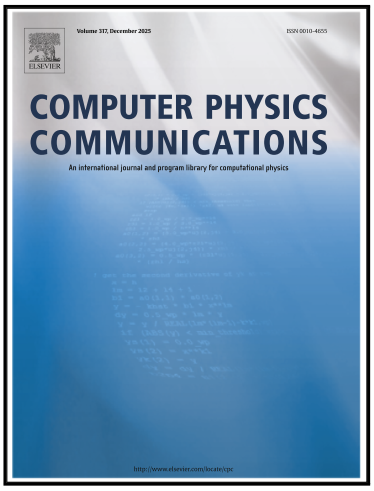
Editor: Prof. Andrew Hazel



Pyrometheus: Symbolic abstractions for XPU and automatically differentiated computation of combustion kinetics and thermodynamics

Esteban Cisneros-Garibay, Henry Le Berre, Dimitrios Adam, Spencer H. Bryngelson, Jonathan B. Freund

PII: S0010-4655(25)00488-6
DOI: https://doi.org/10.1016/j.cpc.2025.109987
Reference: COMPHY 109987

To appear in: *Computer Physics Communications*

Received date: 2 May 2025
Revised date: 7 October 2025
Accepted date: 5 December 2025

Pyrometheus: Symbolic abstractions for XPU and automatically differentiated computation of combustion kinetics and thermodynamics

Esteban Cisneros-Garibay[a,*], Henry Le Berre[b], Dimitrios Adam[b], Spencer H. Bryngelson[b,c,d], Jonathan B. Freund[e]

[a] *Mechanical Science & Engineering, University of Illinois Urbana–Champaign, Urbana, IL, USA*
[b] *School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA, USA*
[c] *Daniel Guggenheim School of Aerospace Engineering, Georgia Institute of Technology, Atlanta, GA, USA*
[d] *George W. Woodruff School of Mechanical Engineering, Georgia Institute of Technology, Atlanta, GA, USA*
[e] *Aerospace Engineering, University of Illinois Urbana–Champaign, Urbana, IL, USA*

Abstract

The cost of combustion simulations is often dominated by the evaluation of net production rates of chemical species and mixture thermodynamics (thermochemistry). Execution on computing accelerators (XPUs) such as graphics processing units (GPUs) can greatly reduce this cost. Established thermochemistry software is not readily portable to such devices, as it sacrifices valuable analytical forms that enable differentiation, sensitivity analysis, and implicit time integration. Symbolic abstractions are developed with corresponding transformations that enable computation on accelerators and automatic differentiation by avoiding premature specification of detail. The software package Pyrometheus is introduced as an implementation of these abstractions and their transformations for combustion thermochemistry. The formulation facilitates code generation from the symbolic representation of a specific thermochemical mechanism in multiple target languages, including Python, C++, and Fortran. The generated code processes array-valued expressions, but does not specify their semantics. The semantics are provided by compatible array libraries, including NumPy, Pytato, and Google JAX. Thus, the generated code retains a symbolic representation of the thermochemistry, which enables computation on accelerators and CPUs and facilitates automatic differentiation. The design and operation of the symbolic abstractions and their companion tool, Pyrometheus, are discussed throughout. Roofline demonstrations show that the computation of chemical source terms within MFC, a Fortran-based flow solver we link to Pyrometheus, is performant.

1. Motivation and Significance

1.1. Computational Challenges of Combustion Thermochemistry

Combustion underpins modern transportation and energy production, so performant tools are required to analysis and prediction. Simulations of reacting flows promise to improve combustion, particularly when experiments are costly or challenging [1]. The governing equations for reacting flows involve nonlinear chemical source terms and equations of state, which must be evaluated in such simulations. Depending on the application, implementations must satisfy different computational and analytical properties [2]. The present work introduces a computational strategy and accompanying tool that enable and accelerate high-fidelity, large-scale predictive simulation, preserving key capabilities. We focus on two capabilities: efficient computation on accelerator-based

[*]Corresponding author, presently at the University of Tennessee Space Institute
Email address: `ecisnero@utsi.edu` (Esteban Cisneros-Garibay)
Code available at: `https://github.com/pyrometheus` and `https://github.com/MFlowCode`

exascale platforms and exact differentiation for sensitivity analysis. Existing approaches to computational thermochemistry provide, at most, only one such capability. Herein, we expand on the utility of these capabilities and the challenges they present.

The evaluation of thermochemistry often significantly adds to the cost of simulating reacting flows. This cost primarily follows three sources. The first is an increase in the number of transported variables compared to simulations of inert flow; the mixture composition can be described by up to hundreds of reacting scalars, each accompanied by its own partial differential equation (PDE). The second source is the evaluation of thousands of exponentials that appear in expressions representing reaction rates. Lastly, there is a time-step restriction that increases the solution time and stems from the imposed chemical time scale, which is over 1000 times smaller than the flow time scale [3]. These effects compound, making a broad range of engineering-scale reacting flow simulations prohibitively expensive. Compute accelerators (XPUs), including graphic processing units (GPUs) and accelerated processing units (APUs, also known as superchips), can help mitigate this cost.

Our approach complements established thermochemistry software, such as dedicated libraries like Cantera [4], Chemkin [5], and TChem [6], which provide broad modeling capabilities but are limited to CPU hardware. While serving the community well, these are not easily optimized for high-performance simulation, especially on accelerator hardware. This limitation follows from their pre-compilation, wherein they serve as standalone executables separate from the flow solver. Due to their breadth and structure, adapting such libraries to GPUs is a challenging task that risks conflicting with their original goals. Establishing thermochemistry software for large-scale flow simulations on accelerators risks sacrificing the modeling and symbolic capabilities that libraries provide, or may encounter limitations as compute hardware evolves. To avoid this problem, we develop and implement an alternative approach to thermochemistry computation based on symbolic abstractions. From these abstractions, performant architecture-specific code is generated.

Our approach enables computation of solution sensitivities to thermochemical variables and parameters for control [7] and uncertainty quantification [8, 9]. We carry corresponding derivatives to obtain these sensitivities. The derivatives of the chemical source terms with respect to the composition vector, called the chemical Jacobians, are the basis for model reduction methods [3, 10, 11] and state-of-the-art time integration [12–14], which addresses numerical stiffness.

The implementation of analytical Jacobians poses challenges to efficiency, and numerical approximations via finite differences require ad hoc tuning of the perturbation size to meet accuracy requirements [15]. Automatic differentiation (AD) is an attractive alternative because it provides the exact derivatives without having to implement them manually in the source code [16]. In practice, AD represents functions as graphs and calculates corresponding derivatives via the chain rule as it traverses them. AD libraries and thermochemistry libraries often belong to disparate compilation units, so the necessary graphs cannot be readily constructed. The presented approach is AD-compatible via the same abstractions that enable accelerated, device-offloaded computation.

1.2. *Principal Contribution*

This work's principal contribution is a hierarchy of abstractions that enable portable and performant computation for combustion thermochemistry. The method facilitates symbolic analysis and GPU execution for performance while minimizing barriers to entry or user intervention. The implementation, called Pyrometheus, is open-source and permissively licensed. The workflow and capabilities are illustrated in fig. 1. The approach uses symbolic programming, as presented in section 2.2, to separate the mathematical operations of combustion thermochemistry from simulation-specific data details on which they are performed. The symbolic representation generates code commonly

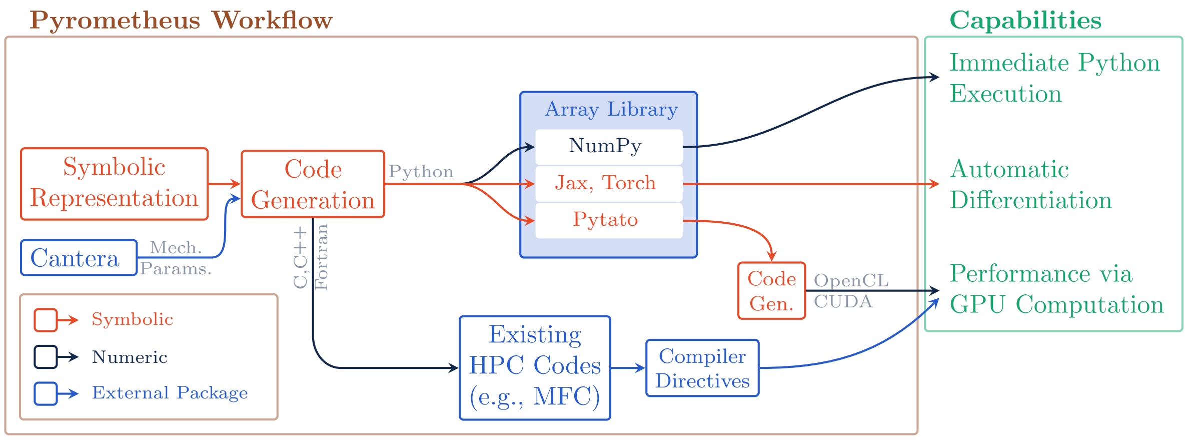


Figure 1: Pyrometheus generates code for combustion thermochemistry that can be transformed to meet the user's goals. Based on a symbolic representation of the formulation (section Appendix A), it can produce Python, C, and Fortran code. The generated Python code can be executed with NumPy, automatically differentiated using compatible libraries like JAX, transformed into CUDA or OpenCL, or offloaded with directives such as OpenACC or OpenMP for accelerator execution.

used for scientific computing, including Python, C++, and Fortran via mappings introduced in section 2.3.

From the generated code, there are two paths to computation. The first retains a Python control environment, from which the code can be executed, automatically differentiated, or offloaded to accelerator hardware. We achieve this environment via intermediate representations at progressively lower levels of abstraction (such as data flow graphs), which are obtained by staged addition of detail, including array semantics and loop patterns (section 2.4). These transformations are demonstrated in section 4. The second path to computation supports C++ or Fortran, which have their idiosyncratic control environments and offloading strategies. This approach puts the computation of the thermochemistry at the same level of abstraction as the flow solver. This path to computation is demonstrated in section 5.

1.3. Related Work & Outline

Pyrometheus overcomes the challenges associated with analyzing and porting Cantera (and related libraries) to GPUs. Cantera imposes assumptions on inputs to source-term computation: they are assumed to be pointwise quantities stored in memory as double-precision floats. These assumptions further condition subsequent thermochemistry calculations (implemented as classes, deeply-nested loops, and conditionals), performance, and capabilities. Principally, computational graphs cannot be obtained from such libraries due to the imposed data types. These assumptions do not constrain Prometheus-generated code, allowing it to provide exact derivatives and GPU execution via computational graph analysis, which are added capabilities over Cantera. As shown in fig. 1, Pyrometheus still uses Cantera to obtain mechanism-specific data (such as Arrhenius parameters) at code generation.

Pyrometheus is related to the pyJac library of Niemeyer et al. [15], which generates CUDA code for GPU thermochemistry evaluation from Cantera. Particularly, pyJac implements analytical Jacobians for GPU evaluation, leveraging sparsity and memory access patterns [17]. Similarly, the Pele suite generates C++ code for quasi-steady state (QSS) chemical source terms and their Jacobians [18], but rely on a Pele-intrinsic portability layer to offload the computation to GPUs [2].

Language-specific solutions achieve excellent GPU performance. For example, Sewerin and Rigopoulos [19] implemented a time-stepping scheme for chemical source terms in OpenCL, and Mao et al. [20] developed a comprehensive CUDA reacting-flow solver that integrates deep neural networks. Beyond the CUDA/OpenCL computational model for GPUs, Bauer et al. [21] implemented the thermochemistry in the Legion language that follows a task-based parallelization strategy. Pyrometheus has the added advantage that it is not tied to machine-specific languages (e.g., CUDA, OpenCL, Legion, or even C++) and is performance-portable across vendors (e.g., AMD, NVIDIA, and Intel accelerators): the same Pyrometheus-generated code can be executed on a CPU, or re-expressed to run on accelerators, which is demonstrated in section 4. Pyrometheus is also advantageous in that neither the thermochemistry engine (Cantera) nor the user needs to provide explicit Jacobian-generation code due to the incorporation of automatic differentiation (AD). This aspect is related to Arrhenius.jl, the Julia code of Ji et al. [22] that uses AD to calibrate combustion mechanisms. However, Pyrometheus and Arrhenius.jl serve different goals. Pyrometheus is oriented towards large-scale simulations, supporting external target languages and control environments.

This work is also related to the computational design for chemical source terms, as presented in Barwey and Raman [23]. In their approach, the equations of combustion thermochemistry are cast in matrix form to mirror the evaluation of artificial neural networks (ANNs). In this way, GPU-optimized ANN libraries can be used to compute the source terms. However, their formulation is limited to computation on GPUs without derivatives, a limitation we overcome with Pyrometheus.

More broadly, our approach is related to the Pystella package [24], a Python-based PDE solver for astrophysics applications. Pystella achieves performance on accelerators through code generation and abstractions. Our approach also shares design principles with the work of Kulkarni [25], where computational concerns are separated to achieve near-roofline GPU performance for finite-element methods. However, Pyrometheus focuses on chemical source terms, whereas Pystella and Kulkarni target numerical discretization of differential operators.

The paper is organized as follows. The novel computational approach is introduced in section 2, and code correctness is confirmed in section 3. Example implementations of thermochemistry in Python, along with quantitative results, are presented in section 4. Incorporation into a high-performant flow solver called MFC is demonstrated in section 5. Section 6 summarizes the principal features and discusses potential of extensions.

2. Computational Approach

2.1. Goal & Choice of Computational Paradigm

To meet the goals in section 1—portable performance on CPUs/GPUs and compatibility with automatic differentiation (AD)—we combine the CUDA/OpenCL execution model with a symbolic program representation. In the CUDA/OpenCL model, a host launches kernels on devices, and arrays are the basic data type. Our approach targets this model while remaining AD-compatible by postponing device- and layout-specific choices until needed. AD systems express programs as graphs and apply the chain rule along graph edges by overloading data types and primitive operations. Although this can appear orthogonal to CUDA/OpenCL, our abstraction reconciles the two by keeping the formulation symbolic until late in the workflow (fig. 1), where we introduce the necessary details for execution and differentiation.

2.2. A Symbolic Computational Representation of Combustion Thermochemistry

The full thermochemistry formulation implemented in Pyrometheus is given in section Appendix A. Here we focus on key expressions that guide the design. For an elementary binary reaction, the forward rate is

$$R_f = k(T)\, C_m\, C_n, \tag{1}$$

with Arrhenius coefficient

$$k(T) = A\, T^b \exp\left(-\frac{\theta_a}{T}\right), \tag{2}$$

where T is temperature, C_m and C_n are reactant concentrations, and $\{A, b, \theta_a\}$ are parameters. Evaluations of (1) recur across reactions, space, and time but are local and free of loop-carried dependencies, making them amenable to CUDA/OpenCL-style parallelism.

