PelePhysics Documentation

Release 2022.10

J.B. Bell, M.S. Day, E. Motheau, D. Graves, M. Henry de Frahan, R.

Mar 16, 2023

Documentation contents:

1 *PelePhysics* Quickstart 3

1.1 Obtaining *PelePhysics* 3

1.2 Install CVODE and *SuiteSparse* 4

2 Introduction 5

2.1 Objectives and State-Of-The-Art 5

2.2 How to naviguate this documentation 7

3 Chemistry 9

3.1 A brief introduction to CVODE 9

3.2 CVODE implementation in *PelePhysics* 11

3.3 CVODE implementation in *PelePhysics* on GPU 20

3.4 Analytically reduced chemistry via quasi-steady state (QSS) assumption in *PelePhysics* 22

3.5 CEPTR: Chemistry Evaluation for Pele Through Recasting 28

4 Transport 31

5 Equation of State 33

5.1 Soave-Redlich-Kwong (SRK) 33

6 Tutorials 39

6.1 Tutorial 1 - Generating NC12H26 QSS mechanism with analytical Jacobian 39

6.2 Tutorial 2 - Generating NC12H26 QSS mechanism without analytical Jacobian 40

6.3 Tutorial 3 - Generating NC12H26 Skeletal mechanism 40

7 Developer Guidelines 43

7.1 C++ Code 43

7.2 Python 43

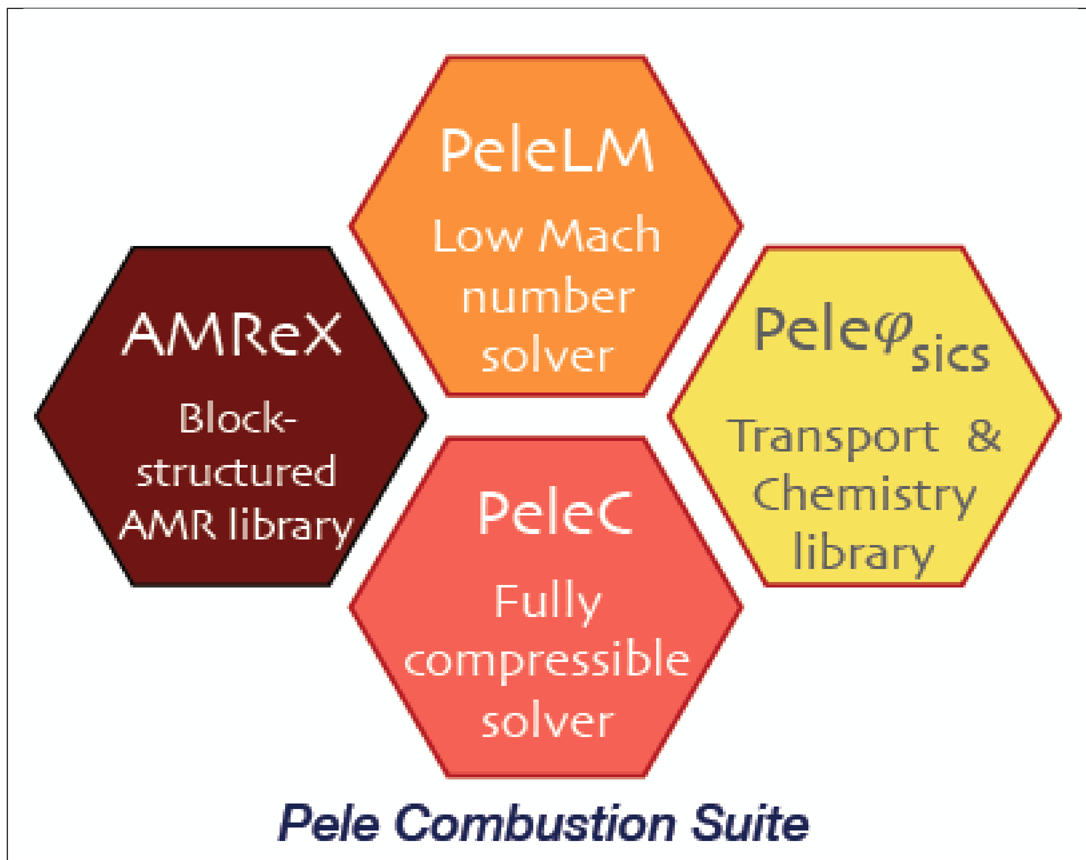
8 Indices and tables 45

Bibliography 47

PelePhysics is a repository of physics databases and implementation code for use within the other *Pele* codes. In particular, the choice of chemistry and transport models as well as associated functions and capabilities are managed in *PelePhysics*. *PelePhysics* has an official project homepage, and can be obtained via GitHub. The documentation pages appearing here are distributed with the code in the `Docs/sphinx` folder as "restructured text" files. The html is built automatically with certain pushes to the *PelePhysics* GibHub repository. A local version can also be built as follows

```
cd ${PELE_PHYSICS_DIR}/build
sphinx-build -M html ../Docs/sphinx .
```

where `PELE_PHYSICS_DIR` is the location of your clone of the *PelePhysics* repository. To view the local pages, point your web browser at the file `${PELE_PHYSICS_DIR}/build/html/index.html`.



CHAPTER 1

PelePhysics Quickstart

Greetings impatient user. Once again, note that this documentation focuses on the CHEMISTRY part of *PelePhysics*.

- If you are familiar with *PelePhysics*, have it installed already and would simply like to know which chemistry-related keywords and/or environment variables to set in your various input files to perform a simulation with one of the codes available in the *PeleSuite*, then I invite you to directly skip to section *Activating the different CVODE solver options via the input files*.
- If you are a complete beginner, I urge you to carefully read the two following chapters *Obtaining PelePhysics* and *Install CVODE and SuiteSparse*, to properly set-up your working environment.
- If you are in a hurry but still would like more context, visit section *Introduction* to be referred to portions of this document that are of interest to you.

1.1 Obtaining *PelePhysics*

First, make sure that "Git" is installed on your machine—we recommend version 1.7.x or higher. Then...

1. Download the *AMReX* repository by typing:

```
git clone https://github.com/AMReX-Codes/amrex.git
```

This will create an `amrex/` folder on your machine. Next, set the environment variable `AMREX_HOME` to point to the location where you have downloaded *AMReX*:

```
export AMREX_HOME=/path/to/amrex/
```

2. Clone the *Pele* repository:

```
git clone git@github.com:AMReX-Combustion/PelePhysics.git
```

This will create a `PelePhysics` folder on your machine. Set the environment variable `PELE_PHYSICS_HOME` to point to the location of this folder.

3. Periodically update both of these repositories by typing `git pull` within each repository.

1.2 Install CVODE and *SuiteSparse*

***The user is in charge of installing the proper CVODE version, as well as installing and properly linking the KLU library if sparsity features are needed.**

1.2.1 SuiteSparse

SuiteSparse is a suite of Sparse Matrix software that is very handy when dealing with big kinetic mechanisms (the Jacobian of which are usually very sparse). In such a case, CVODE can make use of the KLU library, which is part of *SuiteSparse*, to perform sparse linear algebra. Documentation and further information can be found on SuiteSparse website.

At the time this note is written, the recommended **SuiteSparse version** is **5.4.0**. Follow these steps to set-up your working environment and build the required libraries:

1. Go to the SuiteSparse website and download the compressed file for the recommended version
2. Copy the tar file into `$PELE_PHYSICS_HOME/ThirdParty`
3. Untar ('tar -zxvf'), cd into it and type 'make' into the following folders: `SuiteSparse_config`, `AMD`, `COLAMD`, `BTF`
4. Go into `metis-5.1.0` and type 'make config shared=1' followed by 'make'
5. Go into `KLU` and type 'make'
6. Check that all dynamic libraries have correctly been generated and copied into the folder `$PELE_PHYSICS_HOME/ThirdParty/SuiteSparse/lib`
7. It is recommended that you add the path `$PELE_PHYSICS_HOME/ThirdParty/SuiteSparse/lib` to your `LD_LIBRARY_PATH`, for precaution
8. Note that depending upon your compiler, the static `.a` versions of the libraries might also be required. In such a case, you can copy them directly from each program folder into the `SuiteSparse/lib` folder

1.2.2 CVODE

CVODE is a solver for stiff and nonstiff ordinary differential equation (ODE) systems. Documentation and further information can be found online. At the time this note is written, the recommended **CVODE version** is **v5.0.0**.

The CVODE sources are distributed as compressed archives, with names following the convention `cvode-x.y.z.tar.gz`. They can be downloaded by following this link. However, we have designed a simple script enabling to install the current version the correct way. Simply:

1. Go into `$PELE_PHYSICS_HOME/ThirdParty`
2. Execute either `get_sundials_v5dev1.sh` or `get_sundials_v5dev1_CUDA.sh` depending on your application (GPU or not) and machine
3. Set the `SUNDIALS_LIB_DIR` environment variable to point to the location where all CVODE libraries have been generated. If you followed these guidelines, it should be `$PELE_PHYSICS_HOME/ThirdParty/sundials/instdir/lib/`
4. It is recommended, here also, that you add the path `$PELE_PHYSICS_HOME/ThirdParty/sundials/instdir/lib` to your `LD_LIBRARY_PATH`, as paths can get lost in the build of external libraries

Note that if you do not want to use the KLU library, you can also disable the flags (`-DKLU_ENABLE`) in the `*.sh` scripts.

CHAPTER 2

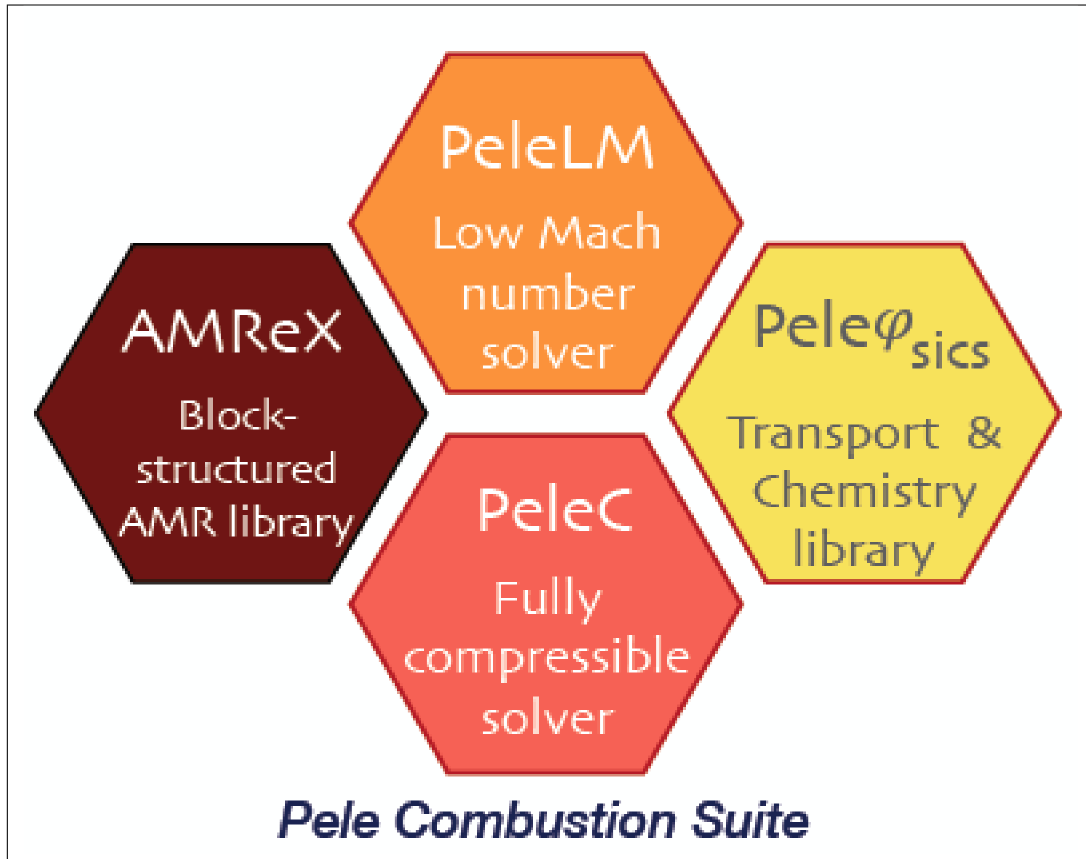
Introduction

2.1 Objectives and State-Of-The-Art

What we will call the *PeleSuite* is currently composed of 3 separate codes:

- PelePhysics is a repository of physics databases and implementation code for use within the other *Pele* codes. In particular, the choice of chemistry and transport models as well as associated functions and capabilities are managed in *PelePhysics*.
- PeleLM is an adaptive-mesh Low-Mach number hydrodynamics code for reacting flows. It has a sibling, PeleLMeX, that solves for the same type of flow using a subtly different numerical approach.
- PeleC is an adaptive-mesh compressible hydrodynamics code for reacting flows.

