptive-time BDF method, known as 'FBDF', ran the simulation into the diffusion ruled domain.

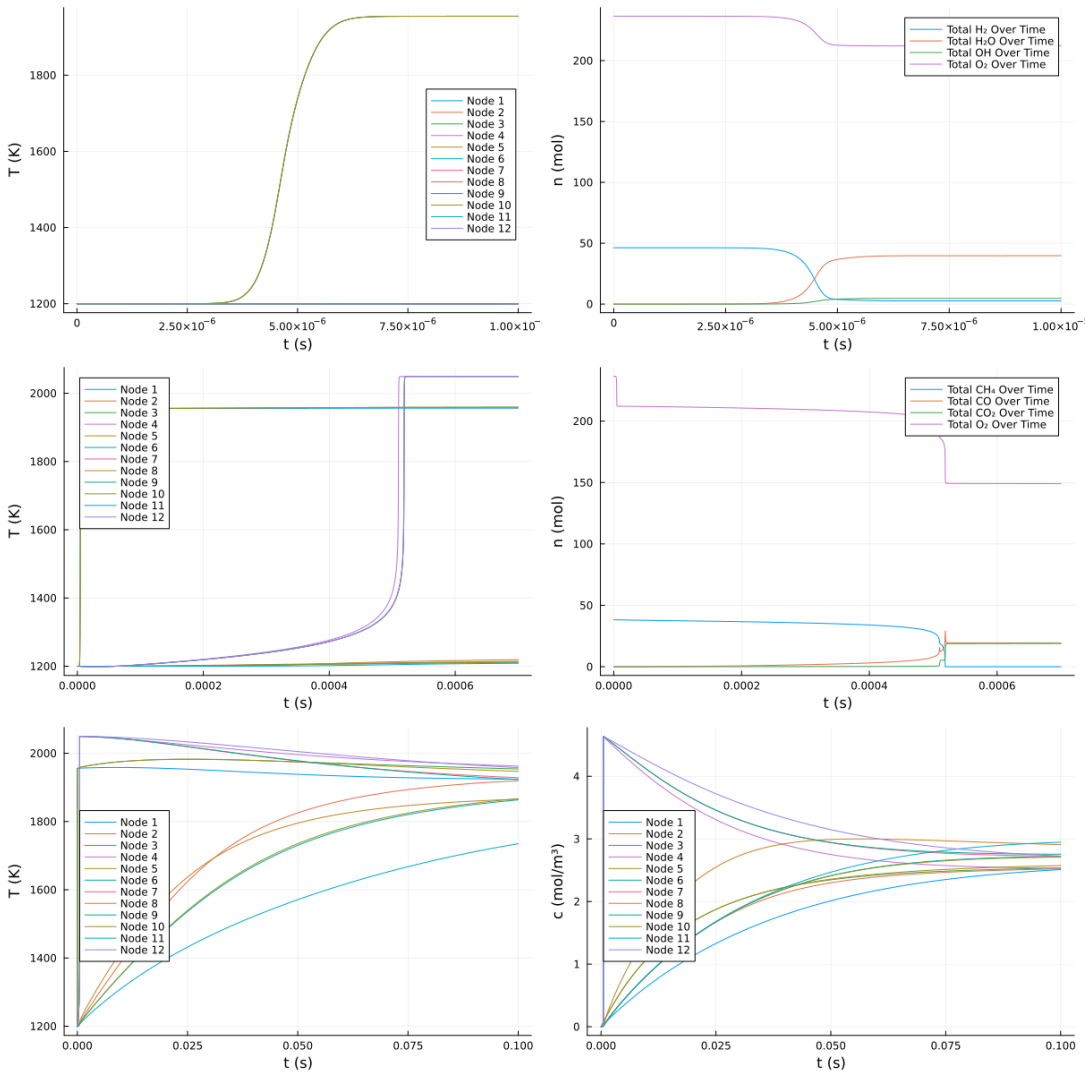

Figure 4.3: The combustion of methane and hydrogen in a 12 node reactor system using Laurent's reaction mechanism [26].

5

Results & Discussion

5.1. Performance & Accuracy

To analyse the performance, some data was obtained which is collected in table 5.1. These are the models that have been described all throughout this thesis. All their initial conditions were $1200\mathrm{K}$ and $1\mathrm{bar}$. Algorithm choices have been named in section 3.6.3 and 4.5, but to re-emphasize: explicit first stage singly diagonally implicit Runge-Kutta (ESDIRK) methods have been used primarily, and especially so the Kennedy-Carpenter algorithms.

Two values give information about performance. Execution time is the time it took to integrate the problem, which varies a lot in Julia depending on if the program is ran for the first time; the initial run takes longer because everything still has to be compiled, so a second run was used to collect data. Memory usage is the amount of information that had to be stored in order to integrate the problem, and it is a good indication of the efficiency of the code. When too much memory is used because the code has excessive allocations, then memory can become a bottleneck for performance.

Another two values give information about correctness of the model, or at least, they give us the possibility to compare models. Peak temperature is the hottest temperature that occurred in the simulation. Relaxation time has a more complex definition, which requires us to think of the initial condition as a perturbed state and the final solution value as the equilibrium: for the purpose of this research the relaxation time τ is defined as how long it takes for the perturbation from equilibrium values ($\Delta U(T) = ||U_f - U(T)||_2$) to decrease by $1/e$. See appendix G for the Julia code that computed τ. Because too many processes happened at once in the spatial models there is no defnite relaxation time for these integrations.

Table 5.1: Performance metrics of the different combustion simulation algorithms.

Problem	Execution Time (s)	Memory Usage (MB)	Peak Temp. (K)	Relaxation Time (s)
1-Step Constant T	8.549×10^{-2}	2.872×10^{-2}	-	2.335
1-Step + Thermo	4.893	372.1	3044	7.188×10^{-2}
4-Step Mechanism	53.79	21.67×10^{3}	3049	1.047×10^{-6}
Skeletal: 2-Step	6.651×10^{-3}	21.43	2784	8.168×10^{-5}
Skeletal: Laurent	0.1440	33.63	2019	5.272×10^{-4}
Skeletal: Lu	1.233	137.9	2019	4.485×10^{-4}
Skeletal: GRI-Mech	15.49	1.293×10^{3}	2019	4.484×10^{-4}
12-Node: 2-step	22.60	8.559×10^{3}	2782	-
12-Node: Laurent	17.11	6.917×10^{3}	2019	-

While there are considerably more variables in Laurent's 12-node model than in the 2-step variant, the runtime was better because more effort and attention was spent on optimising this run, which is the run from figure 4.3.

5.1.1. Reflections on correctness

One conclusion is that the sophisticated zero dimensional model combustions take place over timescales which are in excellent agreement with literature values [21, 7], which predict methane combustion completing in tenths of milliseconds at the high T, high P conditions of the simulations. It seems that mechanisms of too few steps appear not to be able to model the speeds of reactions realistically, as the 2-step mechanism is too slow and the 4-step mechanism far too fast. That GRI-Mech is reliable, this much is clear.

A second conclusion is that the peak temperature values are somewhat close to the adiabatic flame temperature found in literature [45], which is $2235\mathrm{K}$. The models that cannot simulate incomplete combustion have temperatures which are high, and the sophisticated models all seem to be missing the same $200\mathrm{K}$: as was discussed at the end of chapter 3 this is because of excessive dissociative activity. The energy is there, it is just present as the chemical energy of a broken bond, not as heat.