We represent (1) and (2) as expression trees using symbolic algebra. This elevates the computation to match the mathematical formulation, independent of execution venue (CPU/GPU) or AD. At this level, only mathematical operations are expressed; numerical values and device details are introduced later.

At the user-facing level of the Pyrometheus workflow (fig. 1), the representation is symbolic and implemented in Python for its flexible data model that supports operator overloading. Expression trees organize computations for efficient traversal and multi-language code generation (section 2.3).

Figure 2 illustrates expression trees for (1). The object `rxn_rate` has two children: the concentration product $C_m C_n$ and $k(T)$. We formalize trees with the symbolic programming Pymbolic package [26]; a minimal demonstrator for (1) is available at `https://github.com/pyrometheus/mini-pyrometheus`. We show only key elements here. Listing 1 shows the core definitions; the full implementations are open source.

The Pyrometheus approach of fig. 1 is general. It can be implemented using other symbolic engines such as SymPy. However, Pymbolic underlies many of the complementary tools that transform the symbolic representation into device code. It allows us to maintain cohesion in the approach.

Nodes are expressions, and overloaded operators create new expression types. Leaves have no children and are represented as `Variable`. Mixture variables, e.g., T and $\vec{C}$, are leaves. Function names are treated as `Variable` at this level and are disambiguated during code generation (section 2.3).

To assemble $k(T)$ in (2), numeric parameters $\{A, b, \theta_a\}$ are retrieved via a chemistry library. We use Cantera for its comprehensive Python interface, but the approach is general. Listing 2 shows the construction of k and R_f.

2.3. Code Generation via Translation of the Symbolic Representation

We generate target-language code by traversing expression trees and applying mapper rules that preserve semantics [27]. Although we illustrate Python code generation, the approach is language-agnostic; Pyrometheus also supports Fortran and C++. Listing 3 shows a mapper where each `map_*` method translates an expression node by recursively emitting code fragments (listing 4) for its children and composing a target-language expression.

Generated fragments are embedded in a static template for the target language. For Python (listing 5), the generated code depends on an abstract array library `pyro_np`. This separates concerns:

```python
class Expression:
    def __init__(self, children):
        self.children = children
    def __add__(self, other):
        return Sum((self, other))
    def __mul__(self, other):
        return Product((self, other))

class Sum(Expression):
    mapper_method = ’map_sum’

class Product(Expression):
    mapper_method = ’map_product’
```

Listing 1: Symbolic computational expressions underpin the representation. Concrete expressions such as `Sum` and `Product` carry `mapper_method` tags to guide code generation (listing 3). The release uses an extended version based on Pymbolic [26].

```python
def arrhenius_expr(rxn: ct.Reaction, temp: Variable):
    # Nonlinear functions
    exp = Variable(’exp’)
    log = Variable(’log’)
    # Arrhenius parameters
    log_a = np.log(rxn.rate.pre_exponential_factor)
    b = rxn.rate.temperature_exponent
    act_temp = -1 * rxn.rate.activation_energy / ct.gas_constant
    # Construct the Arrhenius expression
    return exp(log_a + b * log(temp) + act_temp / temp)

rxn = ct.Reaction(
    # Reactants & Products for M + N -> P + Q
    {’m’: 1, ’n’: 1}, {’p’: 1, ’q’: 1},
    # Arrhenius coefficients, taken from Reaction 1, San Diego mech
    {’A’: 35127309770106.477, ’b’: -0.7, ’Ea’: 8590 * ct.gas_constant})
k = arrhenius_expr(rxn, temp)
rxn_rate = k * conc_product

print(f’{isinstance(k, Expression)}’)
# >>> True
print(f’{isinstance(rxn_rate, Expression)}’)
# >>> True
```

Listing 2: Expression trees for the Arrhenius coefficient (2) and forward reaction rate (1). Trees combine symbolic variables with Arrhenius parameters obtained from a Cantera `Reaction` object.

```python
class CodeGenerationMapper:
    prec = {'var': 0, 'call': 1, 'sum': 2, 'mul': 3, 'div': 4, 'sub': 5}

    def parenthesize(self, expr_str, prec_expr, prec):
        if prec and prec > prec_expr:
            return f'({expr_str})'  # parenthesize result
        else:
            return expr_str

    def rec(self, expr, *args):
        if args:
            return getattr(self, expr.mapper_method)(expr, *args)
        else:
            return getattr(self, expr.mapper_method)(expr, None)

    def map_variable(self, expr, precedent): return expr.name

    def map_sum(self, expr, prec):
        return self.parenthesize(
            ' + '.join([self.rec(c, self.prec['sum']) for c in expr.children]),
            self.prec['sum'], prec)

    def map_product(self, expr, prec):
        return self.parenthesize(
            ' * '.join([self.rec(c, self.prec['mul']) for c in expr.children]),
            self.prec['mul'], prec)
```

Listing 3: Code-generation mapper translating expression trees into target-language code. Methods are applied recursively via `rec` to form a grammatically correct expression that preserves semantics.

```python
python_cgm = CodeGenerationMapper()
code_fragment = python_cgm.rec(conc_product)
print(isinstance(code_fragment, str))
# >>> True
print(code_fragment)
# >>> concentration[0] * concentration[1]
code_fragment = python_cgm.rec(k)
print(isinstance(code_fragment, str))
# True
print(code_fragment)
# >>> usr_np.exp(31.19 + -0.7 * usr_np.log(temperature) +
# >>>     -8590.0 / temperature)
```

Listing 4: Example fragments generated for the concentration product and Arrhenius coefficient. Applying the mapper `cgm` to a symbolic `Expression` yields a code fragment string.

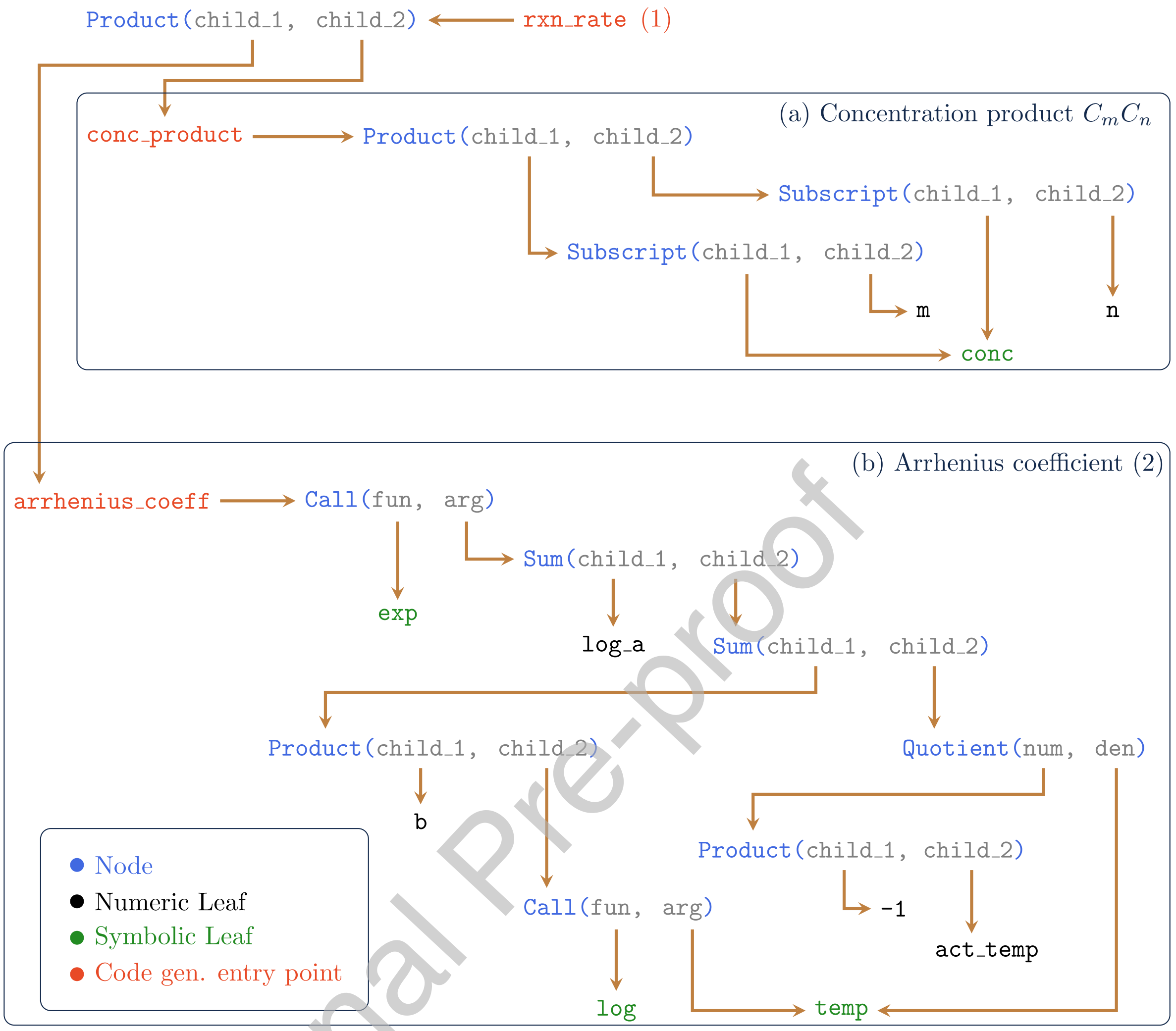

Product(child_1, child_2)
rxn_rate (1)
(a) Concentration product $C_m C_n$
conc_product
Product(child_1, child_2)
Subscript(child_1, child_2)
Subscript(child_1, child_2)
m
n
conc
(b) Arrhenius coefficient (2)
arrhenius_coeff
Call(fun, arg)
Sum(child_1, child_2)
exp
log_a
Sum(child_1, child_2)
Product(child_1, child_2)
Quotient(num, den)
b
Product(child_1, child_2)
Call(fun, arg)
-1
act_temp
log
temp
● Node
● Numeric Leaf
● Symbolic Leaf
● Code gen. entry point

Figure 2: Expression trees for the reaction rate (1): (a) the $C_m C_n$ term; and (b) the Arrhenius coefficient (2). Nodes (blue) and leaves (green) are operations and symbolic variables defined in listing 1. Nonlinear functions are treated as leaves and mapped at code generation. Orange nodes mark code-generation entry points.

generated code expresses the thermochemistry and `pyro_np` supplies array semantics (e.g., NumPy, JAX, or Pytato). The same generated code thus supports CPU execution, GPU offload, and AD depending on the chosen array backend.

Rendering produces executable Python. The generated code can be obtained and visualized through the examples. Correctness is demonstrated in section 3. Because array semantics are deferred, the same generated code can be paired with NumPy for CPU execution or with libraries enabling AD and GPU offload. The Pymbolic-style representation is cohesive with the companion tools that realize downstream transformations to device code, as explained in what follows.

2.4. Intermediate Representations and the Path to AD & GPU Computation in Python

Python offers stable access to HPC tools for communication, I/O, and GPU offloading (e.g., mpi4py, h5py, PyOpenCL, PyCUDA) [28], making it a practical control environment [29, 30]. We obtain

```python
from mako.template import Template

code_tpl = Template("""
class Thermochemistry:
    def __init__(self, pyro_np):
        self.pyro_np = pyro_np

    def get_rxn_rate(self, temperature, concentration):
        k = ${cgm.rec(arrhenius_expr(rxn, Variable("temperature")))}
        conc_product = ${cgm.rec(conc_product)}
        return k * conc_product
""", strict_undefined=True)

code_str = code_tpl.render(
    Variable=Variable,
    arrhenius_expr=arrhenius_expr,
    rxn=rxn,
    cgm=python_cgm,
    conc_product=conc_product
)
```

Listing 5: Python template for computing (1). Braced placeholders denote substitutions performed during rendering. The constructor argument `pyro_np` selects the array library.

AD and GPU capabilities by extracting a data-flow graph (DFG) from the generated Python code and transforming it through intermediate representations (IRs). Users interact with the generated Python. While the DFG and subsequent IRs remain internal, they are fully accessible.

The generated code assumes only NumPy-like array syntax (e.g., `pyro_np.exp`). The choice of array library then determines operational semantics—broadcasting, memory, or device placement—without changing the generated thermochemistry code. In the full release, we pair the code with array libraries JAX or PyTorch for AD, and Pytato and Loopy for GPU offload. To expose the underlying ideas, we implement two minimal custom array libraries: one for AD and one for GPU offload.

2.4.1. Custom Arrays to Showcase Automatic Differentiation

AD exactly computes the gradient

$$\mathcal{G} \equiv \left\{ \frac{\partial R_f}{\partial T}, \frac{\partial R_f}{\partial C_m}, \frac{\partial R_f}{\partial C_n} \right\} \tag{3}$$

by applying the chain rule along the DFG from output R_f to inputs $\{T, C_m, C_n\}$. While (3) is simple analytically, AD becomes essential for the full formulation in section Appendix A.

We construct the DFG by evaluating `get_rxn_rate` with a custom array type (listing 6) that stores numeric data and operational dependencies: overloaded operators record parents and provide local gradient rules (`grad_fn`), composing a small `adiff_np` library that mirrors the interfaces of JAX and PyTorch at a minimal scale.

Pairing the generated code with `adiff_np` evaluates R_f and builds the DFG. A recursive back-propagation (https://github.com/pyrometheus/mini-pyrometheus) then accumulates derivatives from root to leaves via each node's `grad_fn`. The user interface exposes this as a `gradient` method on the DFG (listing 7).

Because the AD pass is recursive, data-dependent control flow (e.g., the temperature inversion of

```python
class AutodiffArray:
    def __init__(self, values: list, children, name=None):
        self.values = np.array(values, dtype=np.float64)
        self.children = children
        self.name = name
    def __mul__(self, other):
        return AutodiffProduct(self.values * other.values, children=[self, other])
    def gradient(self,):
        ad_walker = AutodiffWalker()
        return ad_walker.compute_gradient(self)

class AutodiffVariable(AutodiffArray):
    def __init__(self, values, name):
        self.values = values
        self.name = name
        self.children = []
    def grad_fn(self, grad):
        return np.zeros_like(grad)

class AutodiffProduct(AutodiffArray):
    def grad_fn(self, grad):
        return (grad * self.children[1].values, grad * self.children[0].values)
```

Listing 6: Arrays for AD: numeric values plus a `children` attribute and local `grad_fn` rules for node types (e.g., `AutodiffVariable`, `AutodiffProduct`).

```python
temp_np = np.linspace(1000, 2100, 10, endpoint=False)
conc_np = np.stack((0.2 * np.ones(10), 0.8 * np.ones(10)))

pyro_gas = Thermochemistry(pyro_np=adiff_np)
temp_ad = autodiff.AutodiffVariable(temp_np, name='temperature')
conc_ad = autodiff.AutodiffVariable(conc_np, name='concentration')
rxn_rate = pyro_gas.get_rxn_rate(temp_ad, conc_ad)
g = rxn_rate.gradient()
```

Listing 7: Recursive propagation of derivatives on the DFG.

section Appendix A) must be handled carefully. Comprehensive libraries such as JAX provide the necessary tools; we leverage these in section 4.3 and discuss further limitations of this approach in section 6.