All three codes rely on AMReX, which is a software frameworks that provides the data structure and enable massive parallelization.



PelePhysics (as well as the *ChemDriver* object of *PeleLM*) used to rely upon DVODE [VODE1989] to perform the chemistry integration, no matter the problem at hand. DVODE is a very robust, but slightly outdated, variable-coefficient Ordinary Differential Equation (ODE) solver written in Fortran 77. At its core, it uses a direct dense linear solver. DVODE is very efficient in the resolution of small stiff systems of equations but can become prohibitively expensive when dealing with bigger systems of equations, such as those frequently encountered in combustion systems.

In recent years, the Sundials team at LLNL [LLNL2005] has been involved in the active development of a modern, C++ version, of DVODE called CVODE. CVODE implements the same functionalities as those available in DVODE, but offers much more flexibility through its user-friendly set of interface routines. Additionally, other linear solvers are available, such as iterative or sparse solvers which can prove to be very efficient in handling "larger" systems of equations.

The objective of this user-guide is to document the CVODE-based chemistry integration implemented in *PelePhysics*. Although it is possible to use CVODE in *PeleC*, the following is mainly intended for *PeleLM* users. This user-guide will cover:

- ODE equations (*reactor type*)
- Default settings (tolerances/order/. . .)
- Linear solvers available –along with exemples of performance
- Setting-up a *PelePhysics* test case
- . . .

2.2 How to navigate this documentation

This section provides a short overview of the chemistry-related features of PelePhysics. For an in-depth discussion, relevant references to specific sections of this manuscript are given.

- In *PelePhysics*, the user can select between **two different reactor types**. The first type is a *constant volume* reactor, were the choice of fixed variables are the internal energy and the density, to comply with *PeleC* transported variables. This reproduces what was originally implemented with DVODE in *PelePhysics*. A second reactor type formulated in terms of enthalpy and density has been put in place, to comply with *PeleLM*. Both reactors are available in DVODE (in fortran 90) and CVODE (in cpp). See sections *The different reactors* for additional details, and *Activating the different CVODE solver options via the input files* to see how to activate one or the other via the input files.
- With both reactors, it is possible to use an **Analytical Jacobian** (depending upon the choice of linear solver, see section *Linear Algebra*.)
- Three different types of **linear solvers** are implemented, see section *Linear Algebra* for additional details, and *Activating the different CVODE solver options via the input files* to see how to make a selection:
  - a dense direct solver – with or without Analytical Jacobian
  - a sparse direct solver (requires the KLU library) – always requires an Analytical Jacobian
  - a couple of sparse iterative solvers from the GMRES family – preconditioned or not
- **Regression testings** have been put in place in *PelePhysics* to test the CVODE integration. See section *Validation of the CV reactor implementation in CVODE (with CANTERA)* for validations, and section *The ReactEval_C test case with CVODE in details* for a step-by-step example.
- A CVODE version running on **GPU** is also technically available but the documentation is a WIP. The interested user can contact code developers for additional information
- *PelePhysics* uses an automatic chemistry-related routine generator: **CEPTR**. CEPTR is part of the sources of *PelePhysics*. With CEPTR, a unique chemistry file is generated for a specific kinetic scheme. Instructions to generate your own chemistry files (all you need are chemistry files in the famous Cantera yaml format) are discussed in section *CEPTR: Chemistry Evaluation for Pele Through Recasting*. Note that *PelePhysics* already offers a large choice of more than 20 different kinetic schemes.

CHAPTER 3

Chemistry

3.1 A brief introduction to CVODE

CVODE is part of a software family called sundials for SUite of Nonlinear and DIfferential / ALgebraic equation Solver [LLNL2005].

The version used with PelePhysics for the tests presented in this document is v4.0.2. This does not mean this is the recommended version to link with PelePhysics. Always refer to *Install CVODE and SuiteSparse* to know which versions of CVODE/SuiteSparse to use !!

In the following, details pertaining to the methods implemented in *PelePhysics* are provided. The interested user is referred to the very exhaustive CVODE User-guide for more information.

Most of this section is adapted from the v4.1.0 Cvode Documentation.

3.1.1 Numerical methods overview

The methods implemented in CVODE are variable-order, variable-step *multistep* methods, based on formulas of the form

$$\sum_{i=0}^{K_1} \alpha_{n,i} y^{n-i} + h_n \sum_{i=0}^{K_2} \beta_{n,i} \dot{y}^{n-i} = 0$$

Here the y^n are computed approximations to $y(t_n)$, and $h_n = t_n - t_{n-1}$ is the step size. For stiff problems, CVODE includes the Backward Differentiation Formulas (BDF) in so-called fixed-leading coefficient (FLC) form, given by $K_1 = q$ and $K_2 = 0$, with order q varying between 1 and 5. The coefficients are uniquely determined by the method type, its order, the recent history of the step sizes, and the normalization $\alpha_{n,0} = -1$ [BYRNE1975], [JAC1980].

A nonlinear system must be solved (approximately) at each integration step. This nonlinear system can be formulated as a root-finding problem

$$F(y^n) = y^n - h_n \beta_{n,0} f(t_n, y^n) - a_n = 0$$

where $a_n = \sum_{i>0}(\alpha_{n,i} y^{n-i} + h_n \beta_{n,i} \dot{y}^{n-i})$. CVODE provides several non-linear solver choices. By default, CVODE solves this problem with a Newton iteration, which requires the solution of linear systems

$$M[y^{n(m+1)} - y^{n(m)}] = -F(y^{n(m)}) \tag{3.1}$$

in which

$$M \approx I - \gamma J, \quad J = \frac{\partial f}{\partial y}, \quad and \quad \gamma = h_n \beta_{n,0}$$

3.1.2 Linear Algebra

To find the solution of the linear system (3.1); CVODE provides several linear solver choices. The linear solver modules distributed with Sundials are organized in two families, a *direct* family comprising direct linear solvers for dense, banded, or sparse matrices, and a *spils* family comprising scaled preconditioned iterative (Krylov) linear solvers. The solvers offered through these modules that are of interest to us are:

- a dense direct solver
- a sparse direct solver interface using the *KLU* sparse solver library
- SPGMR, a scaled -possibly preconditioned- GMRES (Generalized Minimal Residual method) solver [BROWN1990]

When using a dense direct solver, the user has the option to specify an Analytical Jacobian. If none is provided, a difference quotients is performed. When a sparse direct solver is employed however, the user **must** specify an analytical Jacobian. All of these options have been enabled in *PelePhysics*.

For large stiff systems, where direct methods are often not feasible, the combination of a BDF integrator and a *preconditioned* Krylov method yields a powerful tool. In this case, the linear solve is *matrix-free*, and the default Newton iteration is an *Inexact* Newton iteration, in which M is applied with matrix-vector products Jv obtained by either difference quotients or a user-supplied routine. In *PelePhysics*, it is possible to use either a non-preconditioned or a preconditioned GMRES solver. In the latter case, the preconditioner can be either specified in a dense or sparse format (if the KLU library is linked to CVODE), and it is provided in the form of a Jacobian approximation, based on the work of [McNenly2015].

3.1.3 Error control, step-sizing, order determination

In the process of controlling errors at various levels, CVODE uses a weighted root-mean-square norm, denoted $|| \bullet ||_{WRMS}$, for all error-like quantities. The multiplicative weights used are based on the current solution and on the relative and absolute tolerances input by the user, namely

$$W_i = \frac{1}{[rtol|y_i| + atol_i]}$$

Because $1/W_i$ represents a tolerance in the component y_i, a vector whose norm is 1 is regarded as small. In *Pele-Physics*, both these tolerances are fixed to a value of $1.0e-10$.

A critical part of CVODE - making it an ODE *solver* rather than just an ODE method, is its control of the local error. At every step, the local error is estimated and required to satisfy tolerance conditions, and the step is redone with reduced step size whenever that error test fails. Note that in *PelePhysics*, the first time step is always forced to $1.0e-9$.

In addition to adjusting the step size to meet the local error test, CVODE periodically adjusts the order, with the goal of maximizing the step size. The integration starts out at order 1 and varies the order dynamically after that. The basic idea is to pick the order q for which a polynomial of order q best fits the discrete data involved in the multistep method. In *PelePhysics*, the maximum order is limited to 2 for stability reasons.

The various algorithmic features of CVODE are inherited from VODE and VODPK, and are documented in [VODE1989] and [BROWN1990]. They are also summarized in the CVODE User-guide as well as in [LLNL2005].

3.2 CVODE implementation in *PelePhysics*

3.2.1 The different reactors

Throughout this document, what we call a *reactor* is in fact a zero-dimensional model, the simplest representation of a chemically reacting system. Depending upon the choice of state variables driving the system, several different types of reactor can be considered; and the "correct" choice is case dependent. In general, the state variables for a reactor model are

- The reactor mass
- The reactor volume
- The energy of the system
- The mass fractions for each species

The most common type of reactor is the *constant-volume* (CV) reactor, which is the one used to advance the chemistry within *PeleC*. This reactor type is equivalent to a rigid vessel with fixed volume but variable pressure. In *PelePhysics*, the constant-volume constraint is ensured by keeping the density ρ fixed -since there is no change of mass; and the indirect choice of energy in the CV reactor implementation is the total energy E. E's evolution in our case is solely due to a constant external source term $\dot{E}_{ext}$, which accounts for the effects of advection and convection in the Spectral Deferred Correction (SDC) scheme that all *Pele* codes use (see the PeleLM documentation for example). In that sense, the CV reactor is an abstraction and is not a true closed vessel.

Note that CVODE still integrates the mass fractions (ρY) together with energy for stability reasons, but a change of variable is applied to effectively transport the temperature T via

$$\rho C_v \frac{\partial T}{\partial t} = \rho \dot{E}_{ext} - \sum_k e_k \dot{\omega}_k^M$$

where the e_k are the species internal energy and $\dot{\omega}_k^M$ is the species k mass production rate.

In a second implementation, that we will label *constant-volume-enthalpy* (CVH), the mass-weighted total enthalpy ρH is used and conserved along with ρ. This reactor type is also an abstraction. Here also, ρH evolves according to an external source term $\dot{\rho H}_{ext}$, and in CVODE, the mass fractions (ρY) and temperature T are integrated according to

$$\rho C_p \frac{\partial T}{\partial t} = \rho \dot{H}_{ext} - \sum_k h_k \dot{\omega}_k^M$$

where the h_k are the species internal energy.

Validation of the CV reactor implementation in CVODE (with CANTERA)

CANTERA is an open-source suite of tools for problems involving chemical kinetics, thermodynamics, and transport processes. It is a very robust and fast tool written in C++ that is also based on CVODE to perform the chemistry integration. CANTERA is well recognized in the combustion community, and by comparing our results to CANTERA reactor simulations, we will be able to validate our implementation.

Note that only the CV reactor model described above can be validated, since as we discussed before, the CVH reactor model is an abstraction needed for our Low-Mach PeleLM chemistry integration. Also, to have a real CV reactor, the external source terms for the energy and species equations in *PelePhysics* have been set to 0 (see *The different reactors*).

The parameters chosen to initialize the simulation in both CANTERA and *PelePhysics* are described in Table 3.1. The kinetic mechanism used for hydrogen combustion is available in *PelePhysics*. Note that small sub-steps are explicitly taken until the final time is reached, but CVODE's internal machinery can subdivides the dt even further. For the

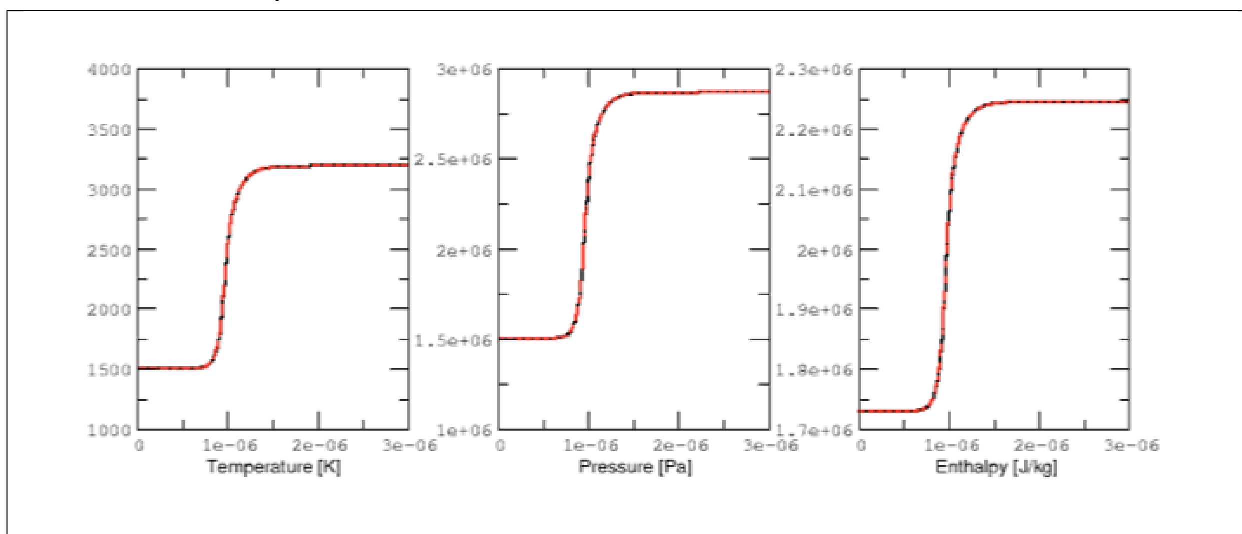
purpose of validation, the direct dense solver of CVODE is selected in *PelePhysics* (see section *Activating the different CVODE solver options via the input files*).