From the table one can clearly read that the 4-step mechanism was very bad. Perhaps it is just unwise to mix and match theories without any greater oversight or understanding of what is going on. GRI-Mech 3.0 is the product of hundreds of carefully selected theories and experiments that together provided the constraints to an optimisation problem of which that dataset is the result. When one is being less careful in selecting a set of Arrhenius coefficients for a reaction mechanism, then the risk is created that the mechanism is inaccurate or even unintegrable.

5.1.2. Discussion on Optimization

What can also be seen in the table is that runtimes are acceptable, but not ideal. The author is a physicist, not a computer scientist, and it shows. The main focus was a correct model and once this model was created, it proved rather inefficient primarily because of suboptimal allocation efficiency. An allocation is the creation of a new variable in a computer program, which demands writing into memory, and a lot of allocations make for slow programs. It is better to store variables in cache for later reuse. Especially the Chemistry.jl file is riddled with allocations and suboptimal parts—it has so far only been rewritten twice, once to enable automatic differentiation and once to go from isobaric to isochoric thermodynamics. Another unexplored option for faster integrations is parallel processing.

A major note on execution times is that all of them are preceded by a compilation time that is often $> 90\%$ of the complete time spent waiting for the result. In practice not much will be recompiled when iterating over initial conditions or some other detail. In fact, as long as the reaction mechanism is not swapped, later runs will practically always be faster than the initial round. Iterating for design purposes can be done at acceptable runtimes (e.g. tens of seconds per iteration). However, the more which is modified and overwritten, the more recompilation is required, and there can be times where inefficient code will still take many, many minutes to complete.

5.2. Future directions

There are several things that could have ameliorated the program developed in this thesis significantly. In this section will be discussed the unresolved problems.

5.2.1. Analytically Deriving the Jacobian

Most worthwhile solvers require (an educated guess of) the Jacobian to run. The complete ODE description is rather large, however, it is everywhere explicitly defined and differentiable, so determining the full Jacobian is a possibility. Code in appendix F currently only describes a method for finding the sparsity structure of the Jacobian, but this does affirm the possibility of full Jacobian determination. The analytically derived Jacobian would allow for the abandonment of automatic differentiation, skipping their approximations while directly receiving a more correct alternative.

One should be careful however, if this Jacobian data-object becomes to taxing on memory it loses its purpose as an efficiency enhancer.

5.2.2. The Clamp

Negative concentrations or temperatures are unphysical and the equations of Arrhenius and NASA7 break down when faced with them. A callback that detects zero crossings over time steps was imple-

mented but insufficient for intra-step evaluations of sophisticated implicit solver algorithms. Numeric stability required a clamp that kept concentrations not just at zero but slightly above, positive, at an arbitrarily small number. This nozero concentration everywhere of everything ought to create small numeric artifacts that possibly play a role in the dissociation by providing trace amounts of whatever molecule or free radical is needed.

5.2.3. Explaining the Dissociation

The author is at a loss in regards to why the molecules in his simulation fall apart so easily, but they do and this results in temperature predictions that deviate from the literature. This is a problem that needs to be understood and resolved before the code developed in this thesis can have any real world application.

5.2.4. Validating the Transport Phenomenology

There is a lot of talk in chapter 4, and most of it needs to be either understood or taken on faith. Both are not ideal. The validity of graph models can only be affirmed when there are clear real world processes that the graphs then become a model of, if then the graph correctly describes the real world, then the validity is confirmed. It remains up for debate of how much importance graph based diffusion simulation is because in practically any relevant setting it will be convective transport which dominates.

6

Conclusion

Julia has been found to be well suited for combustion modelling, it has methods for severely stiff problems such as those posed by thermochemistry. Complex problems of more than 200 variables were solved within minutes of runtime—within seconds even, after compilation. This was achieved using code that was poorly optimised, reaffirming Julia's potential. A model was developed, and while it still lacks validations and has some flaws, it approximates the kinetics and thermochemistry of combustion well and fast.

Through the development of this tech-demo for combustion analysis the suitability of Julia as a language for complex scientific programming with heavy computational demands has been affirmed.

References

[1] Ian Alexander and Krishnavedala. *Lotka-Volterra Model (1.1, 0.4, 0.4, 0.1)*. https://commons.wikimedia.org/wiki/File:Lotka-Volterra_model_(1.1,_0.4,_0.4,_0.1).svg. Parameters by Ian Alexander; vectorization by Krishnavedala. SVG created with Matplotlib. Dec. 2018.

[2] Svante Arrhenius. "Über die Dissociationswärme und den Einfluss der Temperatur auf den Dissociationsgrad der Elektrolyte". In: *Zeitschrift für physikalische Chemie* 4.1 (1889), pp. 96–116.

[3] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. "Yaml ain't markup language (yaml™) version 1.1". In: *Working Draft 2008* 5.11 (2009).

[4] Michele Benzi and Igor Simunec. "Rational Krylov methods for fractional diffusion problems on graphs". In: *BIT Numerical Mathematics* 62.2 (2022), pp. 357–385.

[5] Jeff Bezanson et al. "Julia: A fresh approach to numerical computing". In: *SIAM review* 59.1 (2017), pp. 65–98.

[6] Max Bodenstein. "Eine theorie der photochemischen reaktionsgeschwindigkeiten". In: *Zeitschrift für physikalische Chemie* 85.1 (1913), pp. 329–397.

[7] A Burcat et al. "Shock-tube investigation of ignition in methane-oxygen-argon mixtures(Methane-oxygen-argon mixture ignition behind reflected shock waves in single pulse shock tube)". In: *Combustion and Flame* 16 (1971), pp. 311–321.

[8] Sadi Carnot. "Reflections on the motive power of fire, and on machines fitted to develop that power". In: *Paris: Bachelier* 108.1824 (1824), p. 1824.

[9] Sydney Chapman. "The kinetic theory of simple and composite monatomic gases: viscosity, thermal conduction, and diffusion". In: *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character* 93.646 (1916), pp. 1–20.

[10] Henry J Curran et al. "A comprehensive modeling study of iso-octane oxidation". In: *Combustion and flame* 129.3 (2002), pp. 253–280.

[11] R.S. Devoto. "Thermal conductivity of multicomponent gas mixtures". In: *Physica* 45.4 (1970), pp. 500–505. ISSN: 0031-8914. DOI: https://doi.org/10.1016/0031-8914(70)90062-5. URL: https://www.sciencedirect.com/science/article/pii/0031891470900625.

[12] David Enskog. "Kinetische theorie der Vorgänge in mässig verdünnten Gasen. I. Allgemeiner Teil". In: *Uppsala: Almquist & Wiksells Boktryckeri* (1917).

[13] A Fick. "Über Diffusion, Z. rat". In: *Medicin, NF Bd. VI* (1855), pp. 288–301.

[14] Jean Baptiste Joseph baron de Fourier. *Théorie analytique de la chaleur*. Firmin Didot, 1822.

[15] B. Franzelli et al. "A two-step chemical scheme for kerosene–air premixed flames". In: *Combustion and Flame* 157.7 (2010), pp. 1364–1373. ISSN: 0010-2180. DOI: https://doi.org/10.1016/j.combustflame.2010.03.014. URL: https://www.sciencedirect.com/science/article/pii/S0010218010001021.