2.4.2. Custom Arrays with Underlying Transformations to Enable GPU Computing in Python

For GPU offload, we retain Python control flow while generating device code through transformations applied to the DFG. To avoid expensive host-side evaluation during graph construction, we implement deferred (lazy) arrays that carry shapes but not values as `lazy_np`. The `lazy_np` array library supports efficient data movement and kernel generation.

The DFG represents algebra but not loop structure, memory layout, or parallelism. We map the DFG to a Loopy kernel [31] specifying loop domains for array instructions. From the Loopy kernel, we generate CUDA/OpenCL and invoke it via PyCUDA/PyOpenCL, keeping orchestration in Python.

The initial kernel is sequential. We derive parallelism by splitting loop indices into work-group local

```python
class LazyArray:
    def __mul__(self, other):
        output_shape = np.broadcast_shapes(self.shape, other.shape)
        return ArrayExpression(expr=Product((self, other)),
                               shape=output_shape)
    # def __add__(self, other): ...

class Placeholder(LazyArray):
    def __init__(self, name, shape):
        self.expr = Variable(name)
        self.shape = shape

class ArrayExpression(LazyArray):
    def __init__(self, expr, shape):
        self.expr = expr
        self.shape = shape
        self.cuda_prg = None
    def compile(self, knl_name, wg_size):
        self.wg_size = wg_size
        from minipyro.pyro_np.loopy import assemble_cuda
        self.cuda_prg, self.cuda_code = assemble_cuda(self, knl_name)
    def evaluate(self, *np_data):
        assert self.cuda_prg is not None
        # grid = ...
        # block = ...
        import pycuda.gpuarray as gpuarray
        dev_data = [gpuarray.to_gpu(a) for a in np_data]
        self.cuda_prg(*dev_data, grid=grid, block=block)
```

Listing 8: Deferred-evaluation arrays for GPU offload. `compile` applies a sequence of Loopy transformations and emits CUDA code via the `assemble_cuda` method, available through `https://github.com/pyrometheus/mini-pyrometheus/`.

```python
pyro_class = get_thermochem_class()
pyro_gas = pyro_class(lazy_np)

wg_size = 32
num_x = 32 * wg_size
temp = lazy_np.Placeholder(name='temperature', shape=(num_x, num_x))
conc = lazy_np.Placeholder(name='concentration', shape=(2, num_x, num_x))

rxn_rate = pyro_gas.get_rxn_rate(temp, conc)
rxn_rate.compile('get_rxn_rate', wg_size)

temp = 300 * np.ones((num_x, num_x))
conc = 0.5 * np.ones((2, num_x, num_x))
rate = np.zeros((num_x, num_x))
rxn_rate.evaluate(conc, rate, temp)
```

Listing 9: Python interface to GPU evaluation of (1). The `compile` path generates and launches CUDA with work groups of size 32.

and global components:

$$k = i + g\,j, \tag{4}$$

where g is the work-group size. Loopy applies this transformation and related memory-mapping steps; the resulting kernel maps efficiently to GPU execution. PyCUDA/PyOpenCL handles scheduling concerns.

3. Code Correctness

The generated code is compared to Cantera, a well-tested library, to ensure correctness. The tests cover the equation of state, species thermodynamic properties, and kinetic rates and their coefficients. The equation of state (EOS) tests ensure that pressure and temperature are correctly computed from known states. Species thermodynamic properties are tested over the temperature range of validity for NASA polynomials. Kinetic properties that depend on mixture composition are tested for various states along trajectories of autoignition simulations. Code transformations and intermediate representations implemented using complementary tools such as Loopy and Pytato are extensively tested by their distributions.

To standardize testing and the Pyrometheus API across languages, bindings from C++ and Fortran to Python are used to maintain a single backend-agnostic test suite written in Python (as opposed to writing the tests in all supported languages). Standard build and interfacing tools generate the bindings and dynamic module imports in Python ensure that the tests interact with each backend. As a result, the C++ and Fortran backends are as well tested as the Python code. Each pull request to the GitHub repository tests all backends.

For arbitrary thermochemical quantity $\vec{\phi}$ (such as the reaction rate in (1)), conventional absolute and relative errors

$$\varepsilon_{\mathrm{abs},p}(\vec{\phi}) \equiv \|\vec{\phi}_{\mathrm{pyro}} - \vec{\phi}_{\mathrm{ct}}\|_p, \qquad \varepsilon_{\mathrm{rel},p}(\vec{\phi}) \equiv \frac{\|\vec{\phi}_{\mathrm{pyro}} - \vec{\phi}_{\mathrm{ct}}\|_p}{\|\vec{\phi}_{\mathrm{ct}}\|_p}, \tag{5}$$

quantify code correctness, with $\vec{\phi}_{\mathrm{pyro}}$ and $\vec{\phi}_{\mathrm{ct}}$ the results from Pyrometheus and Cantera, and the vector dimension typically representing temperature range.

Errors are in the order of machine precision, as no approximations are made with respect to Cantera. Absolute errors in thermodynamic properties of selected species—as predicted by NASA polynomials (A.12) and (A.31)—are shown in fig. 3. Rate coefficients and equilibrium constant, with corresponding errors (5), are shown in fig. 4 for selected reactions and species in the GRI-3.0 mechanism. Correctness of mixture-averaged transport properties is demonstrated using states from a one-dimensional freely propagating flame stoichiometric methane–air flame at 101 325 Pa (computed with Cantera). Viscosity (A.20), thermal conductivity (A.23), and the diffusivity (A.18) of exemplar species are shown in fig. 5.

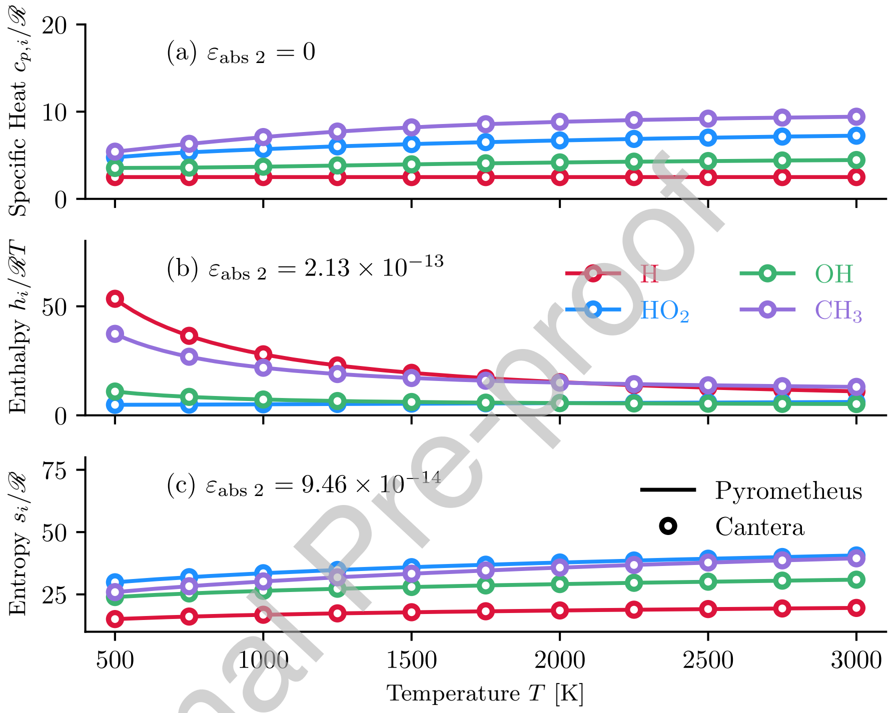


Figure 3: Thermodynamic properties for selected key radicals in the GRI-3.0 mechanism: (a) specific heat at constant pressure, (b) enthalpy, and (c) entropy. The reported error is averaged over all species in the mechanism and follows (5) with $p = 2$.

4. Demonstration I: Computation via Deferred Addition of Details

4.1. Aim of Demonstrations

We demonstrate the use and advantages of the computational approach of section 2 for combustion simulations, applied to the full formulation of section Appendix A. The demonstrations follow the workflow of fig. 1: from symbolic (on the leftmost edge) to the capabilities expressed on the rightmost edge by adding details only when needed. We show how the same generated code can process numeric data using NumPy, be automatically differentiated with special-purpose array libraries such as JAX, and transformed for computation on GPU devices using the symbolic array library Pytato. Homogeneous reactors and flames are simulated, and advanced applications to sensitivity analysis and computational cost modeling are discussed.

Pyrometheus is a generalized thermochemistry toolkit that can be used in combustion simulations without restrictions on the numerical scheme used to discretize reacting flow equations. For the

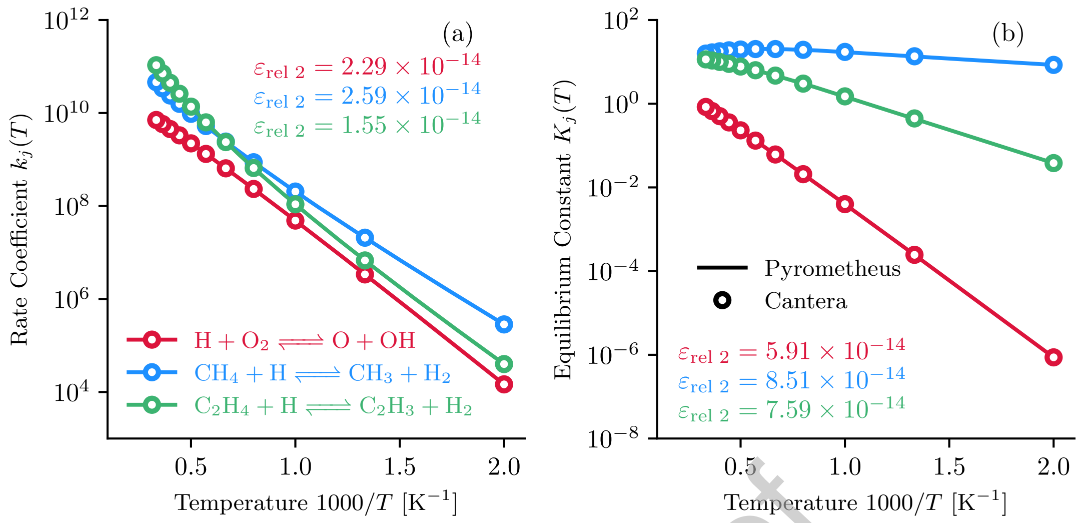


Figure 4: Reaction properties for selected key reactions in the GRI-3.0 mechanism: (a) reaction rate coefficient (A.9), and (b) equilibrium constant (A.10). Reported errors are based on (5) with $p = 2$.

```
sol = ct.Solution('sandiego.yaml')
pyro_cls = pyro.codegen.python.get_thermochem_class(sol)
```

Listing 10: Python thermochemistry code generation.

numerical experiments in this section, specific time integration schemes and spatial discretizations are implemented. This choice illustrates how to use Pyrometheus generated code. The generated code can be integrated with more sophisticated numerical schemes, such as variable-order implicit [32] or semi-implicit [33, 34] time integrators. Dedicated time integration packages such as CVODE [32], with which Pyrometheus is compatible, can provide additional advantages through error control.

Pyrometheus can generate both CUDA and OpenCL code for GPU execution. Both choices are based on the same execution model. We do not expect a substantial performance difference between these two choices. The following demonstrations make use of the CUDA code to leverage NVIDIA's extensive profiling tools. OpenCL provides portability to devices by other vendors, such as Intel and AMD.

4.2. Python Code Generation and Immediate Execution using NumPy

The initial step to computation is the generation of Python code from the symbolic representation. The key input to code generation is a chemical mechanism and its parameters. The mechanism is encapsulated in the Cantera solution object of listing 10, line 1. Here, we use the hydrogen–air San Diego mechanism [35]; larger mechanisms are routinely tested in the distribution. The Cantera `Solution` interface populates mechanism-specific parameters in our symbolic representation. We map the symbolic representation to Python with the procedure described in section 2.2. The generated Python code processes arrays without constraints on array shape or data type (numeric or symbolic). The code is general in this sense. The following examples showcase distinct capabilities derived from it without modification or re-generation.

In Python, Pyrometheus generates a class; computation requires that it be instantiated into an

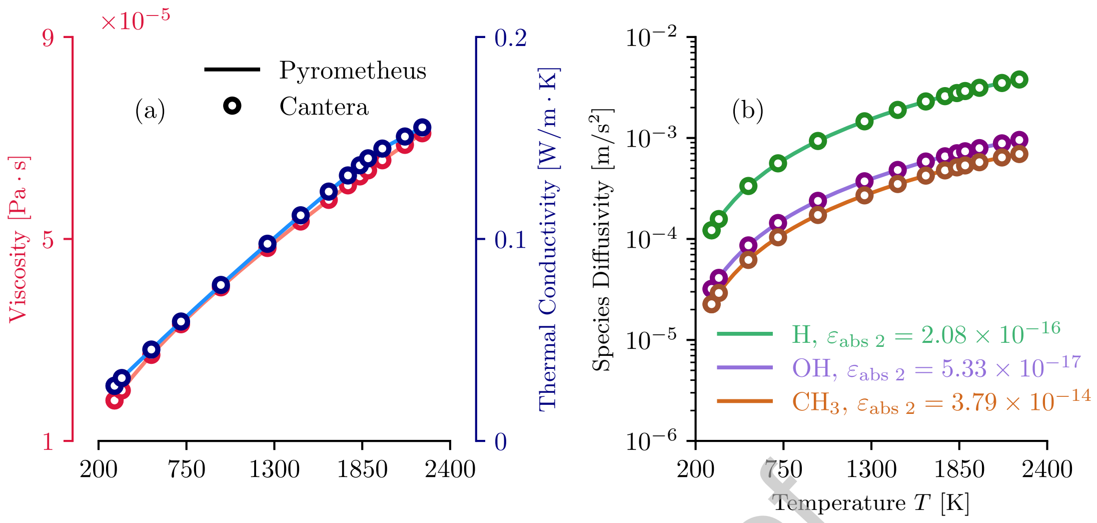


Figure 5: Mixture-averaged transport properties for a laminar methane–air free flame at 101 325 Pa: (a) mixture viscosity and thermal conductivity; and (b) diffusivities of selected species in the GRI-3.0 mechanism. Reported errors follow (5) with $p = 2$.

```
import numpy as np
pyro_gas = make_pyro_object(pyro_cls, np)
```

Listing 11: NumPy-based instance of the generated code for immediate calculation without transformations. The `make_pyro_object` function resolves minor syntactic differences in how arrays are concatenated across multiple array libraries.

object. Per section 2.4, the class constructor expects a NumPy-like library that provides the semantics for the array-based expressions in the generated code. The established library NumPy provides these rules for numerical data, so it to execute the methods of the generated code immediately. A NumPy-based Pyrometheus object is instantiated in listing 11. Subsequent NumPy-based demonstrations build from illustrative calculations to simulations of reacting flows.