3.1: Parameters for initializing simulation

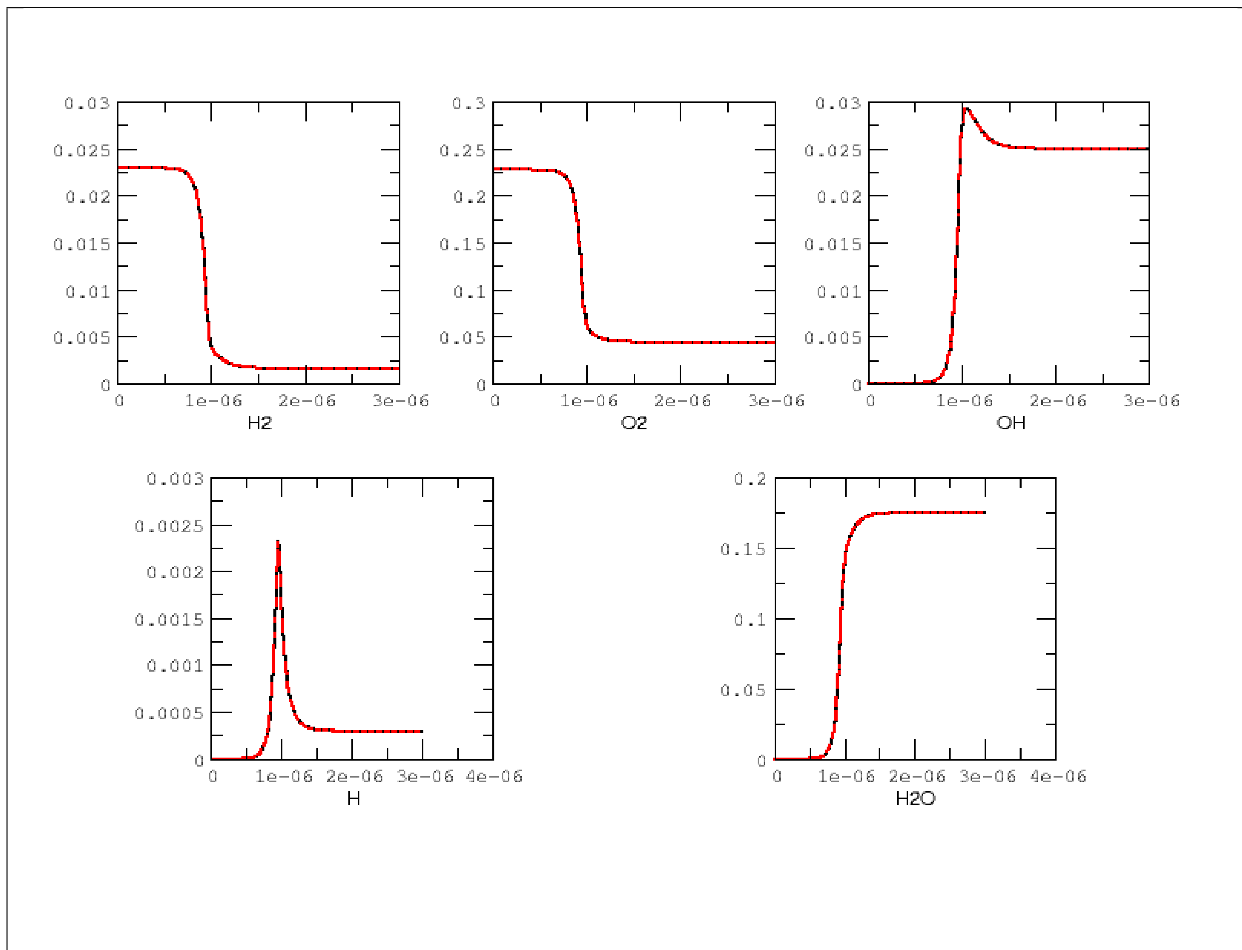
Mechanism	Mixture	Initial T	Initial phi	Pressure	dt	Final time
Li Dryer	H2/O2	1500 K	0.8	101325 Pa	1.0e-8s	3.0e-6s

Results are plotted in Fig 3.2 and 3.3. for the H_2/O_2 mixture. All curves are indistinguishable, so the relative error of all major quantities is also plotted in Fig. 3.4. Note that H_2 and O_2 relative errors have similar features, and that relative errors observed for H and H_2O are representative of those exhibited by, respectively, intermediates and products.

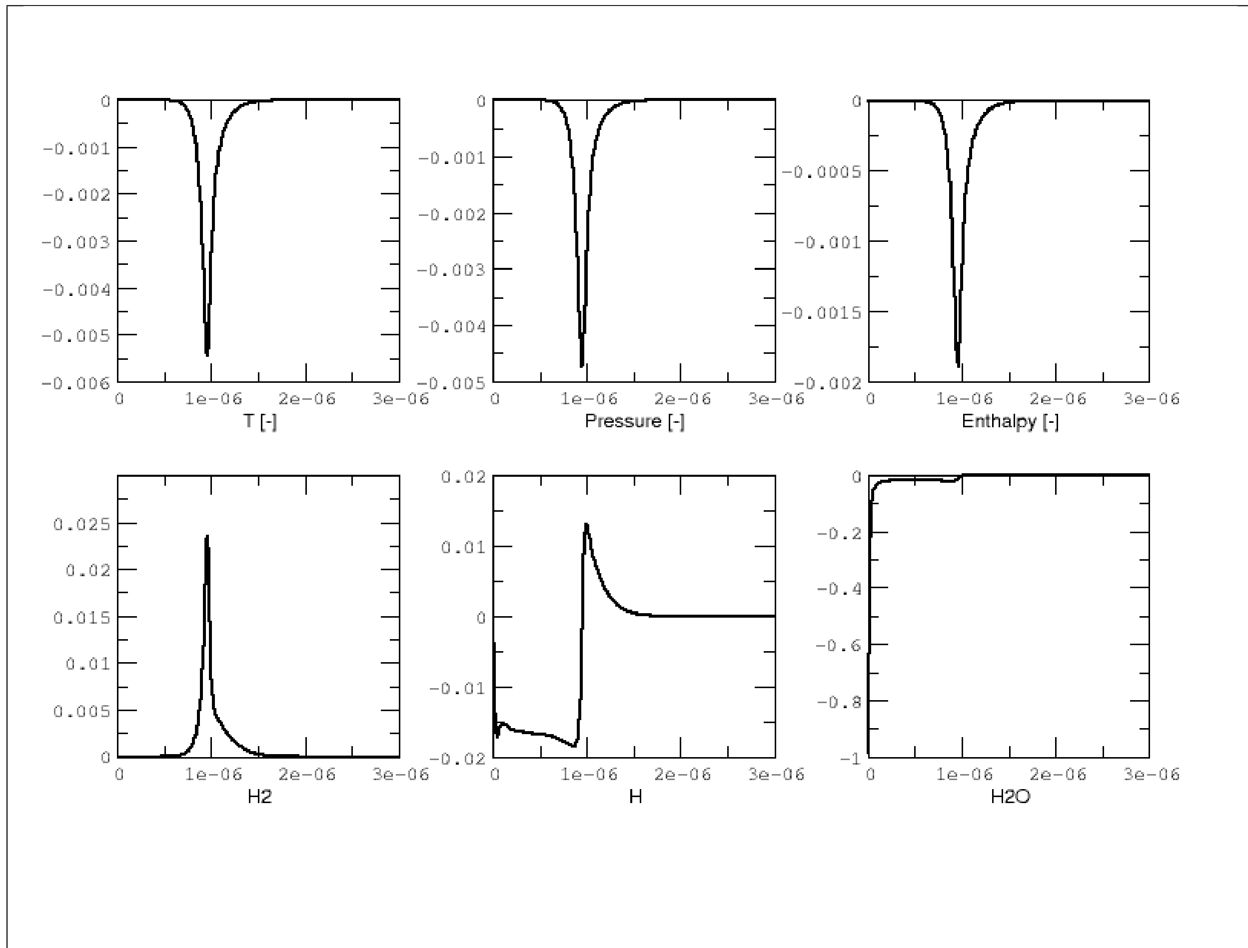
3.2: Evolution of temperature, pressure and enthalpy in a CV reactor, computed with the LiDryer mechanism. Black: CANTERA, red: PelePhysics.



3.3: Evolution of major species in a CV reactor, computed with the LiDryer mechanism. Black: CANTERA, red: PelePhysics.



3.4: Relative errors on the temperature, pressure, enthalpy and major species in a CV reactor, computed with the LiDryer mechanism.



Overall, considering the many CVODE controlling parameters, results are deemed acceptable and that concludes the validation of the reactors implemented in *PelePhysics*.

3.2.2 Activating the different CVODE solver options via the input files

Note that at this point, it is believed that the user has properly installed CVODE as well as the SuiteSparse package. If not, refer to *Install CVODE and SuiteSparse*.

Choosing between DVODE/CVODE (as well as other ODE integrators that will not be discussed in this section) is done at compile time, via the `GNUmakefile`. On the other hand, the type of reactor and specifics of the numerical algorithm are selected via keywords in the input file. There is a subtlety though: when any sparsity feature is required, the choice should also be made at compile time since external libraries will be required; and if the compilation is not performed properly, subsequent options via keywords in the input file can either lead to an error or fall back to a dense formulation of the problem. This is discussed in more depth in what follows.

The GNUmakefile

The default setting is to use DVODE in *PelePhysics*; i.e, if no modifications are done to the original `GNUmakefile` (see the test case ReactEval_FORTRAN of *PelePhysics*), then this option should automatically be selected. To activate

CVODE, the user must first activates the use of Sundials via the following line:

```
USE_SUNDIALS_PP = TRUE
```

Note that this is a *PelePhysics* flag, so it will automatically be recognized in the *Pele* codes. However, if CVODE has not been installed as prescribed in *Install CVODE and SuiteSparse* then a line specifying the location of the Sundials libraries should be added:

```
CVODE_LIB_DIR=PathToSundials/instdir/lib/
```

By default, if Sundials is used then the implicit ODE solver CVODE is selected. The user then has to choose between a number of different methods to integrate the linear system arising during the implicit solve. Add the following line if sparsity features are required:

```
PELE_USE_KLU = TRUE
```

Likewise, if *SuiteSparse* has not been installed as prescribed in *Install CVODE and SuiteSparse*, then a line specifying its location should be added:

```
SUITESPARSE_DIR=PathToSuiteSparse/
```

All of the flags discussed in this subection are used in `$PELE_PHYSICS_HOME/ThirdPartyThirdParty/Make.ThirdParty`.

The input file

The input file is made up of specific blocks containing keywords that apply to specific areas of the integrationof the problem at hand. The suffix associated with each block of keywords should help the user in determining which keywords are needed in his case, depending on the options selected via the `GNUmakefile`. If CVODE is enabled via the `GNUmakefile`, for example, keywords starting with `cvode.*` are relevant. The general `ode.*` keywords are shared by all ODE integrators and thus are also relevant for CVODE:

- `ode.reactor_type` enable to switch from a CV reactor (=`1`) to a CVH reactor (=`2`).
- `cvode.solve_type` controls the CVODE linear integration method: choose `1` to enable the dense direct linear solver, `5` for the sparse direct linear solver (if the KLU library has been linked) and `99` for the Krylov iterative solver
- `ode.analytical_jacobian` is a bit less obvious:
  - If `cvode.solve_type = 1`, then `ode.analytical_jacobian = 1` will activate the use of an Analytical Jacobian.
  - If `cvode.solve_type = 99`, then `ode.analytical_jacobian = 1` will activate the preconditioned GMRES solver while `ode.analytical_jacobian = 0` will activate the non-preconditioned GMRES solver.
  - If `cvode.solve_type = 99`, `ode.analytical_jacobian = 1` **and** the KLU library is linked, then the preconditioned solve is done in a sparse format.
  - With `cvode.solve_type = 5`, the only allowed option is `ode.analytical_jacobian = 1`.

