PERFORM Documentation

Christopher R. Wentland

Feb 01, 2022

USER GUIDE

PERFORM is a combination 1D compressible reacting flow solver and modular reduced-order model (ROM) framework, designed to provide a simple, easy-to-use testbed for members of the ROM community to quickly prototype and test new methods on challenging (yet computationally-manageable) reacting flow problems. The baseline solver is based off the **General Equations and Mesh Solver (GEMS)**[DXSM04], a Fortran 3D reacting flow solver originally developed by Li Ding and Guoping Xia at Purdue University. This code has obviously been simplified to one-dimensional flows, but it aims to be a) open-source, freely available to copy and modify for everyone, and b) much more approachable for researchers outside the high-performance computing community. We hope that this tool lowers the barrier to entry for researchers from a variety of fields to develop novel ROM methods and benchmark them against an interesting set of reacting flow configurations.

This documentation serves as a reference for users to get up and running with **PERFORM**, understand the underlying solver (albeit very superficially), and become acquainted with implementing new ROM routines and evaluating their performance. The mathematical theory behind the solver and ROM methods is only touched on briefly; we refer the interested reader to compile the solver documentation from `perform/doc/solver_theory/main.tex` for more details. This website and the theory documentation is very much a work-in-progress and will be updated regularly.

CHAPTER ONE

DATA-DRIVEN MODELING FOR COMPLEX FLUID PHYSICS

This code also serves as a companion code for the workshop on *Data-driven Modeling for Complex Fluid Physics*, presented at the 2021 AIAA SciTech Forum. Details on the workshop and several proposed benchmark cases can be found at the workshop website. The proposed benchmark cases, among others, are provided in `perform/examples/` . Those interested in keeping up-to-date on workshop activities can send an email to `romworkshop [at] gmail [dot] com` requesting to be added to the workshop mailing list.

CHAPTER

TWO

ACKNOWLEDGEMENTS

Christopher R. Wentland acknowledges support from the US Air Force Office of Scientific Research through the Center of Excellence Grant FA9550-17-1-0195 (Technical Monitors: Fariba Fahroo, Mitat Birkan, Ramakanth Munipalli, Venkateswaran Sankaran). This work has also been supported by a grant of computer time from the DOD High Performance Computing Modernization Program.

2.1 Quick Start

2.1.1 Dependencies

PERFORM is a pure Python code and does not (as of the writing of this section) depend on any non-Python software. As such, everything can be installed directly through `pip` and is done so through the `pip install` command explained below.

The minimum required Python version is 3.6, but is only actively tested with Python 3.8. Additionally, it is only actively test in Ubuntu 20.04, but should run without many problems on other Linux distributions, Windows Subsystem for Linux (WSL), and macOS. Some issues have been noted in non-Ubuntu OS's when executing Bash scripts used for downloading data from Google Drive and executing tests.

The baseline solver only requires three additional packages: `numpy`, `scipy`, and `matplotlib`. These will be installed, or updated if your local installations are older than the minimum required versions, upon installing **PERFORM**.

Neural network ROM models using TensorFlow-Keras of course depend on `tensorflow`, though the code will only throw an error if you attempt to run one of those models and so does not require `tensorflow` to run the baseline solver or other ROM models. These models are only tested for the most recent production release of TensorFlow 2, and we make no guarantees that the code will work correctly or optimally for older versions (and will definitely not work with TensorFlow 1).

2.1.2 Installing

To get up and running with **PERFORM**, first clone the source code repository by executing the following command from your terminal command line:

```
git clone https://github.com/cwentland0/perform.git
```

or use your Git client of choice, along with the above HTTPS address. **PERFORM** is currently installed locally via `pip` (or `pip3`, if your `pip` does not automatically install for your Python 3 distribution). To do this, enter the **PERFORM** root folder and execute

```
pip install -e .
```

This will install **PERFORM**, as well as any required package dependencies which you have not yet installed.

2.1.3 Running

After installation is complete, a new script, `perform`, will be added to you Python scripts. This is the command that you will use to execute **PERFORM**, followed by the path to the working directory of the case you would like to run, e.g.

```
perform /path/to/working/directory
```

The working directory of a case is dictated by the presence of a `solver_params.inp` file, described in detail in *solver_params.inp*. The code will not execute if there is not a properly-formatted `solver_params.inp` file in the specified working directory. **If you are running a ROM case, an additional `rom_params.inp` file must also be placed in the working directory**. This file is described in detail in *rom_params.inp*.