An illustrative computation of the net production rates (A.5) is shown in listing 12. The inputs are scalar density and temperature, as well as species mass fractions represented by simple one-dimensional arrays. The more complex case of data on a grid for multi-dimensional combustion will be shown subsequently. Line 8 shows the call to the specific method that computes (A.5) from these inputs, and line 12 compares the result against Cantera. We stress that these code listings are part of a broader tutorial distributed with the code and can be run by users.

The generated code also operates on multi-dimensional arrays without modification. Two dimensional inputs are shown in listing 13. While obviously different to the inputs of listing 12, the interface to the computation of (A.5) remains identical to lines 8–9 in listing 12. This embodies the separation of concerns in the programming strategy: the burden of array details falls on the array library (NumPy in this case), not the generated code. This cannot be done with thermochemistry libraries, which impose constraints on their input types and require nested calls within loops to compute across multiple points.

```
temperature = 1200
sol.TP = temperature, pyro_gas.one_atm
sol.set_equivalence_ratio(phi=1, fuel='H2:1',
                          oxidizer='O2:0.21, N2:0.79')
mass_fractions = pyro_gas._pyro_make_array(sol.Y)

# Compute the density and molar net production rates
density = pyro_gas.get_density(pyro_gas.one_atm, temperature, mass_fractions)
omega = pyro_gas.get_net_production_rates(density, temperature, mass_fractions)

# Compare against Cantera
assert pyro_gas.usr_np.linalg.norm(omega - sol.net_production_rates) < 1e-14
```

Listing 12: Example computation of the net production rates. Here, the density and temperature are scalars, and the mass fractions are one-dimensional arrays of length $N_s = 9$ for the hydrogen–air San Diego mechanism.

```
num_x = 256
num_y = 256
mass_fractions = pyro_gas._pyro_make_array([
    pyro_gas.usr_np.tile(y, (num_y, num_x)) for y in sol.Y
])
temperature = 1200 * pyro_gas.usr_np.ones((num_y, num_x))

# Compute the density and molar net production rates
density = pyro_gas.get_density(pyro_gas.one_atm, temperature, mass_fractions)
omega = pyro_gas.get_net_production_rates(density, temperature, mass_fractions)
```

Listing 13: Example computation of the net production rates for multi-dimensional arrays that represent thermochemistry data structured on a grid.

4.2.1. Application: Simulations of Autoignition and Flame Propagation

With the NumPy backend of Pyrometheus (listing 11, line 2), we simulate autoignition in a stoichiometric hydrogen–air homogeneous reactor by evolving mass fractions in time per (A.1) via explicit time stepping with standard fourth-order accurate explicit Runge–Kutta. Results are shown in fig. 6. The time step $\Delta t = 10\,\mathrm{ns}$ is used. Larger time steps lead to unstable simulations. Implicit time integrators, discussed in section 4.3, can stabilize simulations At this stage, we have shown that even without advanced AD capabilities, Pyrometheus can be used to simulate combustion using NumPy.

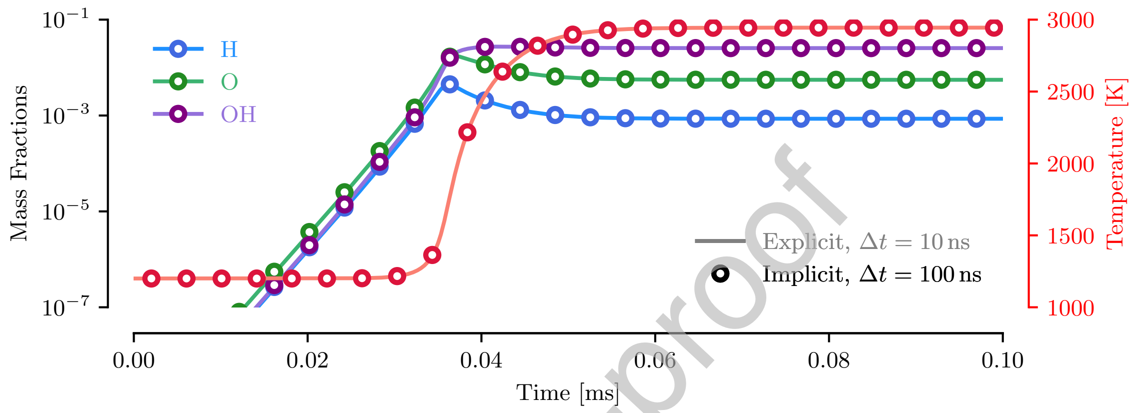


Figure 6: Thermochemical state during auto-ignition of a stoichiometric hydrogen–air mixture. Simulations using Pyrometheus-generated Python code. The explicit time marching uses NumPy-based Pyrometheus thermochemistry, while the implicit scheme uses JAX-based thermochemistry to enable AD for Jacobians.

We use the capability to process multi-dimensional arrays to simulate freely propagating laminar flames in one dimension. We numerically solve the compressible flow equations with chemical reactions [36], using the schemes of section Appendix A.3.2. The system includes the conservation of species mass densities (A.14), along with equations for the conservation of momentum density and total energy density. The $L = 0.08\,\mathrm{m}$ domain is discretized using 2048 points, and the $x = 0$ and $x = L$ boundaries are nonreflecting outflows via characteristic boundary conditions [37]. The pressure is 101 325 Pa throughout. The initial mixture is divided into two regions: a $T = 1600\,\mathrm{K}$ region of equilibrium products (at constant temperature and pressure), and a $T = 300\,\mathrm{K}$ region of unburned reactants in stoichiometric proportion. The two regions are smoothly connected via a hyperbolic tangent.

The resulting solver is provided with the examples. It inherits the advantages of Pyrometheus, expressing just the equations without loops or other entangling computational details. Its routines express only the mathematical operations. NumPy handles array processing, so computational concerns have been separated.

Flame simulation results are shown in fig. 7. After an initial transient, during which the mixture ignites, a flame is established and propagates into the fresh, cold mixture at a constant speed of $2.359\,\mathrm{m/s}$. This result is within 1.42% of the flame speed predicted by Cantera's steady low-Mach number solver. The error decreases to 0.12% for a grid with $N = 4096$ points. To further validate the entire solver–chemistry toolchain, simulations at $\phi = \{0.8, 1.2\}$ for hydrogen– and ethylene–air mixtures have been conducted. All input and post-processing files are provided with the distribution. Errors are within 1.81% for hydrogen and 2.10% for ethylene, which we deem satisfactory in light of other uncertainties. For example, the difference in stoichiometric flame speed

```
import jax.numpy as jnp
pyro_gas = make_pyro_object(pyro_cls, jnp)
```

Listing 14: JAX-based Pyrometheus instance for automatic differentiation.

between mixture-averaged and multi-component transport is 1.87% for hydrogen–air.

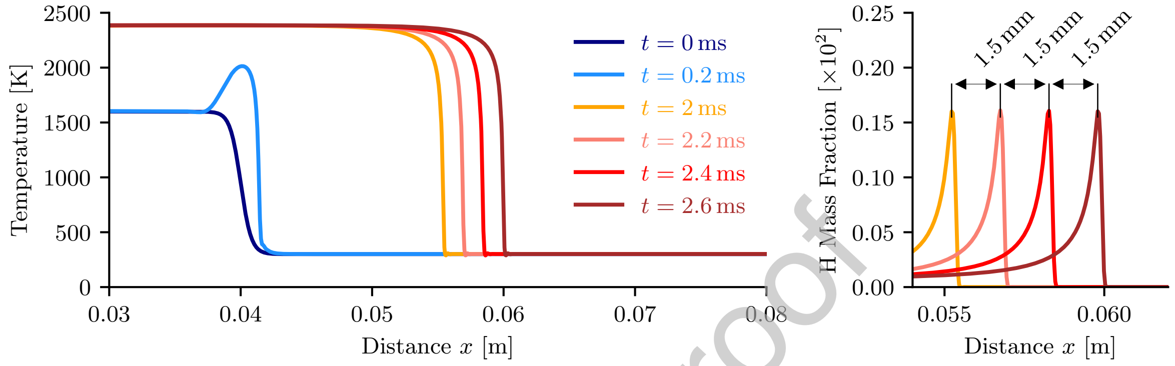


Figure 7: Propagation of a freely propagating laminar flame simulated using Pyrometheus-generated Python code: (a) temperature, and (b) H mass fraction. The pressure is 101 325 Pa and the equivalence ratio is 1. After an initial transient, the flame ignites and propagates steadily into the unburned mixture.

The preceding examples demonstrate how to use the generated code instantly for multiple use cases. Beyond their illustrative value, these examples can serve researchers working on novel chemical mechanisms by providing scripting access to kinetic properties such as ignition time and flame speed. The following examples extend deeper into our pipeline to computation via symbolic array libraries.

4.3. Automatic Differentiation Enables Implicit Time Marching

We now focus on cases that require exact derivatives. These are computed with automatic differentiation tools that provide extended meaning for array expressions (section 2.4.1). They construct the data-flow graph and propagate derivatives along its edges to compute gradients.

The starting point is the generated class of listing 11. From it, we create instances that can be automatically differentiated by pairing the generated class with libraries such as Google JAX and PyTorch. JAX is demonstrated in listing 14. The same strategy applies to PyTorch.

To construct the data-flow graph, we must ensure data-independent control flow. The only iterative procedure in thermochemistry computation is the inversion of temperature from internal energy and mass fractions (section Appendix A.3.1). This procedure is subject to termination by data-dependent tolerance. Listing 15 implements the inversion using JAX constructs that ensure the DFG can be constructed.

We proceed to compute derivatives automatically. Listing 16 demonstrates differentiation of the chemical source terms $\vec{S}$ in (A.1). To obtain their Jacobian

$$\mathbf{J} \equiv \frac{\partial \vec{S}}{\partial \vec{Y}}, \tag{6}$$

with $\vec{Y}$ the mass fractions, JAX builds the computational graph in line 11 of listing 16 for the

```
def get_temperature(self, energy, temp_init, mass_fractions,
                    do_energy=True):

    def cond_fun(temperature):
        f = energy - self.get_mixture_internal_energy_mass(
            temperature, mass_fractions
        )
        j = -self.get_mixture_specific_heat_cv_mass(
            temperature, mass_fractions
        )
        return self.usr_np.linalg.norm(f/j) > 1e-10

    def body_fun(temperature):
        f = energy - self.get_mixture_internal_energy_mass(
            temperature, mass_fractions
        )
        j = -self.get_mixture_specific_heat_cv_mass(
            temperature, mass_fractions
        )
        return temperature - f/j

    return jax.lax.while_loop(cond_fun, body_fun, temp_init)
```

Listing 15: Data-independent temperature inversion, implemented using JAX constructs.

function defined in lines 3–8. It then applies the chain rule as it traverses the graph in line 12. The mixture state for the Jacobian is computed from the explicit auto-ignition simulation of fig. 6 at $t = 0.0375\,\mathrm{ms}$. In fig. 8, the AD Jacobian $\mathbf{J}_{\mathrm{AD}}$ is compared against a numerical approximation $\mathbf{J}_{\mathrm{FD}}(\delta)$ based on first-order finite differences with varying perturbation size δ using

$$\varepsilon_{\mathrm{rel},\delta} \equiv \frac{\|\mathbf{J}_{\mathrm{AD}} - \mathbf{J}_{\mathrm{FD}}(\delta)\|_F}{\|\mathbf{J}_{\mathrm{AD}}\|_F}, \tag{7}$$

where $\|\cdot\|_F$ is the Frobenius norm. As expected, the error decreases linearly with finite-difference perturbation size until it is overcome by machine round-off errors for $\delta > 10^{-9}$.

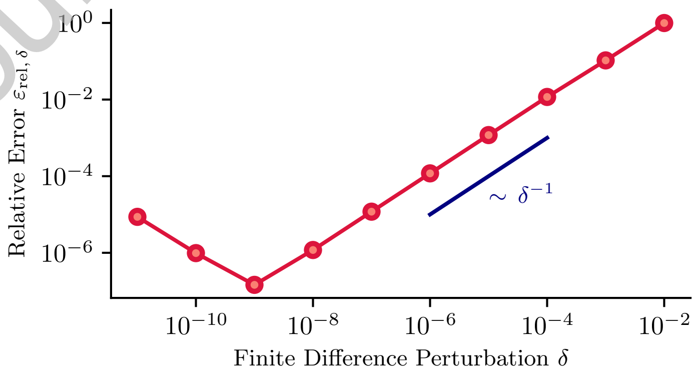


Figure 8: Relative error (7) in the Jacobian (A.36) calculated using AD and first-order finite difference with varying perturbation size δ.

Next, we use the JAX-based Pyrometheus instance to predict Hydrogen–air autoignition with the implicit scheme of section Appendix A.3.2. This scheme uses chemical Jacobians and allows a factor

```python
temp_guess = 1200
density, energy, mass_fractions = from_ignition_file(pyro_gas)
def chemical_source_term(mass_fractions):
    temperature = pyro_gas.get_temperature(energy, temp_guess, mass_fractions)
    return (
        pyro_gas.molecular_weights *
        pyro_gas.get_net_production_rates(density, temperature, mass_fractions)
    )

from jax import jacfwd
jacobian = jacfwd(chemical_source_term)
j_ad = jacobian(mass_fractions)
w_ad = chemical_source_term(mass_fractions)
```

Listing 16: Automatic differentiation of the chemical source terms using JAX. The temperature inversion routine is that of listing 15. The mixture state is obtained from the explicit autoignition simulation of fig. 6 at 0.0375 ms.

of 10 larger time step sizes than standard fourth-order accurate Runge–Kutta schemes. Larger time steps can be taken with variable-order time integrators with error control (e.g., CVODE [32]) and semi-implicit Euler (Seleu) methods [33, 34], which can be used without modification. Figure 6 shows that temperature and species profiles match those predicted with the explicit scheme.

4.4. Code Transformations Enable Execution on GPUs

Our next demonstration shows how, through code transformations, the Python class of listing 10 can be used for execution on GPUs. These devices leverage multiple logical units to perform the same calculation over large datasets efficiently. We seek a representation for computation on GPUs. Languages like CUDA and OpenCL provide the necessary access to their low-level constructs. In our approach, suitable array libraries transform the Python code into these languages. Importantly, we maintain access to every intermediate representation throughout the process. This degree of control is unavailable with machine learning libraries like JAX and PyTorch, which can also be used to execute on GPUs.