3.2.3 The ReactEval_C test case with CVODE in details

This tutorial has been adapted from the *ReactEval_FORTRAN* tutorial employed in the series of regression tests to monitor the DVODE chemistry integration. The domain considered is a $2x1024x2$ box, where the initial temperature

is different in each $(i, j, k)-$ cell, according to a $y-$ evolving sinusoidal profile, see Fig. 3.1:

$$T(i,j,k) = T_l + (T_h - T_l)\frac{y(i,j,k)}{L} + dT sin\left(2\pi\frac{y(i,j,k)}{P}\right)$$

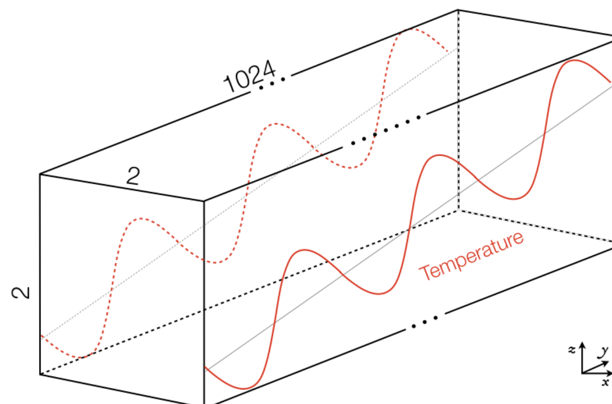
The different parameters involved are summarized in Table 3.5. The initial pressure is 1 atm. The initial composition is the same in every cell, and is a mixture of 0.1 C_nH_m, 0.2 O_2 and 0.7 N_2 in mass fractions.

Various fuels and kinetic mechanisms can be employed. For the purpose of this tutorial, two common fuels will be considered: **methane** (n=1 and m=4) and **n-dodecane** (n=12 and m=26), modelled via the **drm** and **dodecane_wang** kinetic schemes, respectively. Both mechanisms are available in *PelePhysics*.

The following focuses on the CH_4/O_2 example, but performances for both mechanisms and initial composition will be reported in the results section.

3.5: Parameters used to initialize T in the ReactEval_C test case

Tl	Th	dT	L	P
2000 K	2500 K	100 K	1024	L/4



3.1: The ReactEval_C test case

The GNUmakefile

For this example, the `USE_SUNDIALS_PP` flag should be set to true, as the ODE integration is called from the C++ routine directly using CVODE. Additionally, the `FUEGO_GAS` flag should be set to true and the chemistry model should be set to `drm19`. The full file reads as follows:

Note that the `TINY_PROFILE` flag has been activated to obtain statistics on the run. This is an *AMREX* option.

The input file

The run parameters that can be controlled via `inputs.3d` input file for this example are as follows:

```
#ODE solver options
# REACTOR mode
ode.dt = 1.e-05
ode.ndt = 10
# Reactor formalism: 1=full e, 2=full h
ode.reactor_type = 1
```

(continues on next page)

(continued from previous page)

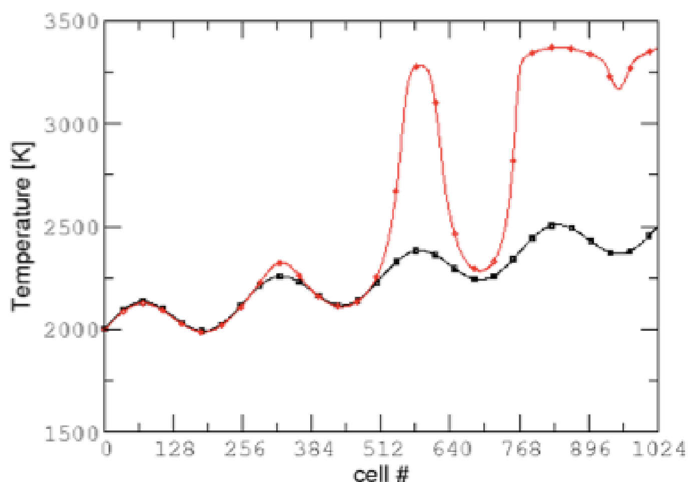
```
# Tolerances for ODE solve
ode.rtol = 1e-9
ode.atol = 1e-9
# Select ARK/CV-ODE Jacobian eval: 0=FD 1=AJ
ode.analytical_jacobian = 0
#CVODE SPECIFICS
# Choose between sparse (5) dense (1/101) iterative (99) solver
cvode.solve_type = 1

#OTHER
# Max size of problem
max_grid_size = 2
# Choose name of output pltfile
amr.plot_file       = plt
# Fuel species
fuel_name = CH4
```

so in this example, a **CV reactor model is chosen** to integrate each cell, and the **dense direct solve without analytical Jacobian** is activated. Each cell is then integrated for a total of $1.e-05$ seconds, with 10 external time steps. This means that the actual dt is $1.e-06s$, which is more than what is typically used in the *PeleC* code, but consistent with what is used in *PeleLM*. Note that the fuel is explicitly specified to be methane. By default, the number of cells integrated simultaneously by one CVODE instance is 1[1], but the *AMREX* block-integration proceeds by blocks of $2x2x2$.

Results

It took 52.61s to integrate the 4096 cells of this box, with 4 MPI processes and no OMP process. The resulting temperature evolution for all cells in the y-direction is displayed in Fig. 3.2.



3.2: Evolution of temperature in the 2x1024x2 example box, using a CV reactor and a dense direct solve, and computed with the DRM mechanism. Black: t=0s, red: t=1e-05s

[1] NOTE that only one cell at a time should be integrated with CVODE right now. The vectorized version on CPU is still WIP and not properly implemented for all linear solvers so that no computational gain should be expected from solving several cells at a time.

3.2.4 To go further: ReactEval_C with CVODE and the KLU library

The GNUmakefile

Only the middle part of the `GNUmakefile` needs to be modified compared to the previous example.

The input file

For the KLU library to be of use, a solver utilizing sparsity features should be selected. We modify the input file as follows:

So that now, a preconditioned iterative Krylov solver is selected, where the preconditioner is specified in a sparse format.

Results

This run now takes 1m34s to run. As expected from the dense Jacobian of the system obtained when using the small DRM mechanism (the fill in pattern is $> 90\%$), using an iterative solver does not enable to reach speed-ups over the simple dense direct solve. **NOTE**, and this is important, that this tendency will revert when sufficiently small time steps are used. For example, if instead of $1e-6s$ we took time steps of $1e-8s$ (consistent with *PeleC* time steps), then using the iterative GMRES solver would have provided significant time savings. This is because the smaller the time step the closer the system matrix is from the identity matrix and the GMRES iterations become really easy to complete.

This example illustrates that choosing the "best" and "most efficient" algorithm is far from being a trivial task, and will depend upon many factors. Table 3.6 provides a summary of the CPU run time in solving the ReactEval_C example with a subset of the various available CVODE linear solvers. As can be seen from the numbers, using an AJ is much more efficient than relying upon CVODE's built-in difference quotients. Using a sparse solver does not appear to provide additional time savings.

3.6: Summary of ReactEval_C runs with various algorithms (methane/air)

Solver	Direct Dense	Direct Dense AJ	Direct Sparse AJ	Iter. not Precond.	Iter. Precond. (S)
KLU	OFF	OFF	ON	OFF	ON
ode.reactor_type	1	1	1	1	1
cvode.solve_type	1	1	5	99	99
ode.analytical_jacobian	0	1	1	1	1
Run time	52.61s	44.87s	48.64s	1m42s	1m34s

The same series of tests are performed for a mixture of n-dodecane and air (see *The ReactEval_C test case with CVODE in details*), the configuration being otherwise the same as in the methane/air case. Results are sumarized in Table 3.7. The overall tendencies remain similar. Note that the non-preconditioned GMRES solver becomes very inefficient for this larger system. Here also, the direct sparse solve –which relies upon the KLU library, does not seem to provide additional time savings. The fill-in pattern is 70%.

3.7: Summary of ReactEvalCvode runs with various algorithms (n-dodecane/air)

Solver	Direct Dense	Direct Dense AJ	Direct Sparse AJ	Iter. not Pre-cond.	Iter. Precond. (S)
KLU	OFF	OFF	ON	OFF	ON
ode.reactor_type	1	1	1	1	1
cvode.solve_type	1	1	5	99	99
ode.analytical_jacobian	0	1	1	1	1
Run time	6m25s	5m33s	6m32s	21m44s	10m14s

3.2.5 Current Limitations

Note that currently, all sparse operations rely on an Analytical Jacobian. This AJ is provided via the chemistry routines dumped by the CEPTR code. Those routines are generated in a pre-processing step, when the sparsity pattern of the AJ is still unknown. As such, all entries of the AJ are computed at all times, and when a sparsity solver is chosen, the AJ is in fact "sparsified" to take advantage of the sparse linear algebra. The "sparsification" process involves a series of loop in the cpp that takes a significant amount of the CPU time most of the time. However, it is always good to verify that this is the case. *AMREX*'s `TINY_PROFILER` features is a handy tool to do so.

3.2.6 Tricks and hacks, stuff to know

When using DVODE, there is a *hack* enabling the user to reuse the Jacobian instead of reevaluating it from scratch. This option is triggered when setting the `extern_probin_module` flag `new_Jacobian_each_cell` to `0`. This can be done in *PelePhysics* by adding the following line in the `probin` file:

```
&extern
 new_Jacobian_each_cell = 0
/
```

A similar feature is currently not available in CVODE, although it would be possible to modify the `CVodeReInit` function to reinitialize only a subset of counters. This is currently under investigation. The user still has some control via the CVODE flag `CVodeSetMaxStepsBetweenJac`.

3.2.7 How does CVODE compare with DVODE ?

Depending on whether the famous Jacobian *hack* is activated or not in DVODE, the completion time of the run can be decreased significantly. The same test case as that described in the previous section can also be integrated with DVODE. For that purpose, the FORTRAN routines implementing the DVODE integration have been interfaced with C++ via a FORTRAN header. The run is thus identical to ReactEval_C with CVODE. Only the `GNUmakefile` needs to be modified:

```
...
#######################
# ODE solver OPTIONS: DVODE (default) / SUNDIALS / RK explicit
#######################
# Activates use of SUNDIALS: CVODE (default) / ARKODE
USE_SUNDIALS_PP = FALSE
...

#######################
...
```

and, as explained in section *Tricks and hacks, stuff to know*, the famous AJ *hack* can be activated via the `probin` file.

Two runs are performed, activating the hack or not. Times are reported in Table 3.8.

3.8: Summary of a CVODE vs a DVODE chemistry integration on the same test case

Solver	Direct Dense	Direct Dense	Direct Dense + *hack*
KLU	OFF	OFF	OFF
USE_SUNDIALS_PP	ON (CVODE)	OFF (DVODE)	OFF (DVODE)
ode.reactor_type	1	1	1
cvode.solve_type	1	N/A	N/A
ode.analytical_jacobian	0	N/A	N/A
Run time	52.61s	53.21s	52.83s

In this case, the hack does not seem to provide significant time savings. Note also that CVODE is usually slightly more efficient than DVODE, consistently with findings of other studies available in the literature – although in this case all options give comparable results.

3.3 CVODE implementation in *PelePhysics* on GPU

3.3.1 Requirements and input files

To use CVODE on a GPU, Sundials should be build with the flag `CUDA_ENABLE` . A CUDA compiler also needs to be specified. Relevant information is provided in the Sundials install guide, and an automatic script is distributed with PelePhysics to ease the process. Refer to *Install CVODE and SuiteSparse*.

Note that the SuiteSparse package does not support GPU architecture and is thus no longer required. Sparse linear algebra operations, when needed, are performed with the help of CUDA's cuSolver.

The GNUmakefile

To run on GPUs, *AMREX* should be build with CUDA enabled. To do so, add this line to the `GNUmakefile`:

```
USE_CUDA    = TRUE
```

This should activate the CUDA features of CVODE in *PelePhysics* too.