You can check that **PERFORM** works by entering any of the example case directories (e.g. `perform/examples/shock_tube`) and executing

```
perform .
```

If running correctly, your terminal's STDOUT should start filling with output from each time step iteration and live field and probe plots should appear on your screen. Alternatively, you can run the test suite described below to check that your installation works as expected.

2.1.4 Testing

A suite of unit, integration, and regression tests are included in `perform/tests/`. These can be run manually from the **PERFORM** root directory by executing

```
tests/run_tests.sh
```

This will automatically run unit and integration tests and report whether they succeeded or failed. You will then be prompted as to whether you would like to run the regression tests. These can take a while to complete, and really only needs to be checked before submitting a pull request. Note that the regression tests use the included example cases, so if you altered the input files for those cases then make sure to reset them before executing the regression tests.

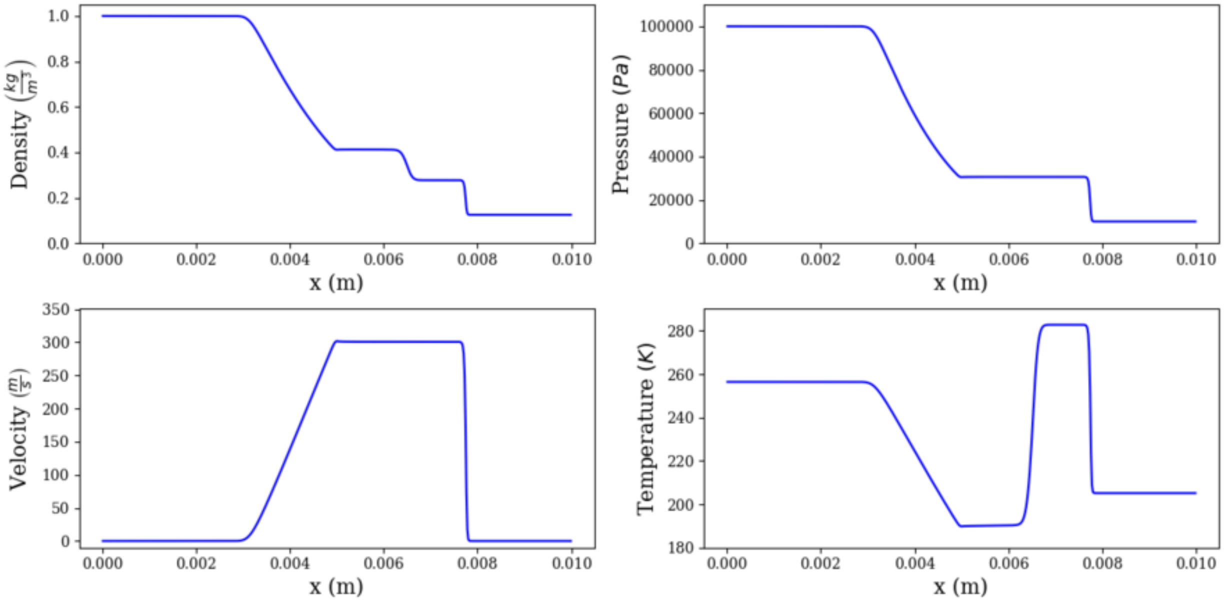
2.2 Example Cases

Several example cases are provided in `perform/examples/` to get you familiar with input file formatting and the various solver outputs. They are presented in a rough hierarchy of ROM modeling difficulty, starting from the simplest and building in difficulty by introducing additional complex physical phenomena. Ideally, new ROM methods should be tested for all problems, and their relative strengths and weaknesses in tackling each problem should be exposed in detail.

Additionally, sample ROM input files may be downloaded via `setup_sample_rom.sh` files included in the `standing_flame` and `transient_flame` sample case directories. After running the scripts, the ROM cases can be immediately executed with `perform .`.