We quantify performance via a roofline cost model [38] to relate performance to memory traffic. The roofline represents hardware-imposed performance bounds, and optimizations (such as parallelization) represent intermediate ceilings. We quantify kernel performance via its nearness to the device roofline, whether memory- or compute-bound.

Recall that the Python class of listing 10 is not tied to particular input data or hardware. As depicted in fig. 1, it can still be thought of as a symbolic representation. To retrieve the DFG from the generated Python code, we pair it with the symbolic Pytato library, as shown in listing 17. Pytato arrays are shown in listing 18. These arrays are related to listing 8 of section 2.4.2. The Pytato arrays are defined through an identifier, the shape, and the element data type; there are no numerical data behind them. The symbolic arrays are inputs to the thermochemistry routines in listing 19. As these are executed, the state of the program is recorded into the corresponding DFG.

Our goal is to evaluate chemical source terms $\{S_i\}_{i=1}^{N}$, as defined in (A.1), using accelerators. The computational graph is tied to the inputs to the computation: mass fractions and temperature. In compressible flow solvers, on which we focus, the temperature is obtained as part of the equation

```python
import pytato as pt
pyro_gas = make_pyro_object(pyro_cls, pt)
```

Listing 17: Pytato-based Pyrometheus instance to retrieve the data flow graph from the generated Python code. Pytato provides lazy evaluation of array-based expressions, as discussed in section 2.4.2.

```python
inner_length = 32
num_x = 32 * inner_length
num_y = 32 * inner_length
density = pt.make_placeholder(name='density', shape=(num_x, num_y),
            ↪ dtype='float64')
temperature = pt.make_placeholder(name='temperature', shape=(num_x, num_y),
            ↪ dtype='float64')
mass_fractions = pt.make_placeholder(name='mass_fractions',
            ↪ shape=(pyro_gas.num_species, num_x, num_y), dtype='float64')
```

Listing 18: Symbolic input arrays to construct the data flow graph for the net production rates (shown in listing 19).

of state (EOS) evaluation before processing the chemical source terms. For this reason, we do not include the cost of temperature inversion in the following profiling results.

The Pyrometheus-based routine to evaluate the chemical source terms is shown in listing 19. While NumPy-like, this method is unconstrained to specific input types. For the symbolic inputs of listing 18, the routine returns a Pytato array expression akin to those in listing 8, from which the computational graph is assembled as a Python dictionary (lines 8–10). This graph expresses arithmetic relations between arrays and is to be transformed to incorporate loops and parallelism. However, it can be used beyond this purpose. For example, it can be used to create computational cost models by counting floating point operations.

In listing 20, the computational graph of listing 19 is translated to a Loopy kernel (section 2.4). The Loopy kernel extends the arithmetic expressed by the computational graph with data access patterns represented by loop variables. At this stage, any code generated from the kernel in line 1 of listing 20 will execute the instructions within its loops sequentially. To target more efficient device code, we imbue parallelization into the kernel. Following the CUDA/OpenCL execution model, loops are partitioned into outer and inner groups per (4), as shown in lines 6–13 of listing 20. Outer groups represent global work groups (or blocks in CUDA), while inner groups represent work items

```python
 1 def chemical_source_term(density, temperature, mass_fractions):
 2     omega = pyro_gas.get_net_production_rates(density, temperature,
             ↪ mass_fractions)
 3     return pyro_gas._pyro_make_array([
 4         w * om for w, om in zip(pyro_gas.molecular_weights, omega)
 5     ])
 6
 7 omega = chemical_source_term(density, temperature, mass_fractions)
 8 pyro_graph = pt.make_dict_of_named_arrays({
 9     f'omega{i}': w for i, w in enumerate(omega)
10 })
```

Listing 19: Construction of data flow graph `pyro_graph` using the symbolic arrays of listing 18.

```
pyro_kernel = pt.generate_loopy(
    pyro_graph,
    function_name='chemical_source_terms',
).program
...
pyro_kernel = lp.split_iname(
    pyro_kernel, 'i', inner_length,
    outer_tag='g.0', inner_tag='l.0'
)
pyro_kernel = lp.split_iname(
    pyro_kernel_split, 'j', inner_length,
    outer_tag='g.1', inner_tag='l.1'
)

pyro_kernel = pyro_kernel.copy(target=lp.CudaTarget())
code_str = lp.generate_code_v2(pyro_kernel).device_code()
```

Listing 20: The data flow graph for the chemical source terms $\{S_i\}_{i=1}^{N}$ is mapped onto a Loopy kernel. The loops are split into global and local domains of dimension (32×32). In CUDA, this dimension corresponds to the number of threads in each direction; in OpenCL, it corresponds to the number of local work items per global work group.

(or threads in CUDA). CUDA code is generated with Loopy and executed using PyCUDA.

Computational performance is shown in fig. 9 for the Intel Xeon Gold CPU (6454S) and the NVIDIA A100 GPU. Peak compute performance for these devices is 1.43 TFLOP/s (CPU) and 9.67 TFLOP/s (GPU). The CPU case was run with OpenCL-generated code compiled via PoCL, while the GPU case was run with CUDA-generated code and compiled with the nvhpc 24.5 toolkit. The problem size is (1024×1024) points, with (32×32) work groups (blocks) and (32×32) work items (threads). The GPU achieves 57.1% peak compute rate with a time per point of 8.86 ns. This performance is a factor of 425.81 faster than the OpenCL kernel on the CPU. This speedup is not due to architectural limits, as the OpenCL kernel is far from peak performance at about 1%. While further OpenCL kernel transformations can help bridge this gap, we do not pursue them. Instead, we focus on improving GPU performance in what follows.

Additional transformations can be applied to improve GPU performance. Data access patterns can be prefetched. Data are moved from global to local memory and registers in each logical unit before they are needed, mitigating latency and improving the rate at which operations are conducted. This memory management technique can be expressed in both CUDA and OpenCL. Loopy provides a corresponding transformation. Data prefetching on CPU devices has no apparent effect on time per point, as this optimization is likely exposed to the compiler. However, on the A100 GPU, prefetching improves performance by a factor of 2.3, or 69.5% of peak compute on the device.

The developed compilation pipeline is open to the user from symbolic to CUDA (or OpenCL). Corresponding PyCUDA (or PyOpenCL) packages handle further compilation while letting the user maintain control through Python. We have demonstrated the advantages of providing access to additional transformations, such as data reuse. Following section 2.4.2, Pytato–Loopy and subsequent transformations can be abstracted to a high-level interface that compiles the generated Python code. This way, users can interact with high-performance code through a high-level control layer with a low barrier to entry.

The approach transfers the burden of defining array-based semantics to an array library. Conse-

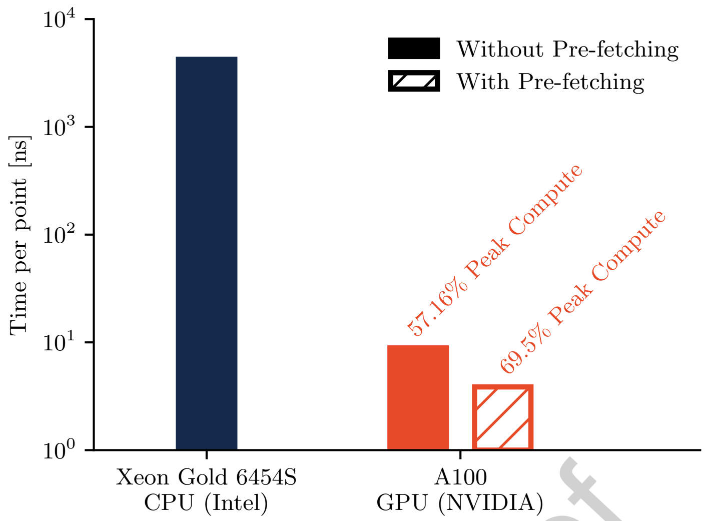


Figure 9: Compute time per point for hydrogen–air chemical source terms $\{S_i\}_{i=1}^{N}$ given temperature and mass fractions. Shorter bars indicate faster execution. The problem size is (1024, 1024) points. Each device executes (32, 32) threads per block. The data prefetching transformation halves execution time on the NVIDIA A100 GPU, reaching 69.5% of the device's peak fused multiply–add double precision performance.

quently, the generated Python code is compatible with numeric and symbolic packages like NumPy and Pytato. By separating algebraic and computational concers, the code can simulate auto-ignition and flames, and execute on CPUs and GPUs.

5. Demonstration II: Computation via Integration into Existing Solvers

The preceding demonstrations show a pipeline from symbolic representation to computation, using multiple code transformations and deferred detail addition. Some applications do not benefit from retaining symbolic capabilities, requiring code that instead operates on numeric data immediately within a different control environment than Python. This situation is common when using existing flow solvers. Here, we demonstrate that Pyrometheus can generate portable code for existing solvers from the symbolic representation without executing the entire pipeline described in section 4.

5.1. Description of the MFC Flow Solver

MFC is an open-source, MIT-licensed Fortran08 codebase. MFC has been developed over the last two decades to advance the understanding of multiphase flows (bubbles and droplets) and utilizes interface- and shock-capturing techniques [39, 40]. A predecessor of MFC was the first to employ the interface-capturing numerical approach to solve the five-equation model [41], which is a single-velocity and energy-equation reduced model of the multiphase model proposed by Baer and Nunziato [42].

For the past decade, MFC has been used to conduct high-fidelity multiphase flow simulations at extreme computational scales [40, 43], including those of high density ratios [44, 45]. Yet, MFC has not supported simulations of reacting flows due to the complexity of integration with exascale, performance-critical application development. Here, we use our computational approach to create a thermochemistry representation for MFC compatible with its parallelization strategy.

The MFC algorithm stores and uses coalesced memory arrays for stencil reconstructions. This strategy results in relatively high arithmetic intensity and performant code on GPU accelerators.

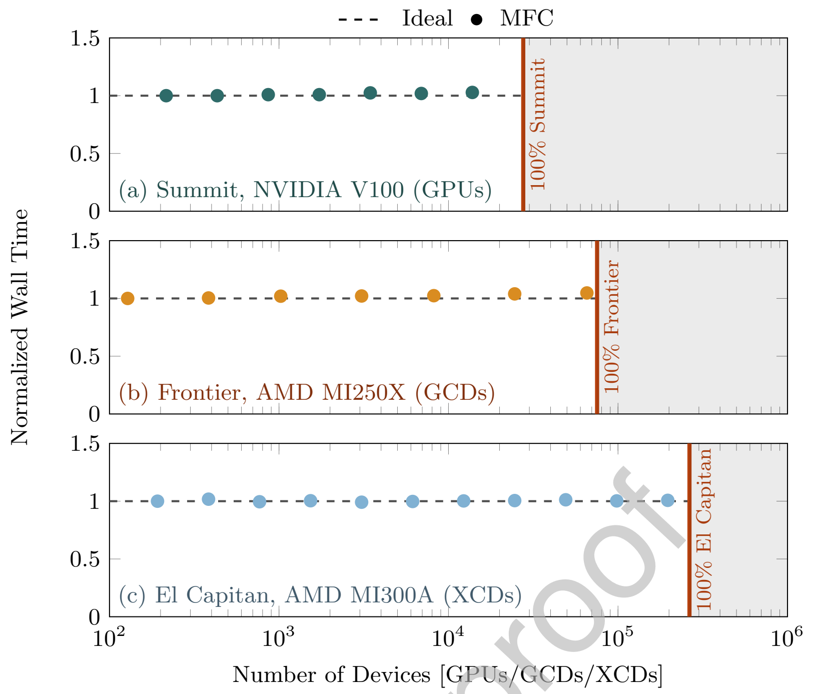


Figure 10: Weak scaling of MFC on LLNL El Capitan (AMD MI300A XCDs) and OLCF Frontier (AMD MI250X GCDs) and Summit (NVIDIA V100 GPUs) for a representative 3D structured-grid two-phase test problem.

It employs a directional splitting approach that evaluates these quantities once per time step, then reuses them [43, 46, 47]. MFC uses established computational methods, including high-order accurate interface reconstruction, approximate Riemann solvers, and Runge–Kutta time steppers. MFC handles parallel processing via MPI3 and domain decomposition.

GPU offloading is accomplished via OpenACC or OpenMP, whichever is more appropriate for the target device. MFC is performant on NVIDIA and AMD GPUs, including OLCF Summit (NVIDIA V100) and OLCF Frontier and LLNL El Capitan machines (AMD MI250X and MI300A). MFC's results on such machines and its GPU offload implementation are documented elsewhere [40, 48–51]. Here, we demonstrate how the generated Fortran code from Pyrometheus is readily integrated into MFC.

Figure 10 shows how MFC scales to better than 60% of all NVIDIA GPUs on OLCF Summit, 88% of the AMD MI250X GPUs (GCDs) ($>$ 66K) on OLCF Frontier (the first exascale computer), and 85% of the AMD MI300A APUs (XCDs) on LLNL El Capitan. The Summit simulations did not extend to the full system before it was decommissioned. Each GPU computation is approximately 500 times faster than one on a comparable CPU device [52].

5.2. Code Generation

The thermochemistry code for MFC is generated directly from the symbolic representation of section 2.2. Expressions are mapped to Fortran, and a template is rendered. The interface to this process is shown in listing 21, line 1, where the Cantera solution object is that of listing 10, line 1. The generated code is at the same abstraction level as MFC and forms part of the same compilation unit as the flow solver. As such, it is readily compatible with directive-based compile-time

```
pyro_code = pyro.codegen.fortran90.gen_thermochem_code(sol)
with open(f'thermochem_{sol.name}.f90', 'w') as outf:
    print(pyro_code, file=outf)
```

Listing 21: Fortran code generation for combustion thermochemistry. The resulting code is saved to a file that is part of the MFC compilation.

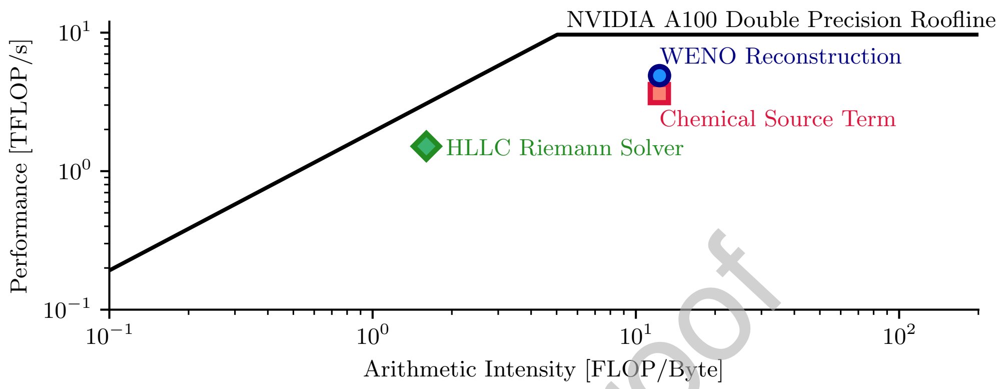


Figure 11: MFC kernel performance on an NVIDIA A100 GPU. The roofline corresponds to peak double-precision fused multiply–add instructions. The kernel for chemical source terms computes S_i in (A.14) for a given density, mass fractions, and internal energy.

optimization. The Pyrometheus Fortran template includes OpenACC and OpenMP directives, so it complies with MFC's (and many GPU-based codes) offloading strategies.