The input file

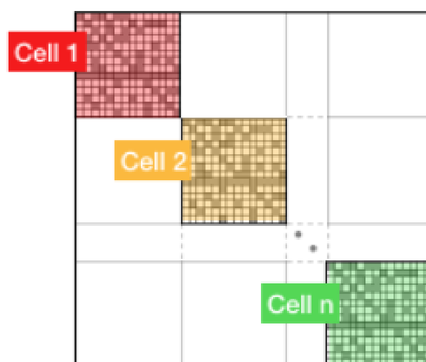
In the `inputs.3d`, the same three main keywords control the algorithm (`ode.reactor_type`, `cvode.solve_type`, `ode.analytical_jacobian`). However, note that there are less linear solver options available.

- Both preconditioned or non-preconditioned GMRES options are available (`cvode.solve_type = 99`). The preconditioned version is triggered via the same flag as on the CPU (`ode.analytical_jacobian = 1`).
- The user has the choice between two different sparse solvers.
  - Sundials offers one option (the SUNLinSol_cuSolverSp_batchQR) relying upon the cuSolver to perform batched sparse QR factorizations. This version is enabled via `cvode.solve_type = 5` and `ode.analytical_jacobian = 1`.

- Another version is available via `cvode.solve_type = 1` and `ode.analytical_jacobian = 1`. This version relies upon a pre-computed Gauss-Jordan Solver, and is fairly efficient for problems of moderate size.

3.3.2 Grouping cells together

To take full advantage of the GPU power, many intensive operations of similar nature should be performed in parallel. In *PelePhysics*, this is achieved by grouping many cells together, and integrating each one in separate threads within one CVODE instance. Indeed, the flow of operations to solve one set of ODEs is very similar from one cell to the next, and one could expect limited thread divergence from this approach. Fig. 3.3 summarizes the idea. Note that the Jacobian of the group of cells is block-sparse, and any chosen integration method should take advantage of this.



3.3: n cells are solved together in one CVODE instance. The big-matrix is block-sparse.

In the current implementation, the number of cells that are grouped together is equal to the number of cells contained in the box under investigation within a MultiFab iteration.

3.3.3 The ReactEval_C_GPU test case in details

A series of tests are performed on the GPU for a mixture of methane and air, with the intent of evaluationg the performance of the chemistry solvers. The test case, configuration and initial conditions are similar to that described in *The ReactEval_C test case with CVODE in details*. The mechanism employed is the **drm**.

The GNUmakefile

The full file reads as follows:

The input file

The Results

Results are sumarized in Table 3.9.

3.9: Summary of ReactEvalCvode_GPU runs with various algorithms (methane/air)

Solver	Direct Sparse I	Direct Sparse II	Iter. not Precond.	Iter. Precond. (S)
reactor_type	1	1	1	1
cvode.solve_type	1	5	99	99
ode.analytical_jacobian	1	1	0	1
Run time	13s	20s	19s	36s

3.3.4 Current Limitations

The current GPU implementation of CVODE relies on the launch of many kernels from the host. As such, a CVODE instance does not live *directly* on the GPU; rather, the user is in charge of identifying and delegating computation-intensive part of the RHS, Jacobian evaluation, etc. The current implementation thus suffers from the cost of data movement, and parallelization is limited due to required device synchronizations within CVODE.

3.4 Analytically reduced chemistry via quasi-steady state (QSS) assumption in *PelePhysics*

3.4.1 The QSS assumption

Given a detailed chemical mechanism, some species can sometimes be assumed to be in a *quasi-steady state (QSS)*. Formally, if species A is assumed to be in a QSS then

$$\frac{\partial [A]}{\partial t} = 0,$$

where $[A]$ is the molar concentration of species A. If the set of QSS species is judiciously chosen, macroscopic quantities of interest such as ignition delay or laminar flame speed are only mildly affected by the QSS assumption [DRG2005].

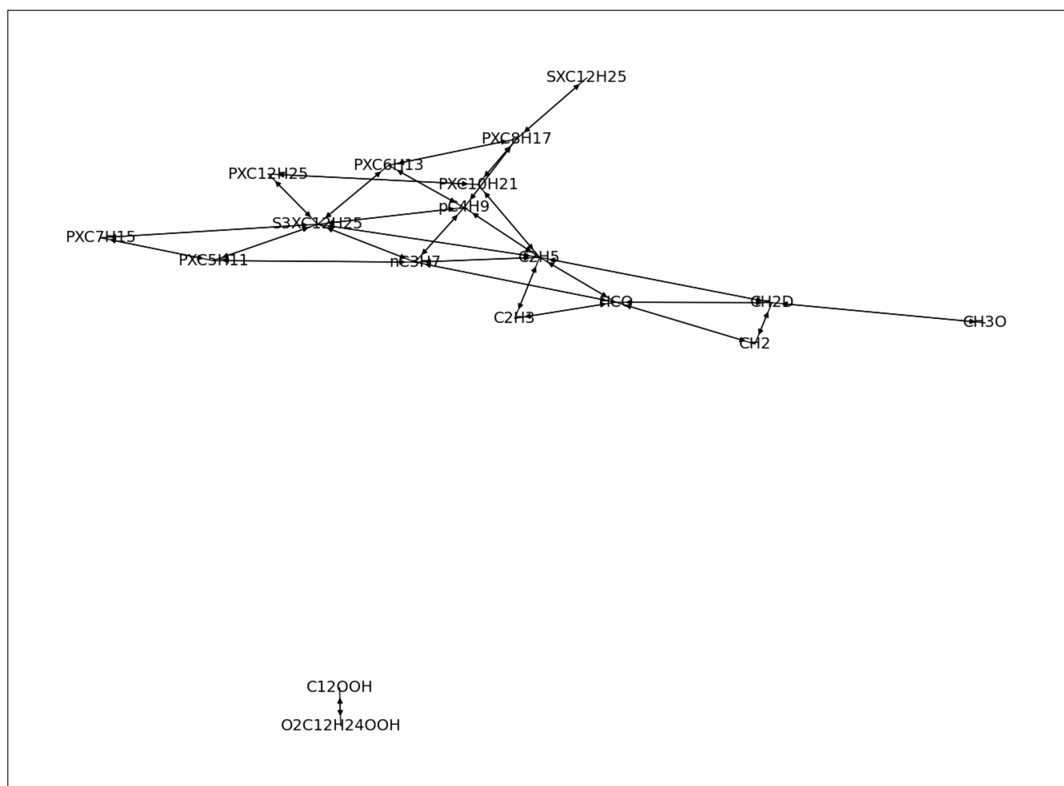
3.4.2 The advantage of the QSS assumption

Using the elementary reactions of the chemical mechanism, it is possible to write a set of algebraic equations that relate *QSS species* to *non-QSS species*. In turn, it is not necessary to transport the *QSS species* in the flow solver, as their concentration can be directly obtained from the *non-QSS species*.

In addition, *QSS species* are typically species that induce stiffness in the chemical mechanism since they evolve on scales different that the *non-QSS species*.

3.4.3 From *non-QSS species* to *QSS species* concentration

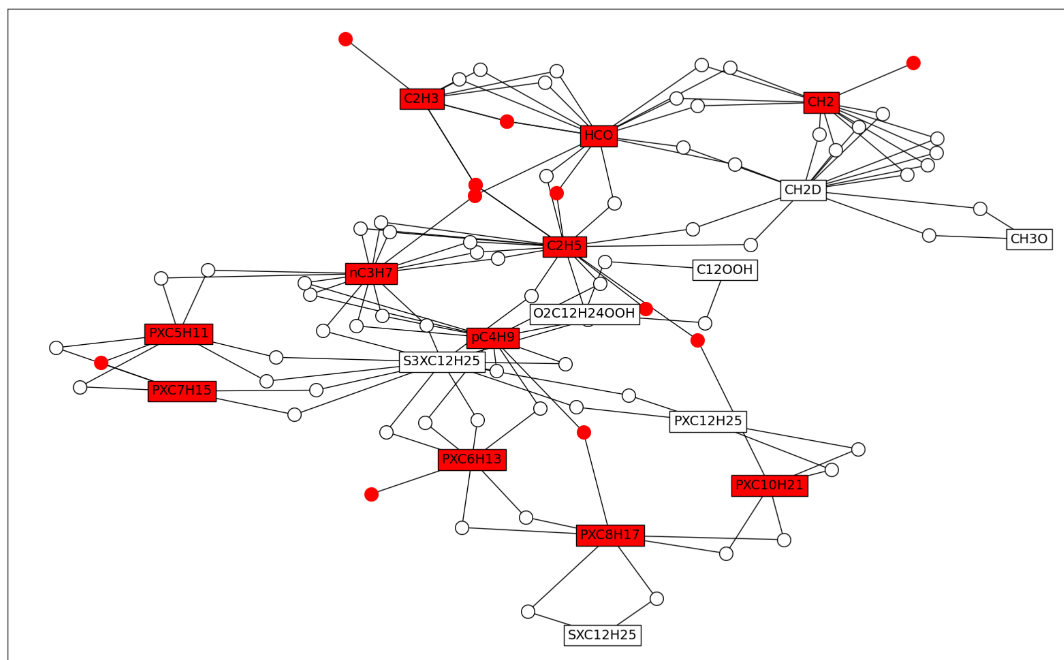
The set of algebraic equations that result from the QSS assumption does not always easily result in a simple algebraic relation between a given *QSS species* and all the *non-QSS species*. To do this, it is necessary to apply some form of linearization to the reactions involved [SYS2006]. In addition, even in the presence of linear relations, *QSS species* may depend on one another. The following figure shows the relation graph between the *QSS species* of a reduced $N - C_{12}H_{26}$ mechanism [ND2018]. The arrows indicate that a *QSS species* concentration depends on another *QSS species* concentration. It can also be seen that dependency groups exist among the *QSS species*. In *PelePhysics* it is possible to deduce invert analytically the linear system of equations given an arbitrary relation graph.



3.4: Relation graph between QSS species for N-dodecane mechanism [ND2018]

3.4.4 Linearizing the set of equations

In general, the QSS assumption results in a set of equations that are non-linearly coupled making it difficult to invert the system. The non-linear relations can arise if two or more *QSS-species* are on the same side of an elementary reaction, or if the stoichiometric coefficient of a *QSS-species* is not equal to one. Below, the relation graph between the QSS species plotted above is expanded with dots that denote reactions that relate *QSS-species*. Dots or species are colored in red if they are involved in a quadratic coupling.



3.5: Graph of dependencies between QSS species for N-dodecane mechanism [ND2018], augmented with reactions. Red species or dots denote species and reactions involved in quadratic coupling.

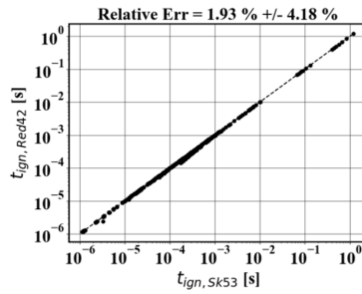
From here, it is necessary to eliminate the quadratic coupling to linearize the set of algebraic equations that result from the QSS assumption. Three methods can be envisioned: either one relaxes the QSS assumption by reducing the set of *QSS species* (Method 1) or one can eliminate the reactions that induce quadratic coupling. In case reactions are reversible, it can happen that either the forward or the backward reaction induces the quadratic coupling. Either one can remove both the forward and the backward reaction (Method 2) or remove either the backward or the forward reaction (Method 3).

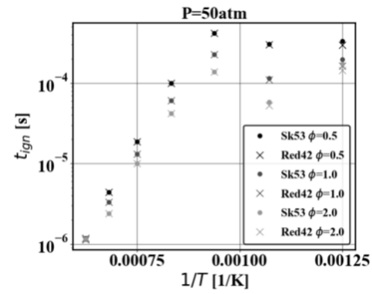
All three methods are available in *PelePhysics*. By default, Method 3 is activated as it is computationally efficient and accurate (as will be shown below). Method 1 is the most accurate and Method 2 is the most computationally efficient in our experience. Given that each method has its advantage, we decided to allow the user to choose either one according to his/her needs.

3.4.5 Validation

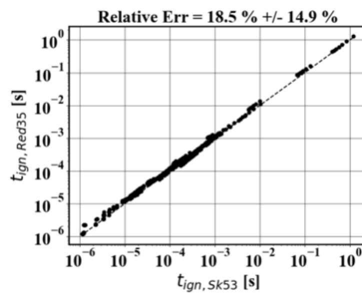
The three linearization methods are validated against the skeletal $N-C_{12}H_{26}$ [SKEL2017]. Using 343 0D calculation that span the range of applicability of the QSS assumption ($\phi = [0.5, 2.0], p = [1atm, 50atm], T = [800K, 1600K]$), the ignition delay is computed using the skeletal mechanism (SK53) and each one of the three linearization methods for the QSS mechanism (RedXX). The left plot shows the correlation between the ignition delay from the skeletal mechanism and the reduced version. The statistics of the relative error between the reduced and the skeletal mechanism are shown in the title of that plot. The right plot shows the ignition delay values at high-pressure conditions only.