2.2.1 Sod Shock Tube

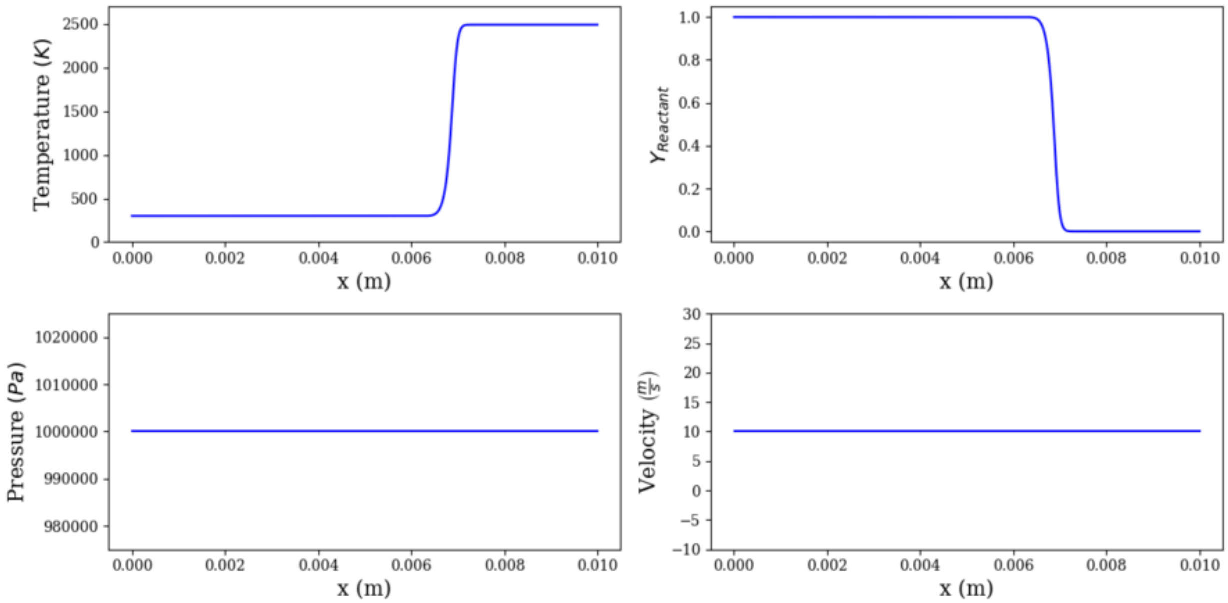
The Sod shock tube is a classic benchmark problem. The setup of the problem models two “chambers” of gas, one at a high pressure and density and the other at a low pressure and density, separated by a diaphragm. The start of the simulation models this diaphragm instantly bursting, with a shock/contact wave combo traveling through the low-pressure gas and a rarefaction wave traveling through the high pressure gas.



This case presents a multitude of challenges for ROMs, featuring several phenomena which can be found in reacting flows, even though this case is a single-species non-reacting flow. Traditional linear subspace ROMs experience strong ringing artifacts near the shock and contact, while most static solution representations fail to propagate the waves beyond the model training period.

2.2.2 Transient Contact Surface

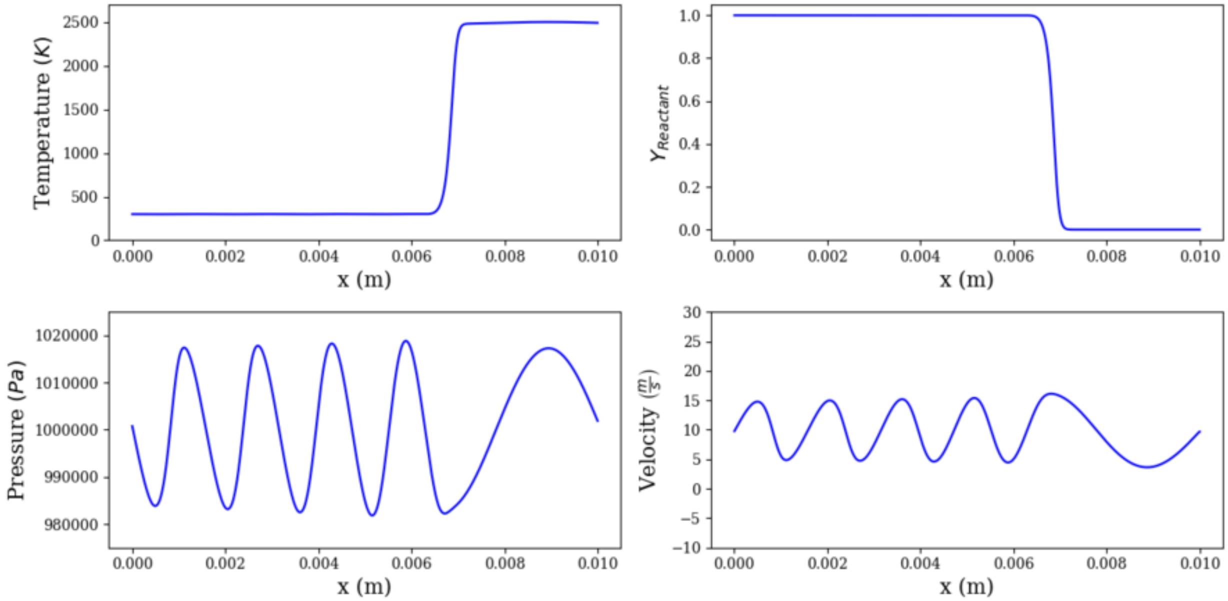
This case introduces a multiple chemical species (two, to be exact) configuration, featuring a low-temperature “reactant” species and a high-temperature “product” species, all at a uniform pressure and velocity. The gas’s average velocity of 10 m/s propels the resulting contact surface downstream. The viscosity and reaction are turned **off** for this case.