5.3. Performance of Pyrometheus-enabled MFC Simulations of Reacting Flows

Code performance is quantified using the roofline approach [38]. We compare the cost of evaluating chemical source terms (A.5) relative to that of kernels associated with fluxes, such as WENO and the HLLC approximate Riemann solver. Because performance is independent of physical configuration, it is measured in three-dimensional simulations of auto-ignition in a quiescent hydrogen–air mixture, modeled using the hydrogen subset of the GRI-3.0 mechanism (with 10 species, including Argon). These are straightforward to configure, and results translate directly to more complex configurations, such as the detonation of section 5.4. Results, shown in fig. 11, are presented for a grid size of 17.98×10^6 cells, corresponding to 84.5% GPU memory utilization. Performance was averaged over kernel calls (with a total of 15 per kernel).

As seen in fig. 11, the computation of chemical source terms is close, by 6.04 TFLOP/s, to the double-precision peak-FLOP ceiling. This performance is comparable to the WENO kernel, which is 4.76 TFLOP/s from the ceiling. In comparison, the computation of chemical source terms outperforms the HLLC approximate Riemann solver, which, while still close to its optimal performance, is memory-bound with an arithmetic intensity of 1.5 FLOP/Byte. The WENO kernel is the most expensive computation in MFC: its time per cell is 137 ms per call, a factor of 8.43 larger than the cost associated with the generated chemical source terms.

5.4. Pyrometheus-enabled MFC Simulations of Detonations

To demonstrate the integration of Pyrometheus into MFC, we simulate a one-dimensional detonation propagating into a quiescent stoichiometric hydrogen–argon mixture. This configuration is chosen because it couples flow and chemistry without complex boundary conditions and is well documented elsewhere [53, 54]. Reactions are modeled using the hydrogen subset of the GRI-3.0 mechanism. The domain of length $L = 0.12\,\mathrm{m}$ is discretized using 400 cells. The boundary at $x = 0$ is a reflecting wall, and the $x = L$ boundary is a non-reflecting outflow. The flow is initialized with a shock at the center of the domain. The shock propagates towards the wall, where it reflects and ignites the mixture as it propagates into the domain. As seen in fig. 12, a detonation forms after $\approx 190\,\mu\mathrm{s}$. Our simulations reproduce the location of the detonation front at $t = 230\,\mu\mathrm{s}$. The difference in temperature downstream of the wave with respect to the solution of Ferrer et al. [54] is $\lesssim 5.7\%$. This difference is small and can be attributed to the use of different chemical mechanisms by Ferrer et al. [54].

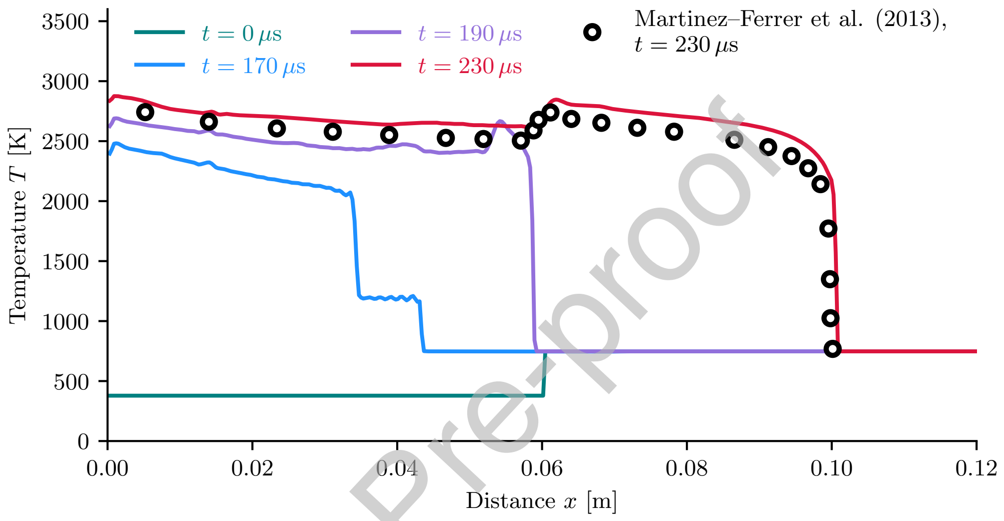


Figure 12: Temperature of a one-dimensional detonation in a H_2–Ar mixture, simulated using MFC with Pyrometheus-generated Fortran code.

5.5. Pyrometheus-enabled MFC Simulations of Flame–Vortex Interactions

To demonstrate our capability for simulating thermochemistry and transport, we simulate a two-dimensional premixed hydrogen–air flame perturbed by two superimposed vortices. This simulation closely follows the configuration of Yu et al. [55] and Bando et al. [56]. In contrast with the previous one-dimensional detonation, this case incorporates mixture-averaged mass diffusion fluxes and corresponding transport coefficients (A.18), (A.20) and (A.23). Reactions are modeled using the hydrogen subset of the GRI-3.0 mechanism. Fifth-order accurate WENO reconstruction and an HLLC approximate Riemann solver are used to compute the fluxes, following [40].

Figure 13 shows the flow configuration. The computational domain is $[0, 8\,\mathrm{mm}] \times [0, 16\,\mathrm{mm}]$, discretized with $(N_x, N_y) = (512, 1026)$ grid cells. Boundary conditions in x are periodic; inflow and outflow boundary conditions are imposed at $y = 0$ and $y = 2L$. The simulation is initiated with an extruded one-dimensional H_2–air flame with equivalence ratio $\phi = 0.6$. The flame front is positioned 4.7 mm from the right of the inflow boundary. The vortices are superimposed 2.7 mm from the inflow, symmetrically arranged about $x = 0$ with 2 mm between their centers.

The flame–vortex interaction is shown in fig. 13. We compare the flame shape with the results of Bando et al. [56]. Close agreement in flame shape is observed. Differences in absolute spatial position are due to the lack of sufficient geometric information in the reference solution. Still, the

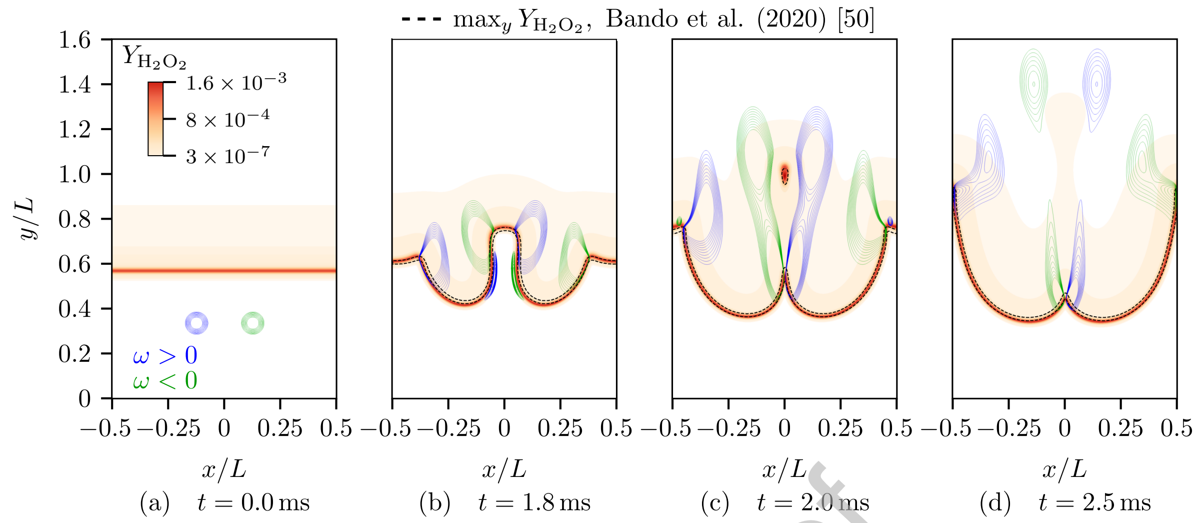


Figure 13: Flow configuration showing the evolution of flame–vortex interaction, represented by H_2O_2 mass fraction and vorticity (blue and green contours, for positive and negative values) at (a) $t = 1.8$ ms, (b) $t = 2.0$ ms, and (c) $t = 2.5$ ms. The dashed line denotes the flame front from Bando et al. [56].

vortex structures observed in our simulation also show good agreement with the results of Yu et al. [55], where similar features in the vorticity field are reported. Vortex structures downstream of the flame arise due to flame curvature, which induces baroclinicity. The agreement in flame and flow structures supports the correctness of the implementation.

As stated, these simulations necessitate the computation of transport coefficients, which are provided to MFC by Pyrometheus. This verification is in contrast to the simulation of one-dimensional detonation shown in fig. 12. For this hydrogen–air mechanism, the cost of calculating mixture-averaged transport properties is $\approx 4.2\%$ of total run time, which is $\approx 25\%$ of the time that it takes to assemble the corresponding mass diffusion fluxes. This cost balance indicates the efficiency of our Pyrometheus implementation for transport.

The simulations presented in this section show how Pyrometheus addresses the challenges associated with coupling exascale flow solvers with computational models for chemical kinetics, thermodynamics, and transport. In contrast with pre-compiled libraries, Pyrometheus places the computation of the thermochemistry and transport at the same compilation level as the flow solver. MFC then offloads the fully coupled flow–combustion computation to GPUs using compiler directives, achieving near-roofline performance (fig. 11), which has not been demonstrated by pre-compiled libraries.

6. Summary of Principal Features and Scope, with Additional Discussion

A computational approach for combustion thermochemistry has been developed. The strategy is based on a symbolic representation, with the only assumption that the computation will eventually be array-based. Computational details, such as loop patterns and parallelism, are added incrementally. With each addition, the representation is lowered closer to a case-specific target through code generation. The approach is sufficiently general to achieve multiple conflicting goals that are otherwise best served by disparate data structures.

Two main compilation pipelines are presented. The first preserves a control environment in Python throughout the calculation, and the second saves the code to a file. This setup offers avenues for both modern and traditional high-performance computing. It enables Python to continue expanding into HPC, particularly in scientific machine learning, while supporting existing HPC solvers.

The achieved capabilities were demonstrated. The generated code can simulate, for example, homogeneous reactors and flames. It is compatible with AD libraries for sensitivity analysis, and can be offloaded to GPUs. Every compilation unit is exposed to the user, allowing the code to be manipulated to improve performance. This property also enables integration into CFD codebases such as MFC, as the generated code can be bundled with the flow solver for compilation and directive-based loop optimization. This strategy contrasts with pre-compiled thermochemistry libraries such as Cantera, which only expose parameters through an interface.

Our abstractions face the potential limitations of any domain-specific language, such as data-dependent control flow and non-differentiability. According to the principle of separation of concerns, the array libraries are responsible for handling these issues appropriately. The generated code is designed with this choice in mind. The formulation presents two instances where data-dependent control flow can arise: temperature-dependent conditionals for thermodynamic variables and the temperature inversion. We assume the array libraries provide non-vectorized-select `where` for conditionals and, indeed, NumPy, JAX, Torch, and Pytato do. The temperature inversion requires inner loop iterations. Pyrometheus addresses the inner-loop logistics via inner-loop abstractions (e.g., in JAX) or by fixing the number of Newton iterations (such as for Pytato, where inner-loop abstractions are unavailable). Non-differentiability is less of a concern because standard chemical kinetics is differentiable, without singularities. Users seeking to implement novel rates with non-differentiable forms must ensure that the AD package supports such functions.

The Pyrometheus approach can impact any large-scale combustion simulation because of the properties and capabilities of the generated code. Application areas include turbulent combustion simulations, dynamic adaptive chemistry, machine learning of chemical source terms, model reduction, and uncertainty quantification (UQ). For example, for UQ, additional array axes can represent parametric realizations. An early version of this approach has already been used for chemical model reduction with UQ [57].

Beyond MFC, our implementation serves as the thermochemistry engine for multiple compressible-flow solvers. The MIRGE-Com solver [58] is rooted in the same concepts and implements discontinuous Galerkin discretizations through symbolic manipulation to maintain separation of concerns. The PyFlowCL package [59] implements data-driven sub-grid-scale models with PyTorch, enabled by Pyrometheus differentiable generated code. The fully-differentiable JAX-Fluids solver [60, 61] provides Pyrometheus as an alternative to its native implementation of thermochemistry. PlasCom2 [62–64], a C++ plasma-coupled combustion solver, uses an early version of our implementation.

7. Conflict of Interest

The authors have no known conflicts of interest associated with this publication, and there has been no significant financial support for this work that could have influenced its outcome.

Acknowledgments

This material is based in part upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Numbers DE-NA0002374 and DE-NA0003963. The authors gratefully acknowledge Cory Mikida, Michael T. Campbell, and Dr. Tulio Ricciardi for assisting in the implementation. ECG gratefully acknowledges Prof. Andreas Klöckner, Dr. Matthias Diener, and Matthew J. Smith (University of Illinois) for extensive discussion on code generation, array semantics, and GPU computing.

SHB gratefully acknowledges support from the U.S. Department of Defense, Office of Naval Research under grant number N00014-24-1-2094. Some computations were performed on the Tioga, Tuolumne, and El Capitan clusters at Lawrence Livermore National Laboratory's Livermore Computing facility. This research also used resources of the Oak Ridge Leadership Computing Facility—Frontier, Summit, and Wombat—at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725 (allocation CFD154, PI Bryngelson). In addition, this work used Delta at the National Center for Supercomputing Applications and Bridges2 at the Pittsburgh Supercomputing Center through allocations PHY210084 and PHY240200 (PI Bryngelson) from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296.

Appendix A. Combustion Thermochemistry Formulation & Methods

Appendix A.1. Governing Equations

A homogeneous adiabatic isochoric reactor is sufficient for introducing thermochemistry. The chemical composition is characterized by the mass fractions $\{Y_i\}_{i=1}^N$ of N species, which are governed by

$$\rho\frac{dY_k}{dt} = \overbrace{W_k\dot{\omega}_k}^{\equiv S_k}, \qquad Y_k(0) = Y_k^0, \qquad k = 1, \ldots, N, \tag{A.1}$$

where W_k is the molecular weight of the k^{th} species, $\dot{\omega}_k = \dot{\omega}_k(Y_1, \ldots, Y_N, T)$ its net production rate (with T the temperature), ρ the density, and $\{Y_k^0\}_{k=1}^N$ the initial condition. The right-hand side S_k is typically referred to as the chemical source terms. The pressure p is obtained from the equation of state

$$p = \rho\mathscr{R}T/W, \tag{A.2}$$

where

$$W = \left(\sum_{i=1}^N Y_i/W_i\right)^{-1} \tag{A.3}$$

is the mixture molecular weight and $\mathscr{R}$ the universal gas constant. The focus of this work is the implementation and evaluation of $\dot{\omega}_k$, so corresponding detailed expressions are provided next. Generalization to include advection and diffusion of the chemical species is briefly discussed in section Appendix A.2.