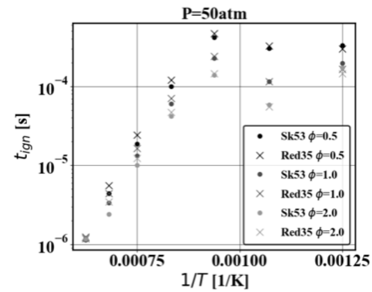
Method 1



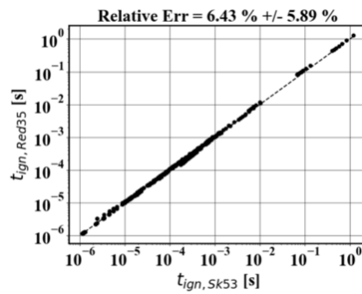


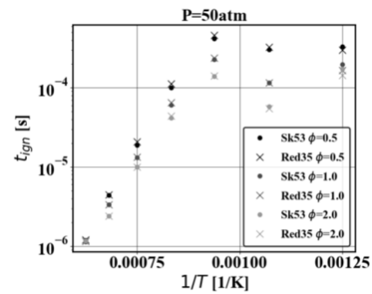
Method 2





Method 3

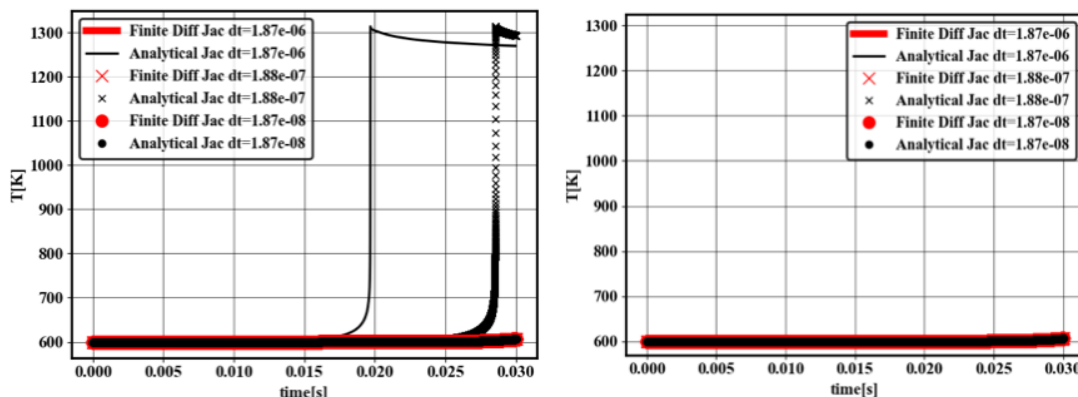




3.6: Left: Scatter plot of the ignition delay measured with the QSS mechanism linearized and the skeletal mechanism. Right: Ignition delays measured for the skeletal mechanism and QSS mechanism linearized at high-pressure conditions. Top: Method 1. Middle: Method 2. Bottom: Method 3.

3.4.6 Analytical Jacobian

In several computational experiments, using analytical Jacobians was found to provide better stability or efficiency compared with finite difference approximation or numerical inversion (see also *fig:qss_integrator*). Compared with non-QSS mechanisms, analytical Jacobians need to reflect the dependence of each QSS species on non-QSS species. However, QSS species may depend on an ensemble of other non-QSS species and therefore ensemble of reactions. Therefore, analytical Jacobian cannot be constructed by sequentially adding the contribution of each reaction. This significantly complicates the analytical jacobian generation. Failure to include the dependence of QSS species with respect to non-QSS species typically results in wrong ignition profiles, unless very small timesteps are used, as seen in figure *fig:qss_aj*.



3.7: Temperature of a 0D reactor at constant pressure for NC12H26. Initial temperature is 600K, initial molar fraction of O2 is 0.7 and initial molar fraction of fuel is 0.3. Left: Results without inclusion of dependence of QSS species with respect to non-QSS species. Right: Result with inclusion of dependence of QSS species with respect to non-QSS species.

To ease the implementation of analytical Jacobian in presence of QSS species, a symbolic approach is used to construct the analytical Jacobian. This strategy has the advantage of not requiring complex logic, and being flexible and readable for future development. For the last row of the Jacobian (partial difference of reaction rate with respect to temperature), finite difference is used since perturbations in temperature are less susceptible to numerical errors than perturbations in species concentrations. During the construction of the reaction rates, the operations printed to file are recorded symbolically using the `sympy` and `symengine` library [SYMPY]. For computational efficiency during the symbolic differentiation, the chain-rules terms are computed and the final expressions are computed and assembled by chain-ruling using logic internal to *CEPTR* rather than `sympy`. We have found that this speeds up the Jacobian construction cost by a factor 10.

To activate the use of symbolic jacobian, one needs to pass the flag `--qss_symbolic_jacobian` to *CEPTR*.

Printing the Jacobian terms one by one is not possible since the expressions that include QSS species are typically very large. Instead, the expressions are reduced via common sub-expression precomputing that are later used in each term of the Jacobian. The number of subexpressions may be orders of magnitude larger than the number of Jacobian entries which can be problematic if the computational architecture has limited memory.

Several formatting strategies have been implemented to mitigate the memory footprint of the symbolic Jacobian. They can be adjusted by providing a `.toml` file to *CEPTR* via the `qss_format_input` flag. A model input file is provided in the *Tutorials* section of this documentation. A model execution script for generating a mechanism for `dodecane_lu_qss` is available under the *Tutorials* section of this documentation.

The formatting options are the following

- **`hformat`** **(string)**

  - cpu will print intermediate variables used for chainruling. This gives a "readable" version of the Jacobian entries, albeit memory consuming.
  - gpu will not print intermediate variables used for chainruling, and instead will replace them directly in the Jacobian entries. This gives a less readable version of the Jacobian, but more memory efficient.

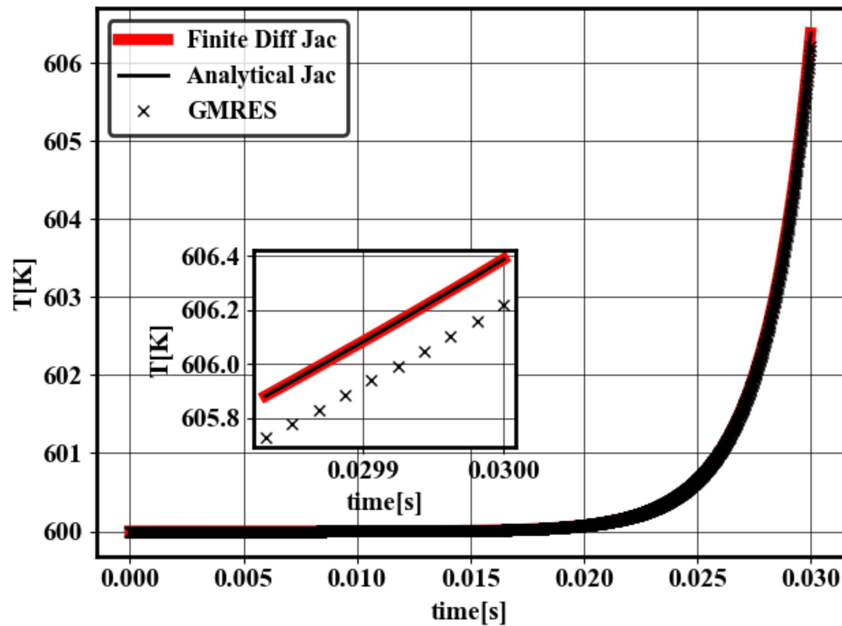
- `remove_1` **(boolean)** will replace expressions of the type `1.0 * xxx` into `xxx`.
- `remove_pow` **(boolean)** will convert expressions of the type `pow(xxx,n)` into multiplications or division. The conversion occurs for `n<=3` and `n>=-3` consistent with optimCuda
- `remove_pow10` **(boolean)** will convert expressions of the type `pow(10,xxx)` into `exp(ln(10)*xxx)`, consistent with optimCuda
- `min_op_count` **(integer)** counts number operations used to construct each common subexpression and replace the common subexpression if the number of operations is less or equal to `n`
- `min_op_count_all` **(integer)** is similar to `min_op_count` but also counts how many times that common subexpression is used later. The meaning of `n` is different than for `min_op_count` as it refers to how many more operations will be done if the common subexpression is eliminated. This option should be prefered to `min_op_count` as it tends to only marginally increase the file size (therefore compile time), while still being memory efficient.
- `gradual_op_count` **(boolean)** is useful if `min_op_count` or `min_op_count_all` are active. It loops from 1 to `n` and gradually eliminate the common subexpressions. This has the advantage of ensuring that the memory footprint is strictly monotonically decreasing as n is increased.
- `store_in_jacobian` **(boolean)** will use the Jacobian array as a temporary space to store intermediate variables. In particular, the last row of the Jacobian (dependence with respect to temperature) is done by finite difference which requires storing intermediate variables (production rate, forward and backward reactions). When the option is active, the `productionRate` function used to compute the finite difference is replaced with a `productionRate_light` functions where references to different parts of the Jacobian are used in place of allocating new arrays.
- `round_decimals` **(boolean)** will round floats printed by `sympy` when possible to minimize character count in the `mechanism.H` file.
- `recycle_cse` **(boolean)** will reuse subexpressions that are not used later to avoid declaring new temporary reals.
- `remove_single_symbols_cse` **(boolean)** will remove common subexpressions that are made of 1 operation and 1 symbol. Those common subexpressions are typically `-xxx` and may not appear as worth replacing because they save 1 operations and are reused multiple times. However, when replaced in the later expressions, the `-` operations typically disappear or is merged into another operations which actually does not increase the total number of operations.

The analytical Jacobian for QSS mechanisms is typically more accurate and stable than GMRES, and is on par with the finite difference Jacobian of *CVODE* as seen in *fig:qss_integrator*

In terms of speed, the analytical Jacobian 0D reactor is faster on CPU than finite difference Jacobian and GMRES. For the piston bowl challenge problem, the analytical Jacobian relative speed depends on the prevalence of chemical reactions. At some points, the AJ is slower than GMRES with PeleC, at others, AJ is faster. In PeleLM cases, AJ was found to be faster than GMRES. Further optimization and tests are still ongoing.

3.4.7 Using the Analytical Jacobian

To use the analytic Jacobian QSS mechanisms in a Pele run, one must specify the correct `solve_type`. The choice of the `solve_type` will determine whether or not the analytic Jacobian is used. With CUDA, both `magma_direct` and `sparse_direct` will use the analytic Jacobian, while on HIP only `magma_direct` will trigger the use of the analytic Jacobian parts.



3.8: Temperature of a 0D reactor at constant pressure for NC12H26. Initial temperature is 600K, initial molar fraction of O2 is 0.7 and initial molar fraction of fuel is 0.3. Results are shown for finite difference jacobian (red thick line), analytical jacobian (black line) and GMRES (crosses) using the same tolerances.

3.5 CEPTR: Chemistry Evaluation for Pele Through Recasting

We use CEPTR to generate C++ mechanism files from Cantera yaml chemistry files. CEPTR is a python package part of the PelePhysics source code.

3.5.1 Software requirements

The CEPTR package uses poetry to manage the Python dependencies. Poetry is therefore required to use CEPTR and can typically be installed through system package managers (e.g. HomeBrew) or following the instructions in poetry's documentation.

To install CEPTR dependencies:

```
$ cd ${PELE_PHYSICS_HOME}/Support/ceptr
$ poetry update
```

3.5.2 Usage

There are three ways to use CEPTR to generate C++ mechanism files for a given chemistry

1. Using CEPTR directly:

```
$ cd ${PELE_PHYSICS_HOME}/Support/ceptr
$ poetry run convert -f ${PELE_PHYSICS_HOME}/Support/Mechanism/Models/LiDryer/
↪mechanism.yaml
```

2. Using a helper script in the directory containing the `mechanism.yaml` file:

```
$ ./convert.sh
```

3. Using a helper script in the `Models` directory:

```
$ bash ${PELE_PHYSICS_HOME}/Support/Mechanism/Models/converter.sh -f ./LiDryer/
↪mechanism.yaml
```

For non-reduced chemistries, CEPTR can take a file with a list of `mechanism.yaml` files to convert:

```
$ cd ${PELE_PHYSICS_HOME}/Support/ceptr
$ poetry run convert -l ${PELE_PHYSICS_HOME}/Support/Mechanism/Models/list_mech
```

For reduced chemistries, CEPTR can take a file with a list of `qssa.yaml` and `qssa_input.toml` to convert:

```
$ cd ${PELE_PHYSICS_HOME}/Support/ceptr
$ poetry run convert -lq ${PELE_PHYSICS_HOME}/Support/Mechanism/Models/list_qss_mech
```

3.5.3 Converting CHEMKIN files

We rely on Cantera's `ck2yaml` utility to convert CHEMKIN files to the Cantera yaml format (and proceed as above with CEPTR on the resulting yaml file):

```
$ cd ${PELE_PHYSICS_HOME}/Support/ceptr
$ poetry run ck2yaml --input=${PATH_TO_CHEMKIN_DIR}/mechanism.inp --thermo=${PATH_TO_
↪CHEMKIN_DIR}/therm.dat --transport=${PATH_TO_CHEMKIN_DIR}/tran.dat --permissive
```

The files `tran.dat` and `therm.dat` are optional if already included in the `.inp` file.