[16] Josiah Willard Gibbs. "On the equilibrium of heterogeneous substances". In: *American journal of science* 3.96 (1878), pp. 441–458.

[17] David G Goodwin et al. "Cantera: An object-oriented software toolkit for chemical kinetics, thermodynamics, and transport processes". In: *Zenodo* (2018).

[18] Sanford Gordon and Bonnie J McBride. *Computer Program for Calculation of Complex Chemical Equilibrium Compositions, Rocket Performance, Incident and Reflected Shocks, and Chapman-Jouguet Detonations. Interim Revision, March 1976*. Tech. rep. NASA, 1976.

[19] Shin-itiro Goto and Hideitsu Hino. "Diffusion equations from master equations—A discrete geometric approach". In: *Journal of Mathematical Physics* 61.11 (2020).

[20] Wendell M Graven and F John Long. “Kinetics and mechanisms of the two opposing reactions of the equilibrium CO+ H2O= CO2+ H21”. In: *Journal of the American Chemical Society* 76.10 (1954), pp. 2602–2607.

[21] KA Heufer and HJSW Olivier. “Determination of ignition delay times of different hydrocarbons in a new high pressure shock tube”. In: *Shock Waves* 20.4 (2010), pp. 307–316.

[22] WP Jones and RP Lindstedt. “Global reaction schemes for hydrocarbon combustion”. In: *Combustion and flame* 73.3 (1988), pp. 233–249.

[23] Robert J Kee, Fran M Rupley, and James A Miller. *The Chemkin thermodynamic data base*. Tech. rep. Sandia National Lab.(SNL-CA), Livermore, CA (United States), 1990.

[24] Christopher A Kennedy and Mark H Carpenter. “Additive Runge–Kutta schemes for convection–diffusion–reaction equations”. In: *Applied numerical mathematics* 44.1-2 (2003), pp. 139–181.

[25] Morten R Kristensen et al. “Efficient integration of stiff kinetics with phase change detection for reactive reservoir processes”. In: *Transport in porous media* 69.3 (2007), pp. 383–409.

[26] C. Laurent. “Low-Order Modeling and High-Fidelity Simulations for the Prediction of Combustion Instabilities in Liquid-Rocket Engines and Gas Turbines”. Ecole doctorale MeGE. PhD thesis. Institut National Polytechnique de Toulouse, 2020.

[27] Alfred J Lotka. “Analytical note on certain rhythmic relations in organic systems”. In: *Proceedings of the National Academy of Sciences* 6.7 (1920), pp. 410–415.

[28] Tianfeng Lu and Chung K. Law. “A criterion based on computational singular perturbation for the identification of quasi steady state species: A reduced mechanism for methane oxidation with NO chemistry”. In: *Combustion and Flame* 154.4 (2008), pp. 761–774. ISSN: 0010-2180. DOI: https://doi.org/10.1016/j.combustflame.2008.04.025. URL: https://www.sciencedirect.com/science/article/pii/S0010218008002009.

[29] Osama A. Marzouk. “Adiabatic Flame Temperatures for Oxy-Methane, Oxy-Hydrogen, Air-Methane, and Air-Hydrogen Stoichiometric Combustion using the NASA CEARUN Tool, GRI-Mech 3.0 Reaction Mechanism, and Cantera Python Package”. In: *Engineering, Technology & Applied Science Research* 13.4 (Aug. 2023), pp. 11437–11444. ISSN: 2241-4487. DOI: 10.48084/etasr.6132. URL: http://dx.doi.org/10.48084/etasr.6132.

[30] Jochem Jelle van der Meer. *Time Integration of the Chemistry of Combustion Processes in Industrial Furnaces, using Julia*. Bachelor’s thesis. Faculty of Electrical Engineering, Mathematics and Computer Science, Delft Institute of Applied Mathematics. Delft, Netherlands, 2023.

[31] Manjiri Moharir et al. “Graph representation and distributed control of diffusion-convection-reaction system networks”. In: *Chemical Engineering Science* 204 (2019), pp. 128–139.

[32] William S. Moses et al. “Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: 10.1145/3458817.3476165. URL: https://doi.org/10.1145/3458817.3476165.

[33] Codina Movileanu et al. “Adiabatic flame temperature of fuel-air mixtures in isobaric and isochoric combustion processes”. In: *Rev. Chim* 62.4 (2011), pp. 376–379.

[34] Philip D Neufeld, AR Janzen, and R_A Aziz. “Empirical equations to calculate 16 of the transport collision integrals Ω (l, s)* for the Lennard-Jones (12–6) potential”. In: *The Journal of chemical physics* 57.3 (1972), pp. 1100–1102.

[35] A Ölander. *Nobel Prize in Chemistry 1956: Award ceremony speech*. https://www.nobelprize.org/prizes/chemistry/1956/ceremony-speech/. Accessed 25-6-2025.

[36] H.-O. Pörtner et al. *Climate Change 2022: Impacts, Adaptation and Vulnerability*. Technical Summary. Cambridge, UK and New York, USA: Cambridge University Press, 2022, pp. 37–118. ISBN: 9781009325844.

[37] Christopher Rackauckas and Qing Nie. “Confederated modular differential equation APIs for accelerated algorithm development and benchmarking”. In: *Advances in Engineering Software* 132 (2019), pp. 1–6.

[38] Christopher Rackauckas and Qing Nie. “DifferentialEquations.jl–a performant and feature-rich ecosystem for solving differential equations in Julia”. In: *Journal of Open Research Software* 5.1 (2017).

[39] C Robinson and DB Smith. “The auto-ignition temperature of methane”. In: *Journal of hazardous materials* 8.3 (1984), pp. 199–203.

[40] Wil Roebroeks and Paola Villa. “On the earliest evidence for habitual use of fire in Europe”. In: *Proceedings of the National Academy of Sciences* 108.13 (2011), pp. 5209–5214.

[41] L.F. Shampine et al. “Non-negative solutions of ODEs”. In: *Applied Mathematics and Computation* 170.1 (2005), pp. 556–569. ISSN: 0096-3003. DOI: https://doi.org/10.1016/j.amc.2004.12.011. URL: https://www.sciencedirect.com/science/article/pii/S0096300304009683.

[42] G. P. Smith et al. *Optimized Kinetics Mechanism and Calculator for Natural Gas Combustion – GRI-Mech 3.0*. Presented at the International Symposium on Combustion, Maui, 1998. http://www.me.berkeley.edu/gri_mech/. 1999.

[43] G. P. Smith et al. *RATE COEFFICIENT FORMAT*. http://combustion.berkeley.edu/gri-mech/data/k_form.html. Accessed 12-7-205.

[44] tec-science. *Thermal conductivity of gases*. Accessed: 20-08-2025. 2020. URL: https://www.tec-science.com/thermodynamics/heat/thermal-conductivity-of-gases/.

[45] The Engineering ToolBox. *Adiabatic Flame Temperatures*. https://www.engineeringtoolbox.com/adiabatic-flame-temperature-d_996.html. Accessed on 14-7-2025. 2005.

[46] Pierre-François Verhulst. “Notice sur la loi que la population suit dans son accroissement”. In: *Correspondence mathematique et physique* 10 (1838), pp. 113–129.

[47] Bryan W Weber and Kyle E Niemeyer. “ChemKED: A Human-and Machine-Readable Data Standard for Chemical Kinetics Experiments”. In: *International Journal of Chemical Kinetics* 50.3 (2018), pp. 135–148.