Net production rates $\{\dot{\omega}_k\}_{k=1}^N$ represent changes in composition due to chemical reactions, which can be expressed as

$$\sum_{i=1}^N \nu'_{ij}\mathcal{S}_i \rightleftharpoons \sum_{i=1}^N \nu''_{ij}\mathcal{S}_i, \qquad j = 1, \ldots, M, \tag{A.4}$$

where ν'_{ij} and ν''_{ij} are the forward and reverse stoichiometric coefficients of species $\mathcal{S}_i$ in the j^{th} reaction. Per (A.4), species $\mathcal{S}_i$ can only be destroyed or produced in proportion to ν'_{ij} or ν''_{ij} in the

j^{th} reaction. Thus, $\{\dot{\omega}_i\}_{i=1}^N$ are linear combinations of the reaction rates of progress R_j,

$$\dot{\omega}_i = \sum_{j=1}^{M} \nu_{ij} R_j, \qquad i = 1, \dots, N, \tag{A.5}$$

where

$$\nu_{ij} = \nu''_{ij} - \nu'_{ij} \tag{A.6}$$

is the net stoichiometric coefficient of the i^{th} species in the j^{th} reaction. The law of mass action gives the rates of progress,

$$R_j = k_j(T) \left[\prod_{\ell=1}^{N} C_\ell^{\nu'_{\ell j}} - \frac{1}{K_j(T)} \prod_{k=1}^{N} C_k^{\nu''_{kj}} \right], \qquad j = 1, \dots, M, \tag{A.7}$$

where

$$C_k = \rho Y_k / W_k, \qquad k = 1, \dots, N, \tag{A.8}$$

are the species molar concentrations, $k_j(T)$ is the rate coefficient of the j^{th} reaction and $K_j(T)$ its equilibrium constant. Depending on the reaction, the rate coefficient $k_j(T)$ may take different forms (and even become a function of pressure). Its simplest form is the Arrhenius expression,

$$k_j(T) = A_j T^{b_j} \exp\left(-\theta_{a,j}/T\right), \qquad j = 1, \dots, M, \tag{A.9}$$

where A_j is a constant (often called pre-exponential), b_j is the temperature exponent, and $\theta_{a,j}$ is the activation temperature. Some reactions can display concentration dependence in their rate of progress, which is modeled via more complex expressions for their rate coefficient [65]. Pyrometheus includes both these and additional potential pressure dependencies.

The equilibrium constant K_j in (A.7) is evaluated through equilibrium thermodynamics

$$K_j(T) = \left(\frac{p_0}{\mathscr{R}T}\right)^{\sum_{i=0}^{\nu_{ij}}} \exp\left(-\sum_{i=1}^{N} \frac{\nu_{ij}\hat{g}_i^0(T)}{\mathscr{R}T}\right), \qquad j = 1, \dots, M, \tag{A.10}$$

where $p_0 = 1\,\text{atm}$, $\mathscr{R}$ is the universal gas constant, and

$$\hat{g}_k^0(T) = \hat{h}_k(T) - T\,\hat{s}_k^0(T), \qquad k = 1, \dots, N \tag{A.11}$$

is the species standard Gibbs functions (in J/mol), with $\{\hat{h}_k\}_{k=1}^N$ and $\{\hat{s}_k^0\}_{k=1}^N$ the species enthalpies and standard entropies (in J/mol and J/mol $\cdot$ K). These are modeled using NASA polynomials [66]

$$\frac{\hat{h}_i}{\mathscr{R}T} = \frac{\alpha_h}{T} + \sum_{m=1}^{5} \frac{\alpha_m}{m} T^{m-1}, \tag{A.12}$$

$$\frac{\hat{s}_i^0}{\mathscr{R}} = \alpha_s + \alpha_h \log T + \sum_{m=2}^{5} \frac{\alpha_m}{m-1} T^m, \tag{A.13}$$

where α_h, α_s, and $\{\alpha_m\}_{m=1}^5$ are the fit coefficients.

The presented combustion thermochemistry formulation needs to be solved numerically, even for small mechanisms such as hydrogen–air combustion. Corresponding numerical methods are standard but judiciously selected to address numerical stiffness that arises as a consequence of the broad range of chemical time scales involved; these are summarized in section Appendix A.3.2. As will be shown, these methods use chemical Jacobians; therefore, the corresponding implementations used in section 4 employ the proposed computational approach, which enables automatic differentiation and is presented in full in section 2.

Appendix A.2. Extensions to Flow & Transport

For problems that involve flow, (A.1) generalizes to

$$\frac{\partial}{\partial t}(\rho Y_k) + \frac{\partial}{\partial x_j}(\rho Y_k u_j) + \frac{\partial \varphi_{kj}}{\partial x_j} = S_k, \qquad k = 1, \ldots, N, \tag{A.14}$$

with appropriate boundary conditions. In (A.14), $\mathbf{u} = \{u_j\}_{j=1}^3$ is the flow velocity, and $\boldsymbol{\varphi}_k = \{\varphi_{kj}\}_{j=1}^3$ the mass diffusion flux of the kth species. This term must be modeled; there are multiple options, a typical choice being the mixture-averaged approximation

$$\varphi_{kj} = \varphi^*_{kj} + \varphi^c_{kj} \tag{A.15}$$

where

$$\varphi^*_{kj} = -\rho \mathscr{D}_{(k),m} \frac{W_{(k)}}{W} \frac{\partial X_k}{\partial x_j} \tag{A.16}$$

is the mixture-averaged approximation, with $X_k = W Y_k / W_{(k)}$ the mole fraction of the kth species, and

$$\varphi^c_{kj} = -Y_k \sum_{n=1}^{N} \varphi^*_{nj} \tag{A.17}$$

is a correction to ensure mass conservation. In (A.15),

$$\mathscr{D}_{k,m} = \frac{W - X_{(k)} W_k}{W} \left(\sum_{j \neq k} \frac{X_j}{\mathscr{D}_{kj}} \right)^{-1} \tag{A.18}$$

is the mixture-averaged diffusivity of species k, where the binary diffusivities $\mathscr{D}_{kj}$ are often approximated using polynomials:

$$\mathscr{D}_{ij} = \sqrt{T} \sum_{m=0}^{4} a_{i,j,m} (\log T)^m . \tag{A.19}$$

In (A.14), the flow velocity $\mathbf{u}$ is obtained from the full set of conservation equations for mass, momentum, and energy density, provided elsewhere [67]. These involve two additional transport properties that must be modeled: the mixture viscosity μ and thermal conductivity κ. In the mixture-averaged formulation, the viscosity is

$$\mu = \sum_{k=1}^{N} \frac{X_k \mu_k}{\sum_{j=1}^{N} \Phi_{kj} X_j}, \tag{A.20}$$

where

$$\Phi_{kj} = \frac{\left[1 + \sqrt{\left(\frac{\mu_k}{\mu_j}\sqrt{W_j/W_k}\right)}\right]^2}{\sqrt{8\left(1 + W_k/W_j\right)}}, \tag{A.21}$$

and

$$\mu_i = \sqrt{T}\left[\sum_{m=0}^{4} b_{i,m}\left(\log T\right)^m\right]^2 \tag{A.22}$$

approximates the viscosity of species k. The coefficients $b_{i,m}$ are obtained via least-squares fitting to collision integrals [68]. The thermal conductivity is

$$\kappa = \frac{1}{2}\left[\sum_{k=1}^{N} X_k \kappa_k + \left(\sum_{k=1}^{N} \frac{X_k}{\kappa_k}\right)^{-1}\right], \tag{A.23}$$

where the conductivity of individual species is also approximated using polynomials

$$\kappa_i = \sqrt{T}\sum_{m=0}^{4} c_{i,m}\left(\log T\right)^m . \tag{A.24}$$

Material coefficients $\mathscr{D}_{k,m}$ (A.18), μ (A.20), and κ (A.23) are provided by the computational design of section 2, extending our computational capabilities to flow simulations.

Appendix A.3. Numerical Methods for Combustion Thermochemistry

We next introduce numerical methods to solve the formulation in the preceding section. Throughout their presentation, aspects that benefit from the proposed computational design of section 2 are highlighted. This section shows how to compute the temperature by inverting the internal energy via Newton iteration. We then discuss time integration and discretization schemes for thermochemistry systems (A.1) and (A.14).

Appendix A.3.1. Temperature Inversion

Following section Appendix A.1, the net production rates (A.5) are expressed in terms of density ρ, temperature T, and mass fractions $\vec{Y} \equiv \{Y_i\}_{i=1}^{N}$. Any $N+2$ variables, not just $\{\rho,\, T,\, \vec{Y}\}$, can define the state. If the temperature is not explicitly solved for, such as in simulations of compressible flows, transported variables are the density ρ, the internal energy e, and the mass fractions $\vec{Y}$, so T must be inverted from $\{\rho,\, e,\, \vec{Y}\}$. The internal energy per unit mass (in J/kg) of an ideal gas mixture is

$$e = \vec{Y} \cdot \vec{e}(T), \tag{A.25}$$

where $\vec{e} = \{e_k(T)\}_{k=1}^{N}$ are the species internal energies

$$e_k(T) = \frac{\hat{h}_k(T) - \mathscr{R}T}{W_k}, \qquad k = 1, \ldots, N, \tag{A.26}$$

with $\hat{h}_k$ species molar enthalpy (in J/kmol). Thus, given e, (A.25) provides an implicit equation for the temperature T. In Pyrometheus, (A.25) is inverted using a Newton method

$$T_{m+1} = T_m + \Delta T_m \tag{A.27}$$

where the subscript indicates the Newton iteration, and

$$\Delta T_m = -\left[\vec{Y} \cdot \frac{\partial \vec{e}}{\partial T}(T_m)\right]^{-1} \left[\vec{Y} \cdot \vec{e}(T_m) - e\right] = -\frac{1}{c_v(T_m, \vec{Y})} \left[\vec{Y} \cdot \vec{e}(T_m) - e\right], \tag{A.28}$$

with

$$c_v \equiv \frac{\partial e}{\partial T} = c_p - \mathscr{R} \tag{A.29}$$

as the mixture's specific heat capacity at constant volume. In (A.29),

$$c_p = \sum_{i=1}^{N} c_{p,i}(T) Y_i \tag{A.30}$$

is the mixture specific heat capacity at constant pressure; $c_{p,i}$, like the enthalpy and entropy of each species, is modeled with NASA polynomials

$$\frac{c_{p,i}}{\mathscr{R}} = \sum_{m=1}^{4} m \, \frac{\alpha_m}{m} T^{m-1}. \tag{A.31}$$

In some cases, such as those involving characteristic boundary conditions or simulation of low-Mach number combustion, it is advantageous to use enthalpy h rather than internal energy. For these cases, the temperature can be analogously inverted from h and $\vec{Y}$ with minor modifications to the preceding Newton procedure.

Appendix A.3.2. Time Integration & Spatial Discretization

The stiff ODE system (A.1) is discretized via the Crank–Nicolson method

$$\underbrace{Y_k^{m+1} - Y_k^m - \frac{\Delta t}{2}\left(S_k^{m+1} + S_k^m\right)}_{\equiv \mathscr{F}_k} = 0, \tag{A.32}$$

where the superscript indicates the time step. The scheme (A.32) is nonlinear through S_k^{m+1}, so Y_k^{m+1} is obtained via Newton iterations

$$Y_{k,\,n+1}^{m+1} = Y_{k,\,n}^{m+1} + \Delta Y_{k,\,n}^{m+1}, \tag{A.33}$$

where the second subscript indicates the n^{th} Newton iteration and $\Delta Y_{k,\,n}^{m+1}$ are the Newton updates. These are the solutions to

$$\mathscr{J}_{jk,\,n}^{m+1} \, \Delta Y_{k,\,n}^{m+1} = -\mathscr{F}_{k,\,n}. \tag{A.34}$$

where

$$\mathscr{J}_{jk,\,n}^{m+1} \equiv \frac{\partial \mathscr{F}_{j,\,n}}{\partial Y_{k,\,n}^{m+1}} = \delta_{jk} - \frac{\Delta t}{2} J_{jk,\,n}^{m+1}, \tag{A.35}$$

with

$$J_{jk} \equiv \frac{\partial S_j}{Y_k}, \qquad j = 1, \ldots, \; N, \;\; \text{and} \;\; k = 1, \ldots, N \tag{A.36}$$

the chemical Jacobian (evaluated at $\{Y_{k,\,n}\}_{k=1}^{N}$), and δ_{jk} the Kronecker delta. The Newton method for each time step is initialized with $Y_{k,\,0}^{m+1} = Y_k^m$, $k = 1, \ldots, N$.

Simulations of reactors with flow, such as flame, entail spatial discretization of (A.14). As with time marching, Pyrometheus is independent of the choice of the discretization scheme. In subsequent demonstrations of section 4.2.1, we approximate spatial derivatives in (A.14) using standard fourth-order central stencils in the interior and one-sided second-order stencils near the boundaries. Additionally, to suppress high-wavenumber oscillations, an eight-order filter is applied after every time step.

Declaration of interests

Esteban Cisneros Garibay, Spencer H. Bryngelson, Jonathan Freund reports financial support was provided by Department of Energy. Spencer H. Bryngelson reports financial support was provided by US Department of Defense. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] F. Alexander, A. Almgren, J. Bell, A. Bhattacharjee, J. Chen, P. Colella, D. Daniel, J. DeSlippe, L. Diachin, E. Draeger, A. Dubey, T. Dunning, T. Evans, I. Foster, M. Francois, T. Germann, et al., Exascale applications: Skin in the game, Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences **378** (2020) 20190056.

[2] M. T. H. de Frahan, J. S. Rood, M. S. Day, H. Sitaraman, S. Yellapantula, B. A. Perry, R. W. Grout, A. Almgren, W. Zhang, J. B. Bell, J. H. Chen, PeleC: An adaptive mesh refinement solver for compressible reacting flows, The International Journal of High Performance Computing Applications **37** (2022) 115–131.

[3] U. Maas, S. B. Pope, Simplifying chemical kinetics: Intrinsic low-dimensional manifolds in composition space, Combustion and Flame **88** (1992) 239–264.

[4] D. G. Goodwin, H. K. Moffat, I. Schoegl, R. L. Speth, B. W. Weber, Cantera: An object-oriented software toolkit for chemical kinetics, thermodynamics, and transport processes, 2022. Version 2.6.0.