3.5.4 Generating a QSS chemistry file

To generate a QSS chemistry yaml file from another yaml file, one executes:

```
$ poetry run qssa -f ${PATH_TO_YAML}/skeletal.yaml -n ${PATH_TO_YAML}/non_qssa_list.
↪yaml
```

The full list of options is:

```
$ poetry run qssa -h
usage: qssa [-h] -f FNAME -n NQSSA [-m {0,1,2}] [-v]

Mechanism converter

optional arguments:
  -h, --help            show this help message and exit
  -f FNAME, --fname FNAME
                        Mechanism file
  -n NQSSA, --nqssa NQSSA
                        Non-QSSA species list
  -m {0,1,2}, --method {0,1,2}
                        QSSA method (default: 2)
  -v, --visualize       Visualize quadratic coupling and QSSA dependencies
```

For a detailed description of these options and a further information on the way QSS mechanism are treated in *CEPTR* the reader may consult *the QSS section*.

See Tutorials (*Generating NC12H26 QSS mechanism with analytical jacobian* and *Generating NC12H26 QSS mechanism without analytical jacobian*) for generating QSS mechanisms from the `.yaml` files.

CHAPTER 4

Transport

Note: Placeholder, to be written

CHAPTER 5

Equation of State

PelePhysics allows the user to use different equation of state (EOS) as the constitutive equation and close the compressible Navier-Stokes system of equations. All the routines needed to fully define an EOS are implemented through PelePhysics module. Available models include:

- An ideal gas mixture model (similar to the CHEMKIN-II approach)
- A simple *GammaLaw* model
- Cubic models such as *Soave-Redlich-Kwong*; *Peng-Robinson* support was started but is currently stalled.

Examples of EOS implementation can be seen in *PelePhysics/Eos*. The following sections will fully describe the implementation of Soave-Redlich-Kwong, a non-ideal cubic EOS, for a general mixture of species. Integration with CEPTR, for a chemical mechanism described in a chemkin format, will also be highlighted. For an advanced user interested in implementing a new EOS this chapter should provide a good starting point.

5.1 Soave-Redlich-Kwong (SRK)

The cubic model is built on top of the ideal gas models, and is selected by specifying its name as the *Eos_dir* during the build (the *Chemistry_Model* must also be specified). Any additional parameters (e.g., attractions, repulsions, critical states) are either included in the underlying FUEGO database used to generate the source file model implementation, or else are inferred from the input model data.

SRK EOS as a function of Pressure (p), Temperature(T), and τ (specific volume) is given by

$$p = RT \sum \frac{Y_k}{W_k} \frac{1}{\tau - b_m} - \frac{a_m}{\tau(\tau + b_m)}$$

where Y_k are species mass fractions, R is the universal gas constant, and b_m and a_m are mixture repulsion and attraction terms, respectively.

5.1.1 Mixing rules

For a mixture of species, the following mixing rules are used to compute b_m and a_m.

$$a_m = \sum_{ij} Y_i Y_j \alpha_i \alpha_j \quad b_m = \sum_k Y_k b_k$$

where b_i and a_i for each species is defined using critical pressure and temperature.

$$a_i(T) = 0.42748 \frac{(RT_{c,i})^2}{W_i^2 p_{c,i}} \bar{a}_i \left(T/T_{c,i}\right) \quad b_i = 0.08664 \frac{RT_{c,i}}{W_i p_{c,i}}$$

where

$$\bar{a}_i(T/T_{c,i}) = \left(1 + \mathcal{A}\left[f\left(\omega_i\right)\left(1 - \sqrt{T/T_{c,i}}\right)\right]\right)^2$$

where ω_i are the accentric factors and

$$f\left(\omega_i\right) = 0.48508 + 1.5517\omega_i - 0.151613\omega_i^2$$

For chemically unstable species such as radicals, critical temperatures and pressures are not available. For species where critical properties are not available, we use the Lennard-Jones potential for that species to construct attractive and repulsive coefficients.

$$T_{c,i} = 1.316 \frac{\epsilon_i}{k_b} \quad a_i(T_{c,i}) = 5.55 \frac{\epsilon_i \sigma_i^3}{m_i^2} \quad \text{and} \quad b_i = 0.855 \frac{\sigma_i^3}{m_i}$$

where σ_i, ϵ_i are the Lennard-Jones potential molecular diameter and well-depth, respectively, m_i the molecular mass, and k_b is Boltzmann's constant.

In terms of implementation, a routine called *MixingRuleAmBm* can be found in the SRK eos implementation. The following code block shows the subroutine which receives species mass fractions and temperature as input. The outputs of this routine are b_m and a_m .

```
do i = 1, nspecies
  Tr = T*oneOverTc(i)
  amloc(i) =  (1.0d0 + Fomega(i)*(1.0d0-sqrt(Tr))) *sqrtAsti(i)

  bm = bm + massFrac(i)*Bi(i)

enddo
do j = 1, nspecies
   do i = 1, nspecies

      am = am + massFrac(i)*massFrac(j)*amloc(i)*amloc(j)

   end do
end do
```

5.1.2 Thermodynamic Properties

Most of the thermodynamic properties can be calculated from the equation of state and involve derivatives of various thermodynamic quantities and of EOS parameters. In the following, some of these thermodynamic properties for SRK and the corresponding routines are presented.

Specific heat

For computing mixture specific heat at constant volume and pressure, the ideal gas contribution and the departure from the ideal gas are computed. Specific heat at constant volume can be computed using the following

$$c_v = \left(\frac{\partial e_m}{\partial T}\right)_{\tau,Y}$$

For SRK EOS, the formula for c_v reduces to

$$c_v = c_v^{id} - T\frac{\partial^2 a_m}{\partial T^2}\frac{1}{b_m}ln(1+\frac{b_m}{\tau})$$

where c_v^{id} is the specific heat at constant volume. Mixture specific heat at constant volume is implemented through the routine *SRK_EOS_GetMixtureCv*

```
subroutine SRK_EOS_GetMixtureCv(state)
implicit none
type (eos_t), intent(inout) :: state
real(amrex_real) :: tau, K1

state % wbar = 1.d0 / sum(state % massfrac(:) * inv_mwt(:))

call MixingRuleAmBm(state%T,state%massFrac,state%am,state%bm)

tau = 1.0d0/state%rho

! Derivative of the EOS AM w.r.t Temperature - needed for calculating enthalpy, Cp,␣
↪Cv and internal energy
call Calc_dAmdT(state%T,state%massFrac,state%am,state%dAmdT)

! Second Derivative of the EOS AM w.r.t Temperature - needed for calculating enthalpy,
↪ Cp, Cv and internal energy
call Calc_d2AmdT2(state%T,state%massFrac,state%d2AmdT2)

! Ideal gas specific heat at constant volume
call ckcvbs(state%T, state % massfrac, iwrk, rwrk, state % cv)

! Real gas specific heat at constant volume
state%cv = state%cv + state%T*state%d2AmdT2* (1.0d0/state%bm)*log(1.0d0+state%bm/tau)

end subroutine SRK_EOS_GetMixtureCv
```

Specific heat at constant pressure is given by

$$c_p = \left(\frac{\partial h_m}{\partial T}\right)_{p,Y}$$

$$c_p = \frac{\partial h_m}{\partial T} - \frac{\frac{\partial h}{\partial \tau}}{\frac{\partial p}{\partial \tau}}\frac{\partial p}{\partial T}$$

where all the derivatives in the above expression for SRK EOS are given by

$$\frac{\partial p}{\partial T} = \sum Y_k/W_k \frac{R}{\tau - b_m} - \frac{\partial a_m}{\partial T}\frac{1}{\tau(\tau + b_m)}$$

$$\frac{\partial p}{\partial \tau} = -\sum Y_k/W_k \frac{RT}{(\tau - b_m)^2} + \frac{a_m(2\tau + b_m)}{[\tau(\tau + b_m)]^2}$$

$$\frac{\partial h_m}{\partial \tau} = -\left(T\frac{\partial a_m}{\partial T} - a_m\right)\frac{1}{\tau(\tau + b_m)} + \frac{a_m}{(\tau + b_m)^2} - \sum Y_k/W_k \frac{RTb_m}{(\tau - b_m)^2}$$

$$\frac{\partial h_m}{\partial T} = c_p^{id} + T\frac{\partial^2 a_m}{\partial T^2}\frac{1}{b_m} ln(1 + \frac{b_m}{\tau}) - \frac{\partial a_m}{\partial T}\frac{1}{\tau + b_m} + \sum Y_k/W_k \frac{Rb_m}{\tau - b_m}$$

```
subroutine SRK_EOS_GetMixtureCp(state)
implicit none
type (eos_t), intent(inout) :: state
real(amrex_real) :: tau, K1
real(amrex_real) :var: : Cpig
real(amrex_real) :: eosT1Denom, eosT2Denom, eosT3Denom
real(amrex_real) :: InvEosT1Denom,InvEosT2Denom,InvEosT3Denom
real(amrex_real) :: dhmdT,dhmdtau
real(amrex_real) :: Rm

state % wbar = 1.d0 / sum(state % massfrac(:) * inv_mwt(:))

call MixingRuleAmBm(state%T,state%massFrac,state%am,state%bm)

tau = 1.0d0/state%rho

! Derivative of the EOS AM w.r.t Temperature - needed for calculating enthalpy, Cp,␣
↪Cv and internal energy
call Calc_dAmdT(state%T,state%massFrac,state%dAmdT)

! Second Derivative of the EOS AM w.r.t Temperature - needed for calculating enthalpy,
↪ Cp, Cv and internal energy
call Calc_d2AmdT2(state%T,state%massFrac,state%d2AmdT2)

K1 = (1.0d0/state%bm)*log(1.0d0+state%bm/tau)

eosT1Denom = tau-state%bm
eosT2Denom = tau*(tau+state%bm)
eosT3Denom = tau+state%bm

InvEosT1Denom = 1.0d0/eosT1Denom
InvEosT2Denom = 1.0d0/eosT2Denom
InvEosT3Denom = 1.0d0/eosT3Denom

Rm = (Ru/state%wbar)

! Derivative of Pressure w.r.t to Temperature
state%dPdT = Rm*InvEosT1Denom - state%dAmdT*InvEosT2Denom

! Derivative of Pressure w.r.t to tau (specific volume)
state%dpdtau = -Rm*state%T*InvEosT1Denom*InvEosT1Denom + state%am*(2.0*tau+state
↪%bm)*InvEosT2Denom*InvEosT2Denom

! Ideal gas specific heat at constant pressure
call ckcpbs(state % T, state % massfrac, iwrk, rwrk,Cpig)
```

(continues on next page)

(continued from previous page)

```
! Derivative of enthalpy w.r.t to Temperature
dhmdT = Cpig + state%T*state%d2AmdT2*K1 - state%dAmdT*InvEosT3Denom + Rm*state
↪%bm*InvEosT1Denom

! Derivative of enthalpy w.r.t to tau (specific volume)
dhmdtau = -(state%T*state%dAmdT - state%am)*InvEosT2Denom + state
↪%am*InvEosT3Denom*InvEosT3Denom - &
   Rm*state%T*state%bm*InvEosT1Denom*InvEosT1Denom

! Real gas specific heat at constant pressure
state%cp = dhmdT - (dhmdtau/state%dpdtau)*state%dPdT

end subroutine SRK_EOS_GetMixtureCp
```

Internal energy and Enthalpy

Similarly mixture internal energy for SRK EOS is given by

$$e_m = \sum_k Y_k e_k^{id} + \left(T\frac{\partial a_m}{\partial T} - a_m \right) \frac{1}{b_m} ln \left(1 + \frac{b_m}{\tau} \right)$$

and mixture enthalpy $h_m = e + p\tau$

$$h_m = \sum_k Y_k h_k^{id} + \left(T\frac{\partial a_m}{\partial T} - a_m \right) \frac{1}{b_m} \ln \left(1 + \frac{b_m}{\tau} \right) + RT \sum \frac{Y_k}{W_k} \frac{b_m}{\tau - b_m} - \frac{a_m}{\tau + b_m}$$

and the implementation can be found in the routine *SRK_EOS_GetMixture_H*.