[48] Charles K Westbrook and Frederick L Dryer. “Simplified reaction mechanisms for the oxidation of hydrocarbon fuels in flames”. In: *Combustion science and technology* 27.1-2 (1981), pp. 31–43.

[49] Wikipedia. *List of chemical process simulators*. https://en.wikipedia.org/wiki/List_of_chemical_process_simulators. Accessed: 13-4-2025.

[50] Wikipedia. *Mass diffusivity*. https://en.wikipedia.org/wiki/Mass_diffusivity. Accessed: 19-20-2025.

[51] Peng Zhang et al. “A comprehensive review of chemical mechanism reduction methods for computational combustion modeling”. In: *Combustion Science and Technology* 191.7 (2019), pp. 1138–1164. DOI: 10.1080/00102202.2018.1529036.

[52] Bowen Shuchai Zhu. *A Chemical Reactor Network Approach for Modeling Pollutant Formation in Large Industrial Furnaces in Julia*. Bachelor’s thesis. Faculty of Electrical Engineering, Mathematics and Computer Science. Delft, Netherlands, 2021.

A

1-step constant T Julia code

The structure of the Julia files is large the same across simple and complex models. It begins by loading all the needed modules and defining the problem parameters.

```
using DifferentialEquations
using Plots

#temperature (assumed constant)
T = 1200 #K
P = 1e6 #Pa

R = 8.314 # J/molK

c_tot = P / (R*T) # ideal gas law

# Initial species concentrations 1. N2, 2. O2, 3. CH4, 4. H2O, 5. CO2
X0 = c_tot*[7.57/10.57, 2/10.57, 1/10.57, 0, 0] #mol/m^3, adiabatic mixture

# 2*O2 + CH4 -> 2*H2O + CO2
S = [0, -2, -1, 2, 1]
```

Notice here that the stochiometric matrices have consistently been defined in their transposed form, which is why the equation for change in concentration which follows looks weird. After obtaining the problem parameters, the ODE is defined. It usually consists of a main function, which is falled ”f” here, and some extra helper functions, in this case only ”Arrhenius”.

```
function Arrhenius(T, X)
    # Constants from Westbrook & Dryer 1981
    n_CH4 = -0.3
    n_O2 = 1.3
    A = 1.3e8
    Ea = 48.4 * 4184 # kcal/mol to J/mol
    R = 8.314 # J/mol/K

    # Ensure positive concentrations for exponentiation
    CH4_concentration = max(X[3], 0) * 1e-6 # Arrhenius takes /cm3
    O2_concentration = max(X[2], 0) * 1e-6 # Arrhenius takes /cm3
    k = A * exp(-Ea / (R * T)) # rate constant
    r = k * CH4_concentration^n_CH4 * O2_concentration^n_O2 # reaction rate
    return r * 1e6 # Back to /m3
end
```

```
function f!(dX, X, p, t)
    r = Arrhenius(T, X)
    dX .= r .* S
end
```

Next the ODE is integrated, which looks simple but is a rather sophisticated art when problems get more complex...

```
# define time span
tend = 10
tspan = (0, tend)

# define problem
@time problem = ODEProblem(f!, X0, tspan)

# solve problem
sol = solve(problem, alg_hints=[:stiff])
```

Lastly the solution is plotted, and this part of the code will be omitted from the rest of the sample code appendices.

```
# plot solution obtained
plot1 = plot(sol,
    xaxis="time [s]",
    yaxis="concentration [mol/m^3]",
    label=["N2" "O2" "CH4" "H2O" "CO2"]
)

plot(plot1, legend=true)
display(plot1)
```

B

Using the CHEMKIN database

I wrote Chemkin.jl in the hope that it would be my definitive program for thermodynamic computations, but once I started researching reaction mechanism databases I discovered that the problem required more sophistication which resulted in chemistry.jl (which is not included in the appendices because it is too long). How to use the CHEMKIN database is illustrative of what chemistry.jl does with .YAML files at large and that is why it is shown here. It all begins with the database, which is a text file with a structure like:

```
THERMO
   300.000  1000.000  5000.000
(CH2O)3            70590C   3H   6O   3     G  0300.00   4000.00  1500.00      1
 0.01913678E+03 0.08578044E-01-0.08882060E-05-0.03574819E-08 0.06605142E-12    2
-0.06560876E+06-0.08432507E+03-0.04662286E+02 0.06091547E+00-0.04710536E-03    3
 0.01968843E-06-0.03563271E-10-0.05665403E+06 0.04525264E+03                   4
(CH3)2SICH2        61991H   8C   3SI  1     G  0300.00   2500.00  1500.00      1
 0.15478518E+02 0.10657002E-01-0.12343446E-06-0.12933515E-08 0.02528714E-11    2
-0.06693076E+04-0.05358884E+03 0.02027522E+02 0.04408673E+00-0.03370023E-03    3
 0.14844662E-07-0.02830898E-10 0.03931453E+05 0.01815820E+03                   4
AL                 62987AL  1               G  0300.00   5000.00  0600.00      1
 0.02559589E+02-0.10632239E-03 0.07202828E-06-0.02121105E-09 0.02289429E-13    2
 0.03890214E+06 0.05234522E+02 0.02736825E+02-0.05912374E-02-0.04033937E-05    3
 0.02322343E-07-0.01705599E-10 0.03886794E+06 0.04363879E+02                   4
...
```

To read this database it was necessary to create a function that parses the text and creates data objects that are useable:

```
function load_chemical_data(filename)
    data = Dict{String, Tuple{Float64, Vector{Float64}}}()
    current_name = ""
    current_temp = 0.0
    coefficients = Float64[]

    lines = readlines(filename)

    for line in lines
        line_stripped = strip(line)
        if line_stripped == ""  # Skip empty lines
            continue
        end
```

```
        number_pattern = r"[-+]?(\d+\.\d*|\.\d+)([eE][-+]?\d+)?|[-+]?\d+"
        matches = collect(eachmatch(number_pattern, line_stripped))
        if length(matches) > 0
            nums = [parse(Float64, match.match) for match in matches]

            # Determine the type of line based on the last character
            last_number = endswith(line_stripped, "1") ? 1 :
                          endswith(line_stripped, "2") ? 2 :
                          endswith(line_stripped, "3") ? 3 :
                          endswith(line_stripped, "4") ? 4 : nothing

            if last_number == 1
                # Split the line into parts based on whitespace
                parts = split(line_stripped)
                # The first part is the chemical name
                if length(parts) > 0
                    current_name = parts[1]
                    # The temperature is the penultimate part before '1'
                    temp_str = parts[end - 1]
                    current_temp = parse(Float64, temp_str)
                    coefficients = Float64[]
                end
            elseif last_number  (2, 3, 4)
                # These lines contain coefficients
                append!(coefficients, nums[1:end-1])
                if last_number == 4
                    data[current_name] = (current_temp, coefficients)
                    current_name = ""
                    current_temp = 0.0
                    coefficients = Float64[]
                end
            end
        end
    end
    return data
end
```