[5] R. J. Kee, F. M. Rupley, E. Meeks, J. A. Miller, CHEMKIN-III: A FORTRAN chemical kinetics package for the analysis of gas-phase chemical and plasma kinetics, 1996.

[6] C. Safta, H. N. Najm, O. Knio, TChem – A software toolkit for the analysis of complex kinetic models, Sandia Report, SAND2011-3282 (2011).

[7] J. Capecelatro, D. J. Bodony, J. B. Freund, Adjoint-based sensitivity and ignition threshold mapping in a turbulent mixing layer, Combustion Theory and Modelling **23** (2019) 147–179.

[8] K. Braman, T. A. Oliver, V. Raman, Adjoint-based sensitivity analysis of flames, Combustion Theory and Modelling **19** (2015) 29–56.

[9] X. Zhao, Y. Tao, T. Lu, H. Wang, Sensitivities of direct numerical simulations to chemical kinetic uncertainties: spherical flame kernel evolution of a real jet fuel, Combustion and Flame **209** (2019) 117–132.

[10] S. H. Lam, D. A. Goussis, Understanding complex chemical kinetics with computational singular perturbation, Symposium (International) on Combustion **22** (1989) 931–941.

[11] M. Valorani, S. Paolucci, The G-Scheme: A framework for multi-scale adaptive model reduction, Journal of Computational Physics **228** (2009) 4665–4701.

[12] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, C. S. Woodward, Sundials, ACM Transactions on Mathematical Software **31** (2005) 363–396.

[13] J. F. MacArt, M. E. Mueller, Semi-implicit iterative methods for low mach number turbulent reacting flows: Operator splitting versus approximate factorization, Journal of Computational Physics **326** (2016) 569–595.

[14] S. Rao, W. Zhou, W. Han, Y. Tang, X. Xu, An adaptive implicit time-integration scheme for stiff chemistry based on Jacobian tabulation method, Combustion and Flame **274** (2025) 113997.

[15] K. E. Niemeyer, N. J. Curtis, C.-J. Sung, pyJac: Analytical Jacobian generator for chemical kinetics, Computer Physics Communications **215** (2017) 188–203.

[16] A. Griewank, A. Walther, Introduction to automatic differentiation, PAMM **2** (2003) 45–49.

[17] N. J. Curtis, K. E. Niemeyer, C.-J. Sung, Using SIMD and SIMT vectorization to evaluate sparse chemical kinetic Jacobian matrices and thermochemical source terms, Combustion and Flame **198** (2018) 186–204.

[18] M. Hassanaly, N. T. Wimer, A. Felden, L. Esclapez, J. Ream, M. T. H. de Frahan, J. Rood, M. Day, Symbolic construction of the chemical jacobian of quasi-steady state (QSS) chemistries for exascale computing platforms, Combustion and Flame **270** (2024) 113740.

[19] F. Sewerin, S. Rigopoulos, A methodology for the integration of stiff chemical kinetics on gpus, Combustion and Flame **162** (2015) 1375–1394.

[20] R. Mao, M. Zhang, Y. Wang, H. Li, J. Xu, X. Dong, Y. Zhang, Z. X. Chen, An integrated framework for accelerating reactive flow simulation using gpu and machine learning models, Proceedings of the Combustion Institute **40** (2024) 105512.

[21] M. Bauer, S. Treichler, A. Aiken, Singe, in: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, 2014, pp. 119–130. doi:10.1145/2555243.2555258.

[22] W. Ji, X. Su, B. Pang, S. J. Cassady, A. M. Ferris, Y. Li, Z. Ren, R. Hanson, S. Deng, Arrhenius.jl: A differentiable combustion simulation package, arXiv preprint arXiv:2107.06172 (2021).

[23] S. Barwey, V. Raman, A neural network-inspired matrix formulation of chemical kinetics for acceleration on GPUs, Energies **14** (2021) 2710.

[24] Z. J. Weiner, Stencil solvers for PDEs on GPUs: An example from cosmology, Computing in Science & Engineering **23** (2021) 55–64.

[25] K. G. Kulkarni, Domain-Specific Code Transformations for Computational Science based on the Polyhedral Model, Ph.D. thesis, University of Illinois at Urbana–Champaign, 2023.

[26] A. Kloeckner, M. Wala, I. Fernando, K. Kulkarni, A. Fikl, Z. Weiner, D. Kempf, D. A. Ham, L. Mitchell, L. C. Wilcox, M. Diener, P. Kapyshin, R. Raksi, T. H. Gibson, pymbolic, 2022. URL: https://github.com/inducer/pymbolic.

[27] A. V. Aho, R. Sethi, J. D. Ullman, Compilers: principles, techniques, and tools, Addison-Wesley Longman Publishing Co., Inc., USA, 1986.

[28] L. A. Barba, A. Klockner, P. Ramachandran, R. Thomas, Scientific computing with Python on high-performance heterogeneous systems, Computing in Science & Engineering **23** (2021) 5–7.

[29] J. P. Slotnick, A. Khodadoust, J. Alonso, D. Darmofal, W. Gropp, E. Lurie, D. J. Mavriplis, CFD vision 2030 study: A path to revolutionary computational aerosciences, Technical Report 20140003093, NASA, 2014.

[30] T. Betcke, M. W. Scroggs, Designing a high-performance boundary element library with OpenCL and Numba, Computing in Science & Engineering **23** (2021) 18–28.

[31] A. Klöckner, Loo.py: Transformation-based code generation for GPUs and CPUs, in: Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, ACM, 2014, pp. 82–87.

[32] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, C. S. Woodward, Sundials, ACM Transactions on Mathematical Software **31** (2005) 363–396.

[33] Y. Shi, W. H. Green, H.-W. Wong, O. O. Oluwole, Accelerating multi-dimensional combustion simulations using gpu and hybrid explicit/implicit ode integration, Combustion and Flame **159** (2012) 2388–2397.

[34] H. N. Najm, P. S. Wyckoff, O. M. Knio, A semi-implicit numerical scheme for reacting flow, Journal of Computational Physics **143** (1998) 381–402.

[35] P. Saxena, F. A. Williams, Testing a small detailed chemical-kinetic mechanism for the combustion of hydrogen and carbon monoxide, Combustion and Flame **145** (2006) 316–323.

[36] M. D. Renzo, L. Fu, J. Urzay, Htr solver: An open-source exascale-oriented task-based multi-gpu high-order code for hypersonic aerothermodynamics, Computer Physics Communications **255** (2020) 107262.

[37] T. Poinsot, S. Lelef, Boundary conditions for direct simulations of compressible viscous flows, Journal of Computational Physics **101** (1992) 104–129.

[38] S. Williams, A. Waterman, D. Patterson, Roofline: An insightful visual performance model for multicore architectures, Communications of the ACM **52** (2009) 65–76.

[39] S. H. Bryngelson, A. Charalampopoulos, T. P. Sapsis, T. Colonius, A Gaussian moment method and its augmentation via LSTM recurrent neural networks for the statistics of cavitating bubble populations, International Journal of Multiphase Flow **127** (2020) 103262.

[40] B. Wilfong, H. Le Berre, A. Radhakrishnan, A. Gupta, D. Vaca-Revelo, D. Adam, H. Yu, H. Lee, J. R. Chreim, M. Carcana Barbosa, Y. Zhang, E. Cisneros-Garibay, A. Gnanaskandan, M. Rodriguez Jr., R. D. Budiardja, S. Abbott, T. Colonius, S. H. Bryngelson, MFC 5.0: An exascale many-physics flow solver, arXiv:2503.07953 (2025).

[41] A. K. Kapila, R. Menikoff, J. B. Bdzil, S. F. Son, D. S. Stewart, Two-phase modeling of deflagration-to-detonation transition in granular materials: Reduced equations, Physics of Fluids **13** (2001) 3002–3024.

[42] M. Baer, J. Nunziato, A two-phase mixture theory for the deflagration-to-detonation transition (DDT) in reactive granular materials, International Journal of Multiphase Flow **12** (1986) 861–889.

[43] S. H. Bryngelson, K. Schmidmayer, V. Coralic, J. C. Meng, K. Maeda, T. Colonius, MFC: An open-source high-order multi-component, multi-phase, and multi-scale compressible flow solver, Computer Physics Communications **266** (2021) 107396.

[44] A. Charalampopoulos, S. H. Bryngelson, T. Colonius, T. P. Sapsis, Hybrid quadrature moment method for accurate and stable representation of non-Gaussian processes and their dynamics, Philosophical Transactions of the Royal Society A **380** (2022).

[45] S. H. Bryngelson, T. Colonius, Simulation of humpback whale bubble-net feeding models, Journal of the Acoustical Society of America **147** (2020) 1126–1135.

[46] D. Rossinelli, B. Hejazialhosseini, P. Hadjidoukas, C. Bekas, A. Curioni, A. Bertsch, S. Futral, S. J. Schmidt, N. A. Adams, P. Koumoutsakos, 11 PFLOP/s simulations of cloud cavitation collapse, in: SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE, 2013, pp. 1–13.

[47] K. Schmidmayer, S. H. Bryngelson, T. Colonius, An assessment of multicomponent flow models and interface capturing schemes for spherical bubble dynamics, Journal of Computational Physics **402** (2020) 109080.

[48] A. Radhakrishnan, H. Le Berre, B. Wilfong, J.-S. Spratt, M. Rodriguez Jr., T. Colonius, S. H. Bryngelson, Method for portable, scalable, and performant GPU-accelerated simulation of multiphase compressible flow, Computer Physics Communications **302** (2024) 109238.

[49] A. Radhakrishnan, H. Le Berre, S. H. Bryngelson, Scalable GPU accelerated simulation of multiphase compressible flow, in: The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC), Dallas, TX, USA, 2022, pp. 1–3.

[50] B. Wilfong, A. Radhakrishnan, H. A. Le Berre, S. Abbott, R. D. Budiardja, S. H. Bryngelson, OpenACC offloading of the MFC compressible multiphase flow solver on AMD and NVIDIA GPUs, in: SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2024, pp. 1923–1933.

[51] W. Elwasif, S. Bastrakov, S. H. Bryngelson, M. Bussmann, S. Chandrasekaran, F. Ciorba, M. A. Clark, A. Debus, W. Godoy, N. Hagerty, J. Hammond, D. Hardy, J. A. Harris, O. Hernandez, B. Joo, S. Keller, P. Kent, H. Le Berre, D. Lebrun-Grandie, E. MacCarthy, V. G. M. Vergara, B. Messer, R. Miller, S. Oral, J.-G. Piccinali, A. Radhakrishnan, O. Simsek, F. Spiga, K. Steiniger, J. Stephan, J. E. Stone, C. Trott, R. Widera, J. Young, Early application experiences on a modern GPU-accelerated Arm-based HPC platform, in: HPC Asia '23, International Workshop on Arm-based HPC: Practice and Experience (IWAHPCE), Singapore, 2023, pp. 35–49.

[52] B. Wilfong, A. Radhakrishnan, H. Le Berre, D. J. Vickers, T. Prathi, N. Tselepidis, B. Dorschner, R. Budiardja, B. Cornille, S. Abbott, F. Schäfer, S. H. Bryngelson, Simulating

many-engine spacecraft: Exceeding 1 quadrillion degrees of freedom via information geometric regularization, arXiv preprint arXiv:2505.07392 (2025).

[53] R. P. Fedkiw, B. Merriman, S. Osher, High accuracy numerical methods for thermally perfect gas flows with chemistry, Journal of Computational Physics **132** (1997) 175–190.

[54] P. J. M. Ferrer, R. Buttay, G. Lehnasch, A. Mura, A detailed verification procedure for compressible reactive multicomponent Navier–Stokes solvers, Computers & Fluids **89** (2014) 88–110.

[55] R. Yu, J. Yu, X.-S. Bai, An improved high-order scheme for DNS of low Mach number turbulent reacting flows based on stiff chemistry solver, Journal of Computational Physics **231** (2012) 5504–5521.

[56] K. Bando, M. Sekachev, M. Ihme, Comparison of algorithms for simulating multi-component reacting flows using high-order discontinuous Galerkin methods, in: AIAA SciTech Forum, American Institute of Aeronautics and Astronautics, Orlando, FL, 2020. doi:10.2514/6.2020-1751.

[57] E. Cisneros-Garibay, C. Pantano, J. B. Freund, Accounting for uncertainty in RCCE species selection, Combustion and Flame **208** (2019) 219–234.

[58] T. R. Ricciardi, M. Franco, K. Tang, H. X. Varona, F. Panerai, G. S. Elliott, J. B. Freund, Combustion heat flux estimation for design of carbon fiber-based thermal protection systems, in: AIAA AVIATION FORUM AND ASCEND 2024, American Institute of Aeronautics and Astronautics, 2024. doi:10.2514/6.2024-3813.

[59] X. Liu, J. F. MacArt, Adjoint-based machine learning for active flow control, Physical Review Fluids **9** (2024) 013901.

[60] D. A. Bezgin, A. B. Buhendwa, N. A. Adams, Jax-fluids: A fully-differentiable high-order computational fluid dynamics solver for compressible two-phase flows, Computer Physics Communications **282** (2023) 108527.

[61] D. A. Bezgin, A. B. Buhendwa, N. A. Adams, Jax-fluids 2.0: Towards hpc for differentiable cfd of compressible two-phase flows, Computer Physics Communications **308** (2025) 109433.

[62] C. Mikida, A. Klöckner, D. Bodony, Multi-rate time integration on overset meshes, Journal of Computational Physics **396** (2019) 325–346.

[63] S. R. Murthy, D. J. Bodony, Resolvent analysis of a biconical tactical jet nozzle, in: 28th AIAA/CEAS Aeroacoustics 2022 Conference, American Institute of Aeronautics and Astronautics, 2022. doi:10.2514/6.2022-2969.

[64] B. Vollmer, S. R. Murthy, D. J. Bodony, Revisiting the boundary conditions for unsteady flows adjacent to rigid and dynamic solid walls, in: 28th AIAA/CEAS Aeroacoustics 2022 Conference, American Institute of Aeronautics and Astronautics, 2022. doi:10.2514/6.2022-2895.

[65] J. Troe, Fall-off curves of unimolecular reactions, Berichte der Bunsengesellschaft für physikalische Chemie **78** (1974) 478–488.

[66] B. J. McBride, NASA Glenn Coefficients for Calculating Thermodynamic Properties of Individual Species, National Aeronautics and Space Administration, John H. Glenn Research Center, 2002.

[67] E. Cisneros-Garibay, C. Pantano, J. B. Freund, Flow and combustion in a supersonic cavity flameholder, AIAA Journal **60** (2022) 4566–4577.

[68] R. J. Kee, M. E. Coltrin, P. Glarborg, Chemically Reacting Flow, John Wiley & Sons, Inc., 2003.