Speed of Sound

The sound speed for SRK EOS is given by

$$a^2 = -\frac{c_p}{c_v} \tau^2 \frac{\partial p}{\partial \tau}$$

Species enthalpy

For computation of kinetics and transport fluxes we will also need the species partial enthalpies and the chemical potential. The species enthalpies for SRK EOS are given by

$$h_k = \frac{\partial h_m}{\partial Y_k} - \frac{\frac{\partial h}{\partial \tau}}{\frac{\partial p}{\partial \tau}} \frac{\partial p}{\partial Y_k}$$

where

$$\begin{aligned} \frac{\partial h_m}{\partial Y_k} = h_k^{id} &+ (T\frac{\partial^2 a_m}{\partial T \partial Y_k} - \frac{\partial a_m}{\partial Y_k}) \frac{1}{b_m} \ln \left(1 + \frac{b_m}{\tau} \right) \\ &- \left(T\frac{\partial a_m}{\partial T} - a_m \right) \left[\frac{1}{b_m^2} \ln \left(1 + \frac{b_m}{\tau} \right) - \frac{1}{b_m(\tau + b_m)} \right] \frac{\partial b_m}{\partial Y_k} \\ &+ \frac{a_m}{(\tau + b_m)^2} \frac{\partial b_m}{\partial Y_k} - \frac{1}{\tau + b_m} \frac{\partial a_m}{\partial Y_k} + 1/W_k \frac{RT b_m}{\tau - b_m} \\ &+ \sum_i \frac{Y_i}{W_i} RT \left(\frac{1}{\tau - b_m} + \frac{b_m}{(\tau - b_m)^2} \right) \frac{\partial b_m}{\partial Y_k} \end{aligned}$$

$$\frac{\partial p}{\partial Y_k} = RT\frac{1}{W_k}\frac{1}{\tau - b_m} - \frac{\partial a_m}{\partial Y_k}\frac{1}{\tau(\tau + b_m)} + \left(RT\sum\frac{Y_i}{W_i}\frac{1}{(\tau - b_m)^2} + \frac{a_m}{\tau(\tau + b_m)^2}\right)\frac{\partial b_m}{\partial Y_k}$$

Chemical potential

The chemical potentials are the derivative of the free energy with respect to composition. Here the free energy f‘ is given by

$$\begin{aligned} f &= \sum_i Y_i(e_i^{id} - Ts_i^{id,*}) + \sum_i \frac{Y_iRT}{W_i}ln(\frac{Y_iRT}{W_i\tau p^{st}}) \\ &+ \sum_i \frac{Y_iRT}{W_i}ln(\frac{\tau}{\tau - b_m}) - a_m\frac{1}{b_m}ln(1 + \frac{b_m}{\tau}) \\ &= \sum_i Y_i(e_i^{id} - Ts_i^{id,*}) + \sum_i \frac{Y_iRT}{W_i}ln(\frac{Y_iRT}{W_i(\tau - b_m)p^{st}}) - a_m\frac{1}{b_m}ln(1 + \frac{b_m}{\tau}) \end{aligned}$$

Then

$$\begin{aligned} \mu_k = \frac{\partial f}{\partial Y_k} &= e_k^{id} - Ts_k^{id,*} + \frac{RT}{W_k}ln(\frac{Y_kRT}{W_k(\tau - b_m)p^{st}}) + \frac{RT}{W_k} + \frac{RT}{\bar{W}}\frac{1}{\tau - b_m}\frac{\partial b_m}{\partial Y_k} \\ &- \frac{1}{b_m}ln(1 + \frac{b_m}{\tau})\frac{\partial a_m}{\partial Y_k} + \frac{a_m}{b_m^2}ln(1 + \frac{b_m}{\tau})\frac{\partial b_m}{\partial Y_k} - \frac{a_m}{b_m}\frac{1}{\tau + b_m}\frac{\partial b_m}{\partial Y_k} \end{aligned}$$

Other primitive variable derivatives

The Godunov (FV) algorithm also needs some derivatives to express source terms in terms of primitive variables. In particular one needs

$$\left.\frac{\partial p}{\partial \rho}\right|_{e,Y} = -\tau^2\left(\frac{\partial p}{\partial \tau} - \frac{\frac{\partial e}{\partial \tau}}{\frac{\partial e}{\partial T}}\frac{\partial p}{\partial T}\right)$$

and

$$\left.\frac{\partial p}{\partial e}\right|_{\rho,Y} = \frac{1}{c_v}\frac{\partial p}{\partial T}$$

All of the terms needed to evaluate this quantity are known except for

$$\frac{\partial e}{\partial \tau} = \frac{1}{\tau(\tau + b_m)}\left(a_m - T\frac{\partial a_m}{\partial T}\right) \quad .$$

CHAPTER 6

Tutorials

6.1 Tutorial 1 - Generating NC12H26 QSS mechanism with analytical Jacobian

Update `poetry`:

```
$ cd ${PELE_PHYSICS_HOME}/Support/ceptr
$ poetry update
```

Make sure the list of non-QSS species is correct in `${PELE_PHYSICS_HOME}/Support/Mechanism/Models/dodecane_lu_qss/non_qssa_list.yaml`

In the next step, `skeletal.yaml` is the `mechanism.yaml` of the skeletal version of the mechanism (here available under `${PELE_PHYSICS_HOME}/Support/Mechanism/Models/dodecane_lu`). If only CHEMKIN files are available for the skeletal mechanism, see *Converting CHEMKIN files* for generating `skeletal.yaml`.

Generate `qssa.yaml` from `skeletal.yaml` and `non_qssa_list.yaml`:

```
$ cd ${PELE_PHYSICS_HOME}/Support/ceptr
$ poetry run qssa -f ${PELE_PHYSICS_HOME}/Support/Mechanism/Models/dodecane_lu_qss/
↪skeletal.yaml -n ${PELE_PHYSICS_HOME}/Support/Mechanism/Models/dodecane_lu_qss/non_
↪qssa_list.yaml
```

Generate `mechanism.H` and `mechanism.cpp` from `qssa.yaml`:

```
$ cd ${PELE_PHYSICS_HOME}/Support/ceptr
$ poetry run convert -f ${PELE_PHYSICS_HOME}/Support/Mechanism/Models/dodecane_lu_qss/
↪qssa.yaml --qss_format_input ${PELE_PHYSICS_HOME}/Support/Mechanism/Models/dodecane_
↪lu_qss/qssa_input.toml --qss_symbolic_jacobian
```

We recommend using the following `qssa_input.toml`:

```
$ [Readability]
  hformat =                       "gpu"
```

(continues on next page)

(continued from previous page)

```
[Arithmetic]
remove_1 =                     true
remove_pow =                   true
remove_pow10 =                 true

[Replacement]
min_op_count =                 0
min_op_count_all =             10
gradual_op_count =             true
remove_single_symbols_cse =    true

[Recycle]
store_in_jacobian =            true
recycle_cse =                  true

[Characters]
round_decimals =               true
```

6.2 Tutorial 2 - Generating NC12H26 QSS mechanism without analytical Jacobian

Update `poetry`:

```
$ cd ${PELE_PHYSICS_HOME}/Support/ceptr
$ poetry update
```

Make sure the list of non-QSS species is correct in `Support/Mechanism/Models/dodecane_lu_qss/non_qssa_list.yaml`

In the next step, `skeletal.yaml` is the `mechanism.yaml` of the skeletal version of the mechanism (here available under `${PELE_PHYSICS_HOME}/Support/Mechanism/Models/dodecane_lu`). If only CHEMKIN files are available for the skeletal mechanism, see *Converting CHEMKIN files* for generating `skeletal.yaml`.

Generate `qssa.yaml` from `skeletal.yaml` and `non_qssa_list.yaml`:

```
$ cd ${PELE_PHYSICS_HOME}/Support/ceptr
$ poetry run qssa -f ${PELE_PHYSICS_HOME}/Support/Mechanism/Models/dodecane_lu_qss/
↪skeletal.yaml -n ${PELE_PHYSICS_HOME}/Support/Mechanism/Models/dodecane_lu_qss/non_
↪qssa_list.yaml
```

Generate `mechanism.H` and `mechanism.cpp` from `qssa.yaml`:

```
$ cd ${PELE_PHYSICS_HOME}/Support/ceptr
$ poetry run convert -f ${PELE_PHYSICS_HOME}/Support/Mechanism/Models/dodecane_lu_qss/
↪qssa.yaml
```

6.3 Tutorial 3 - Generating NC12H26 Skeletal mechanism

Update `poetry`:

```
$ cd ${PELE_PHYSICS_HOME}/Support/ceptr
$ poetry update
```

For all the available mechanisms, a Cantera yaml format is provided. If only CHEMKIN files are available, see *Converting CHEMKIN files* for generating `mechanism.yaml`.

Generate `mechanism.H` and `mechanism.cpp` from `mechanism.yaml`:

```
$ cd ${PELE_PHYSICS_HOME}/Support/ceptr
$ poetry run convert -f ${PELE_PHYSICS_HOME}/Support/Mechanism/Models/dodecane_lu/
↪mechanism.yaml
```

CHAPTER 7

Developer Guidelines

The following developer guidelines apply to all code that is to be committed to the *Pele* suite.

Before adding files for a commit to be pushed to a branch of *PelePhysics*, first run the following formatting tools depending on the type of code:

7.1 C++ Code

All C++ code should be processed by the Clang formatter prior to being added for commit.

Run `clang-format`:

```
clang-format -i FILE.cpp
clang-format -i FILE.H
```

Will apply all of the correct formatting and make the changes directly to the cpp source files.

7.2 Python

The tools necessary to format Python code (currently only used within CEPTR) are maintained through Poetry.

1) Run `isort`:

```
poetry run isort .
```

This will perform proper sorting of the installed Python libraries.

2) Run `black`:

```
poetry run black .
```

This will perform proper formatting of all Python files to be consistent with all current files.

3) Run `flake8`:

```
poetry run flake8 .
```

This will run diagnostics on all the Python files and will list a series of issues that need to be addressed to adhere to current Python best practices.

Once all `flake8` messages have been addressed, the code will match the *Pele* suite standard.

CHAPTER 8

Indices and tables

- genindex
- search

Bibliography

[LLNL2005] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic-equation solvers. ACM Transactions on Mathematical Software (TOMS), 31(3):363-396, 2005.

[BYRNE1975] G. D. Byrne, A. C. Hindmarsh. A polyalgorithm for the numerical solution of ordinary differential equations. ACM Transactions on Mathematical Software (TOMS), 1(1):71-96, 1975.

[JAC1980] K. R Jackson and R. Sacks-Davis. An alternative implementation of variable step-size multistep formulas for stiff odes. ACM Transactions on Mathematical Software (TOMS), 6(3):295–318, 1980.

[BROWN1990] P. N. Brown and Y. Saad. Hybrid krylov methods for nonlinear systems of equations. SIAM Journal on Scientific and Statistical Computing, 11(3):450–481, 1990.

[McNenly2015] M. J. McNenly, R. A. Whitesides, and D. L. Flowers. Faster solvers for large kinetic mechanisms using adaptive preconditioners. Proceedings of the Combustion Institute, 35(1):581–587, 2015.

[VODE1989] P. N. Brown, G. D. Byrne, and A. C. Hindmarsh. VODE, a variable-coefficient ODE solver. SIAM journal on scientific and statistical computing, 10(5):1038-1051, 1989.

[DRG2005] T. Lu, C. K. Law, A directed relation graph method for mechanism reduction, Proceedings of the combustion institute, 30(1):1333-1341, 2005.

[SYS2006] T. Lu, C. K. Law, Systematic approach to obtain analytic solutions of quasi steady state species in reduced mechanisms, The Journal of Physical Chemistry A, 110(49):13202-13208, 2006.

[ND2018] G. Borghesi, A. Krisman, T. Lu, J. H. Chen, Direct numerical simulation of a temporally evolving air/n-dodecane jet at low-temperature diesel-relevant conditions, 195:183-202, 2018.

[SKEL2017] T. Yao, Y. Pei, B. J. Zhong, S. Som, T. Lu, K. H. Luo, A compact skeletal mechanism for n-dodecane with optimized semi-global ! low-temperature chemistry for diesel engine simulations, 191:339-349, 2017.

[SYMPY] A. Meurer, C. P. Smith, M. Paprocki, O. v{C}ert'{i}k, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, S. Rouv{c}ka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, A. Scopatz, SymPy: symbolic computing in Python, 3:e103, 2017.