Then, finally, thermodynamic properties of species at given temperatures can be computed, using the NASA polynomials. As you can see, I was rather confused by the double standard of units (calories for activation energies in reactions VS joules in the thermodynamic calculations), which is why I multiplied dimensionless numbers firstly by the gas constant in calories and then by a conversion factor:

```
function species_cp(T, data_for_a_species)
    R_cal = 1.98720425864083
    cal_to_J = 4.184

    T_common, coeffs = data_for_a_species
    # Select the appropriate set of coefficients based on T
    if T >= T_common
        a = coeffs[1:7]    # High-temperature coefficients: a1 to a7
    else
        a = coeffs[8:14]   # Low-temperature coefficients: a8 to a14
    end
    # NASA polynomial for Cp/R
    Cp_R = a[1] + a[2]*T + a[3]*T^2 + a[4]*T^3 + a[5]*T^4
    # Convert to Cp in J/(mol·K)
    Cp_cal = Cp_R * R_cal
```

```
    Cp = Cp_cal * cal_to_J
    return Cp # J/mol/K
end

function h0(T, data_for_a_species)
    R_cal = 1.98720425864083
    cal_to_J = 4.184

    T_common, coeffs = data_for_a_species
    # Select the appropriate set of coefficients based on T
    if T >= T_common
        a = coeffs[1:7]    # High-temperature coefficients: a1 to a7
    else
        a = coeffs[8:14]   # Low-temperature coefficients: a8 to a14
    end
    # NASA polynomial for (H/RT)
    H_RT = a[1] + a[2]*T/2 + a[3]*(T^2)/3 + a[4]*(T^3)/4 + a[5]*(T^4)/5 + a[6]/T
    # Convert to H in cal/mole
    H_cal = H_RT * R_cal * T
    H = H_cal * cal_to_J
    return H # J/mol
end
```

C

1-step thermodynamic Julia code

The 1-step thermodynamic program combines the earlier chemical kinetics with the thermodynamics from CHEMKIN.

```
using DifferentialEquations
using Plots
include("Chemkin.jl")

species_names = ["N2", "O2", "CH4", "H2O", "CO2"]
n_s = 5

#temperature
T = 1200 #K
P = 10e5 #Pa

R = 8.314 # J/molK

c_tot = P / (R*T) # ideal gas law

X0 = zeros(length(species_names)+1)
X0[1] = T # K
X0[2:end] = c_tot*[7.57/10.57, 2/10.57, 1/10.57, 0, 0] #mol/m^3, adiabatic mixture

# Stoichiometry: 2 O2 + CH4 -> 2 H2O + CO2
S = [0, -2, -1, 2, 1]  # N2 does not participate in the reaction
n_r = 1

species_dict = load_chemical_data("CHEMKIN-THERMDAT.txt")

function Arrhenius(Y)
    T = Y[1]
    X = Y[2:end] * 1e-6 # Arrhenius takes /cm3
    # Constants from Westbrook & Dryer 1981
    n_CH4 = -0.3
    n_O2 = 1.3
    A = 1.3e8
    Ea = 48.4e3 # cal/mol
    R = 1.987 # cal/mol/K

    # Ensure positive concentrations for exponentiation
```

```
    CH4_concentration = max(X[3], 0)
    O2_concentration = max(X[2], 0)

    k = A * exp(-Ea / (R * T)) # rate constant
    r = k * CH4_concentration^n_CH4 * O2_concentration^n_O2 # reaction rate
    return r * 1e6 # Back to /m3
end

# Temperature rate of change due to reaction enthalpy
function dT(X, r)
    h0_vec = []
    cp_vec = []
    dH = sum( S' * [h0(X[1],species_dict[species_names[i]]) for i in 1:n_s]) * r
    c_p = sum(X[2:end] .* [species_cp(X[1],species_dict[species_names[i]]) for i in 1:n_s])
    dT = -dH/c_p #J/mols * molK/J
    return dT[1]
end

# Derivative function for the ODE system
function f!(dX, X, p, t)
    r = Arrhenius(X)
    dX[1] = dT(X,r)
    dX[2:end] .= r .* S  # Species concentrations change
end

# define
tend = 5e-8
tspan = (0, tend)

# define problem
problem = ODEProblem(f!, X0, tspan)

# solve problem
sol = solve(problem, alg_hints=[:stiff])
```

D

4-step thermodynamic Julia code

The 4-step model expands on the earlier models.

```
using DifferentialEquations
using Plots
using LinearAlgebra
using Plots
include("Chemkin.jl")

#temperature (assumed constant)
T = 1200 #K
P = 10e5 #Pa

R = 8.314 # J/molK

c_tot = P / (R*T) # ideal gas law

# Initial species concentrations: N2, O2, CH4, H2O, CO2, CO, H2
species_names = ["N2", "O2", "CH4", "H2O", "CO2", "CO", "H2"]
n_s = 7

X0 = zeros(length(species_names)+1)
X0[1] = T # K
X0[2:end] = c_tot*[0.7148, 0.1901, 0.0951, 0, 0, 0, 0] # mol/m^3, 75% air 25% methane mixture

# Atomic accounting check
println("The amount of hydrogen atoms is "*string(X0[4]*4+X0[5]*2+X0[8]*2)*" moles")
println("The amount of carbon atoms is "*string(X0[4]+X0[6]+X0[7])*" moles")
println("The amount of oxygen atoms is "*string(X0[3]*2+X0[5]+X0[6]*2+X0[7])*" moles")
```

It is perhaps interesting to include the output of this atomic accounting check, so we may verify at the end that atoms were preserved. The print statement was:

The amount of hydrogen atoms is 38.12845802261247 moles
The amount of carbon atoms is 9.532114505653118 moles
The amount of oxygen atoms is 38.10841151471414 moles

```
# Stoichiometric matrix
S = [0 -0.5 -1 0 0 1 2;
     0 0 -1 -1 0 1 3;
     0 -0.5 0 1 0 0 -1;
```

```
     0 0.5 0 -1 0 0 1;
     0 0 0 -1 1 -1 1;
     0 0 0 1 -1 1 -1]

n_r = 6

species_dict = load_chemical_data("CHEMKIN-THERMDAT.txt")

# Reaction rate function
function Arrhenius(X)
    R = 1.987204258640

    T = X[1]
    Y = X[2:end]

    # Ensure non-negative concentrations
    Y = max.(Y, 1e-10)  # No zero or negative concentrations to prevent errors

    Y1 = Y * 1e-3 # Jones & Lindstedt use kmol and m
    Y2 = Y * 1e-3 # Graven & Long use mol and liters

    k  = zeros(6)
    k[1] = 0.44e12 * exp(-30000 / (R * T)) #from Jones & Lindstedt 1988
    k[2] = 0.30e9 * exp(-30000 / (R * T)) #from Jones & Lindstedt 1988
    k[3] = 0.25e17 * T^-1 * exp(-40000 / (R * T)) #from Jones & Lindstedt 1988
    k[4] = 0
    k[5] = 5.0e12 * exp(-67300/ (R * T)) #from Graven & Long 1953
    k[6] = 9.5e10 * exp(-57000/ (R * T)) #from Graven & Long 1953


    # Reaction rates
    r = zeros(6)
    r[1] = k[1] * Y1[3]^0.5 * Y1[2]^1.25 #from Jones & Lindstedt 1988
    r[2] = k[2] * Y1[3] * Y1[4] #from Jones & Lindstedt 1988
    r[3] = k[3] * Y1[7]^0.5 * Y1[2]^2.25 * Y1[4]^-1 #from Jones & Lindstedt 1988
    r[4] = 0
    r[5] = k[5] * Y2[6]^0.5 * Y2[4] / (1 + 1.2e4 * Y2[7]) #from Graven & Long 1953
    r[6] = k[6] * Y2[7]^0.5 * Y2[5] / (1 + 3.6e3 * Y2[6]) #from Graven & Long 1953

    r[1:3]=r[1:3] * 1e3 # kmol to mol
    r[3:6]=r[3:6] * 1e3 # /l to /m3

    return r
end

# Temperature rate of change due to reaction enthalpy
function dT(X, r)
    h0_vec = []
    cp_vec = []
    dH = sum([h0(X[1],species_dict[species_names[i]]) for i in 1:n_s]' * S' * r )
    c_p = sum(X[2:end] .* [species_cp(X[1],species_dict[species_names[i]]) for i in 1:n_s])
    dT = -dH/c_p #J/mols * molK/J
    return dT[1]
end

# Derivative function for the ODE system
```

```
function f!(dX, X, p, t)
    X = max.(X, 1e-10)
    r = Arrhenius(X)
    dX[1] = dT(X,r)
    dX[2:end] = S' * r  # Species concentrations change
end

# Define time span and solve the ODE problem
tend = 0.01
tspan = (0.0, tend)

# Define and solve the ODE problem with tighter tolerances
problem = ODEProblem(f!, X0, tspan)
@time sol = solve(problem, alg_hints=[:stiff], abstol = 1e-6, reltol = 1e-4)

# Atomic accounting check
X0 = sol[end]
println("The amount of hydrogen atoms is "*string(X0[4]*4+X0[5]*2+X0[8]*2)*" moles")
println("The amount of carbon atoms is "*string(X0[4]+X0[6]+X0[7])*" moles")
println("The amount of oxygen atoms is "*string(X0[3]*2+X0[5]+X0[6]*2+X0[7])*" moles")
```

The final atomic accounting statement read:

The amount of hydrogen atoms is 38.12845802261251 moles
The amount of carbon atoms is 9.532114505653075 moles
The amount of oxygen atoms is 38.108411514714106 moles

E

Skeletal model Julia code

The skeletal model is at its heart a module I made called Chemistry.jl, which you can imagine as a far more complete version of Chemkin.jl suited for .YAML files and containing many helpful functions for chemical kinetics and thermochemistry. It was used in this script.

```
# Import required packages
using DifferentialEquations     # For solving differential equations
using LinearAlgebra             # Provides linear algebra functionalities
using SparseArrays              # For efficient storage of sparse matrices
#using KLU                      # Sparse LU factorization solver
using IterativeSolvers          # Iterative algorithms for linear systems
using SparseDiffTools           # Tools for differentiating sparse functions
using Plots                     # For plotting results
using YAML                      # For parsing YAML files

include("Chemistry.jl") # Chemistry.jl is the actual program

# Set up the ODE problem
const FILENAME = "2step.yaml"

# Load the mechanism file
mechanism_data = load_mechanism_data(FILENAME)

# Process the chemical data
species_list, reaction_list, species_index_map = process_chemical_data(mechanism_data)

# Number of species and reactions
const n_species = length(species_list)
const n_vars = n_species + 1
const n_reactions = length(reaction_list)

# Build the stoichiometric matrix and kinetics list
S = build_stoichiometric_matrix(reaction_list, n_species)
kinetics_list = build_kinetics_list(reaction_list)

config = ChemistryConfig(
        temperature = 1200.0, # Initial temperature in K
        pressure = 1e6,      # Initial pressure in Pa
        fuel_mixture = Dict("CH4" => 1/10.57, "H2"  => 0.00), air_percentage = 9.57/10.57
)
```

```
X0 = initialize_concentrations(config, species_list, species_index_map)

function dT(X::Vector{Float64}, r::Vector{Float64}, S::Array{Float64,2}, species_list::Vector{Specie
    n_species = length(species_list)
    n_reactions = length(r)

    h0_vec = zeros(n_reactions)      # Enthalpy change per reaction (J/mol)
    cp_vec = zeros(n_species)        # Heat capacity per species (J/(mol·K))

    T = X[1]                         # Temperature in K
    concentrations = X[2:end]        # Species concentrations in mol/m³

    # Pre-compute species enthalpies and heat capacities
    h_species = zeros(n_species)     # Enthalpy for each species (J/mol)
    for (species_index, species) in enumerate(species_list)
        cp_vec[species_index] = species_cp(T, species.thermo)
        h_species[species_index] = h0(T, species.thermo)
    end

    # Compute h0_vec for reactions
    for reaction_index in 1:n_reactions
        # Sum over species: stoichiometric coefficient * species enthalpy
        for species_index in 1:n_species
            stoich_coeff = S[species_index, reaction_index]
            if stoich_coeff != 0.0
                h0_vec[reaction_index] += stoich_coeff * h_species[species_index]
            end
        end
    end

    # Compute the total enthalpy change rate (J/(m³·s))
    dH = sum(r .* h0_vec)

    # Compute the total heat capacity of the mixture (J/(m³·K))
    c_p = sum(concentrations .* cp_vec)

    # Compute temperature rate of change (K/s)
    dT_dt = -dH / c_p

    return dT_dt
end

# Define the ODE function
function reaction_ode!(dX, X, p, t)
    # Unpack parameters
    S, kinetics_tuples, species_list, reaction_list, species_index_map = p

    r = zeros(length(reaction_list))

    # Compute reaction rates
    compute_reaction_rates!(r, X, kinetics_tuples, species_list)

    # Compute temperature rate of change
    dT_dt = dT(X, r, S, species_list)
```

```
    # Compute species concentration rate of change
    dC_dt = S * r  # dC/dt = S * r

    # Populate the derivative vector
    dX[1] = dT_dt       # Temperature derivative
    @views dX[2:end] .= dC_dt  # Species concentration derivatives
end

# Define time span and solver settings
tspan = (0.0, 1)
abstol, reltol = 1e-9, 1e-6

# Set up the ODE problem
params = S, kinetics_list, species_list, reaction_list, species_index_map
problem = ODEProblem(reaction_ode!, X0, tspan, params)

# Solve the ODE problem
@time sol = solve(problem,
            KenCarp4(autodiff = AutoFiniteDiff()),
            verbose=false,
            abstol=abstol,
            reltol=reltol)
```

F

Network model Julia code

The problem was set up by running the following:

```
# Import required packages
using DifferentialEquations     # Main ODE solver framework
using LinearAlgebra             # For matrix operations (mul!)
using SparseArrays              # Sparse matrix handling
using KLU                       # Sparse LU factorization
using SparseDiffTools           # For matrix_colors
using PreallocationTools        # For DiffCache
using DiffEqCallbacks           # For PositiveDomain callback
using SciMLSensitivity, Enzyme
using ADTypes

using Plots                     # For plotting results
using YAML                      # For parsing mechanism files

using Graphs                    # For network graph structures
using GraphPlot                 # For visualizing graphs

# Include the chemistry module
include("Chemistry.jl")

# Chemistry.jl is at the heart of these simulations, it loads the data from .yaml files,
# it creates structures and builds a stochiometric matrix and other needed data-objects
# This file also contains the thermodynamic calculations h0 and species_cp
# It is AD supportive... Good enough for AutoEnzyme at least, with the clamp.

"""
Holds the essential, pre-processed chemical data needed for the hot loop.
"""
struct ChemistryParameters
    S::SparseMatrixCSC{Float64, Int64}
    kinetics_list::SplitKinetics  # CRITICAL: This must use our abstract type
    species_list::Vector{Species}             # Needed for thermodynamic lookups
end

"""
Holds pre-allocated cache arrays for AD-compatibility and performance.
"""
```

```
struct CombustionCache{T}
    r_cache::DiffCache{Vector{T}, Vector{T}}
    dC_dt_cache::DiffCache{Vector{T}, Vector{T}}
    c_p_node_temp_cache::DiffCache{Vector{T}, Vector{T}}
    cp_species_vec_cache::DiffCache{Vector{T}, Vector{T}}
    diffusion_term_cache::DiffCache{Vector{T}, Vector{T}}
    diffusion_term_species_cache::DiffCache{Matrix{T}, Vector{T}}
end
# Note: I added the second type parameter to DiffCache, which is common.
# e.g., DiffCache{Vector{Float64}, Vector{Float64}}

"""
The main parameter object passed to the ODE solver.
"""
struct CombustionParameters{T}
    chem::ChemistryParameters
    A::SparseMatrixCSC{Float64, Int64} # Discretization matrix (e.g., for diffusion)
    V::Vector{Float64}               # Cell volumes
    cache::CombustionCache{T}
end

# Loading the reaction mechanism

# This map contains several mechanisms:
# - GRI-Mech 3.0 with 53 species and 325 reactions, filename 'GRI30.yaml'
# - Tianfeng Lu's reduced mechanism for methane, 30 species and 184 reactions, filename 'LuMechanism
# - Charlélie Laurent's reduced mechanism for methane, 17 species and 82 reactions, filename 'Laurer
# - A 2 step mechanism by  Franzelli et al., 6 species and 2 reactions, filename '2step.yaml'
# - A 1 step mechanism (unrecommended), 6 species and 1 reaction, filename '1step.yaml'

const FILENAME = "LaurentMechanism.yaml"

# Load the mechanism file
mechanism_data = load_mechanism_data(FILENAME)

# Process the chemical data
species_list, reaction_list, species_index_map = process_chemical_data(mechanism_data)

# Number of species and reactions
const n_species = length(species_list)
const n_vars = n_species + 1
const n_reactions = length(reaction_list)

# Build the stoichiometric matrix and kinetics list
S = sparse(build_stoichiometric_matrix(reaction_list, n_species))
kinetics_list = build_kinetics_list(reaction_list)

# Create chemistry parameters
chem_params = ChemistryParameters(S, kinetics_list, species_list)

# Creating the reactor network

# Build network graph
const n_nodes = 12
const T_idx = 1:n_vars:(n_nodes-1)*n_vars+1
```

```
g = SimpleGraph(n_nodes)

# Define network topology
edges = [
    (1, 2), (2, 3), (2, 4), (3, 4), (4, 5),
    (5, 6), (5, 7), (6, 7), (7, 8),
    (3, 9), (6,10), (8, 9), (1, 10),
    (8, 11), (9, 11),
    (1, 12), (10, 12),
]

for (i, j) in edges
    add_edge!(g, i, j)
end

# Create connectivity matrix
A = -laplacian_matrix(g)

# Volumes arbitrarily all set to 1
V = ones(n_nodes)

# Set initial conditions by choosing methane nodes
methane_nodes = [4, 7, 9, 12]
hydrogen_nodes = [1, 3, 5]

# Set up the initial state
u0 = zeros(Float64, n_nodes*(n_species + 1))

# Define initial configurations
initial_conditions = Dict(
    :methane => ChemistryConfig(temperature = 1200.0, pressure = 1e6,
                                fuel_mixture = Dict("CH4" => 1/10.57),
                                air_percentage = 9.57/10.57),
    :air     => ChemistryConfig(temperature = 1200.0, pressure = 1e6,
                                fuel_mixture = Dict("CH4" => 0.00),
                                air_percentage = 1.00),
    :hydrogen=> ChemistryConfig(temperature = 1200.0, pressure = 1e6,
                                fuel_mixture = Dict("H2" => 0.99/6.5),
                                air_percentage = 5.5/6.5)
)

# Assign to nodes
for node in 1:n_nodes
    start_idx = (node-1)*n_vars + 1
    end_idx = (node-1)*n_vars + n_vars

    if node in methane_nodes
        config = initial_conditions[:methane]
    elseif node in hydrogen_nodes
        config = initial_conditions[:hydrogen]
    else
        config = initial_conditions[:air]
    end
    u0[start_idx:end_idx] = initialize_concentrations(config, species_list, species_index_map)
end
```

```
# Define diffusion coefficients... which could be modelled better
const D_TEMP =  1000 # Should be a composition and temperature dependent function
const D_SPECIES = 10 # Should be a species specific temperature dependent function

# === Reaction ODE per node ===

# Reaction ODE per node.
function reaction_ode_inplace!(
    dX::AbstractVector{<:Real},
    X::AbstractVector{<:Real},
    param::ChemistryParameters,
    r::AbstractVector{<:Real},
    dC_dt::AbstractVector{<:Real},
    cp_species_vec::AbstractVector{<:Real}
)
    T_val = X[1]

    # --- 1. Compute Reaction Rates ---

    compute_reaction_rates!(r, X, param.kinetics_list, param.species_list)

    # --- 2. Compute Species Source Terms (dC/dt) ---
    # This uses the pre-computed stoichiometric matrix S.
    mul!(dC_dt, param.S, r)
    dX[2:end] = dC_dt

    # --- 3. Compute Temperature Change (dT/dt) ---
    concentrations = X[2:end]

    # Optimized volumetric heat capacity (unchanged, this was already good)
    for i in eachindex(concentrations)
        # species_cp() is your function for species heat capacity
        cp_species_vec[i] = species_cp(T_val, param.species_list[i].thermo)
    end
    c_p_vol = dot(concentrations, cp_species_vec)

    # Reuse cp_species_vec as a vector of entalphies of species
    for i in eachindex(concentrations)
        # species_cp() is your function for species heat capacity
        cp_species_vec[i] = h0(T_val, param.species_list[i].thermo)
    end

    total_heat = -dot(cp_species_vec, dC_dt)

    # dT = dH / c_P (These are all intrinsic quantities: /m3)
    dX[1] = total_heat / c_p_vol

    # Return the volumetric heat capacity for the diffusion term calculation
    return c_p_vol
end

# === Main system ODE ===
function system_ode!(du::AbstractVector{<:Real}, u::AbstractVector{<:Real}, p, t::Real)
    # Unpack parameters and caches (this part is perfect)
    chem, A, V = p.chem, p.A, p.V
    cache = p.cache
```

```
    r = get_tmp(cache.r_cache, u)
    dC_dt = get_tmp(cache.dC_dt_cache, u)
    c_p_node_temp = get_tmp(cache.c_p_node_temp_cache, u)
    cp_species_vec = get_tmp(cache.cp_species_vec_cache, u)

    u = max.(u, 1e-100)  # This creates an artifact that slowly heats the system
                         # But the alternative makes implicit methods unstable

    # === 1. Reaction calculations (per-node) ===
    for i in 1:n_nodes
        node_idx = (i-1)*n_vars + 1: i*n_vars
        @views c_p_node_temp[i] = reaction_ode_inplace!(du[node_idx] , u[node_idx], chem, r, dC_dt,
    end

    # === 2. Diffusion calculations (vectorized) ===
    invV = Diagonal(1.0 ./ V)
    invc_p = Diagonal(1.0 ./ (V .* c_p_node_temp))

    # Temperature diffusion: dC/dt += (1/VcP) * L * (k*T)
    du[T_idx] += invc_p * A * (D_TEMP * u[T_idx])

    # Species diffusion: dC/dt += (1/V) * L * (D*C
    for i in 1:n_species
        du[T_idx .+ i] += invV * A * (D_SPECIES * u[T_idx .+ i])
    end

    return nothing
end

# === Jacobian Sparsity ===
# I analytically derive Jacobian sparsity patterns

function build_local_jacobian_sparsity(param::ChemistryParameters)
    stoich_mat = param.S
    J = spzeros(Bool, n_vars, n_vars)
    for r in 1:size(stoich_mat, 2)
        participants = findall(!iszero, stoich_mat[:, r])
        for i in participants, j in participants
            J[i+1, j+1] = true
        end
    end
    for spec in 1:n_species
        J[spec+1, 1] = true
        J[1, spec+1] = true
    end
    J[1, 1] = true
    return J
end

function build_network_jacobian_sparsity(param::ChemistryParameters, laplacian::SparseMatrixCSC)
    local_jac = build_local_jacobian_sparsity(param)
    n_total = n_vars * n_nodes
    J_network = spzeros(Bool, n_total, n_total)
    for node in 1:n_nodes
        idx_start = (node - 1) * n_vars + 1
        idx_end = node * n_vars
```

```
        idx_range = idx_start:idx_end
        J_network[idx_range, idx_range] = local_jac
    end
    lap_rows, lap_cols, _ = findnz(laplacian)
    for (i, j) in zip(lap_rows, lap_cols)
        if i != j
            for var in 1:n_vars
                row_idx = (i - 1) * n_vars + var
                col_idx = (j - 1) * n_vars + var
                J_network[row_idx, col_idx] = true
            end
        end
    end
    return J_network
end

jac_sparsity = build_network_jacobian_sparsity(chem_params, A)
colorvec = matrix_colors(jac_sparsity)

# Determine the chunk size
chunk_size = maximum(colorvec)

# === System Initialization ===

# Create the parameter object `p` as a NamedTuple

# Function to build the cache
function build_cache(T::Type, n_species, n_reactions, n_nodes, chunk_size)
    # The chunk size is passed to the DiffCache constructor.
    # The second type parameter (e.g., Vector{T}) is what get_tmp will use.
    return CombustionCache{T}(
        DiffCache(zeros(T, n_reactions), chunk_size),          # r_cache
        DiffCache(zeros(T, n_species), chunk_size),            # dC_dt_cache
        DiffCache(zeros(T, n_nodes), chunk_size),              # c_p_node_temp_cache
        DiffCache(zeros(T, n_species), chunk_size),            # cp_species_vec_cache
        DiffCache(zeros(T, n_nodes), chunk_size),              # diffusion_term_cache
        DiffCache(zeros(T, n_nodes, n_species), chunk_size)    # diffusion_term_species_cache
    )
end

p = CombustionParameters(
    chem_params,
    float.(A),
    V,
    build_cache(Float64, n_species, n_reactions, n_nodes, chunk_size)
)

ode_func = ODEFunction(system_ode!, sparsity=jac_sparsity, colorvec=colorvec)

cb = PositiveDomain(; save = false)

println("ready to go!")
```

After which it was solved in three runs. The first:

```
    # === solve(problem) I ===
# Very stiff so we use a stronger method which handles
```

```
# explosion stiffness and its discontinuities well

tspan = (0.0, 1e-5)

problem = ODEProblem(ode_func, u0, tspan, p)

algo = KenCarp3(
    linsolve = KLUFactorization(),
    autodiff = AutoEnzyme(; function_annotation=Enzyme.Duplicated),
    standardtag = false,
    concrete_jac = true
)

println("Beginning integration using $FILENAME")
println("  Time span: $(tspan)")

@time sol1 = solve(problem, algo;
    abstol=1e-12, reltol=1e-9,   # Extreme tolerances improve stability
    callback = cb,
    save_everystep=false,
    saveat=1e-8
)
```

The second:

```
    # === solve(problem) II ===
# More stiff as we enter timescales wirh many active processes at once

tspan = (sol1.t[end], 0.0007)

problem = ODEProblem(ode_func, sol1.u[end], tspan, p)

algo = KenCarp5(
    linsolve = KLUFactorization(),
    autodiff = AutoEnzyme(; function_annotation=Enzyme.Duplicated),
    standardtag = false,
    concrete_jac = true
)

println("Beginning integration using $algo with dataset $FILENAME")
println("  Time span: $(tspan)")

# Problem becomes unstable at too loose tolerance settings

@time sol2 = solve(problem, algo;
    abstol=1e-12, reltol=1e-9,
    save_everystep=false, saveat=1e-7,
    callback = cb)
```

The third:

```
    # === solve(problem) III ===
# Not so stiff so we use an fast method that solves
# stable systems quicker than previous solvers.

tspan = (sol2.t[end], 0.1)

problem = ODEProblem(ode_func, sol2.u[end], tspan, p)
```

```
# Switch to multistep method because no more discontinuities expected
algo = FBDF(
    linsolve = KLUFactorization(),
    autodiff = AutoEnzyme(; function_annotation=Enzyme.Duplicated),
    standardtag = false,
    concrete_jac = true
)

println("Beginning integration using $algo with dataset $FILENAME")
println("  Time span: $(tspan)")

@time sol3 = solve(problem, algo;
    save_everystep = false, dt=1e-5,
    saveat = 0.0001,  # Can use larger save intervals
    maxiters = 1e7,
    reltol = 1e-6,
    abstol = 1e-9,
    callback = cb
)
```

G

Relaxation time function Julia code

```
using LinearAlgebra: norm

function compute_relaxation_time(sol; tol=1e-6)
    """
    Compute relaxation time  for a solution `sol` of an ODEProblem.

     is defined as the time when ||u(t) - u_steady|| / ||u0 - u_steady||  1/e.

    Args:
        sol: Solution object from DifferentialEquations.jl
        tol: Tolerance for checking convergence (default: 1e-6)

    Returns:
         : Relaxation time (first time when decay reaches 1/e)
        If no such time is found, returns `nothing`.
    """
    u0 = sol.prob.u0          # Initial (perturbed) state
    u_steady = sol[end]       # Equilibrium state (final value)
    Δ0 = norm(u0 - u_steady)  # Initial deviation magnitude

    # Handle cases where Δ0  0 (no perturbation)
    if Δ0 < tol
        @warn "Initial state is already at equilibrium (||Δu|| = $Δ0).  is undefined."
        return nothing
    end

    # Target deviation: Δ0 / e
    target_deviation = Δ0 / MathConstants.e

    # Find the first time when ||u(t) - u_steady||  target_deviation
     = nothing
    for (i, t) in enumerate(sol.t)
        Δu = norm(sol.u[i] - u_steady)
        if Δu  target_deviation + tol  # Allow numerical tolerance
             = t
            break
        end
    end
```

```
    if   === nothing
        @warn "No relaxation time found within solution timeframe. Increase `tspan`."
    end

    return
end
```