Similarly to the Sod shock tube, this case exhibits strong gradients in temperature and species mass fraction that can be difficult for traditional linear subspace methods to capture, along with a transient wave which can be hard to propagate beyond the model training period. Frankly, this case may be even easier than the Sod shock tube, but serves as a gentle introduction to the multi-species formulation.

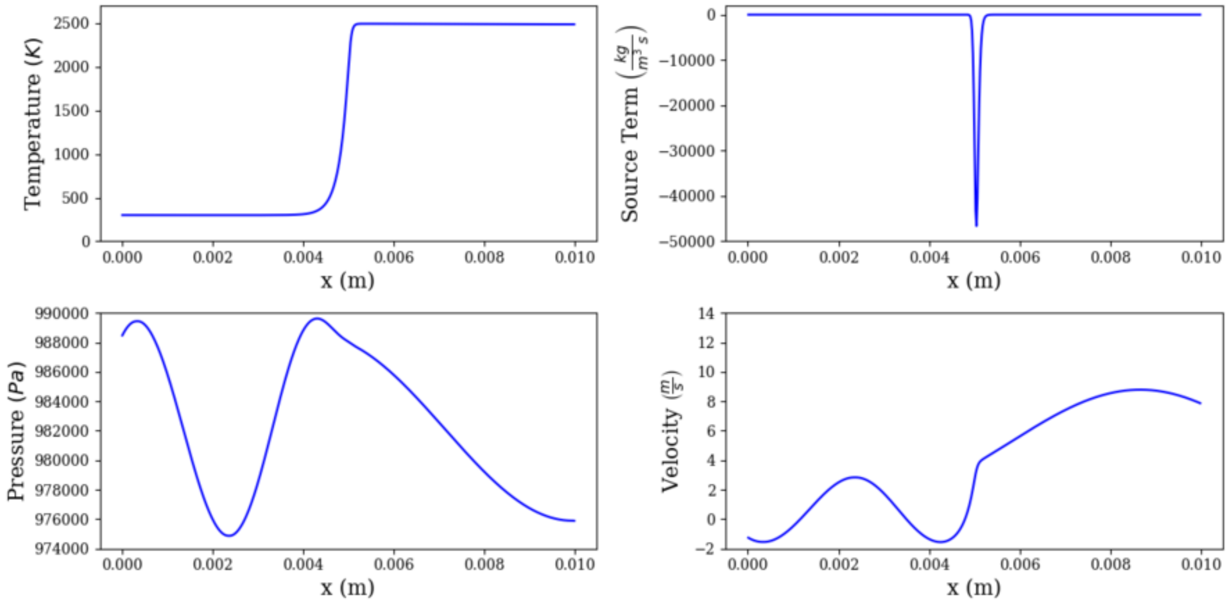
Contact Surface w/ Artificial Forcing

To introduce additional complexity to the transient contact surface case, the user may apply artificial pressure forcing at the outlet to introduce an acoustic wave propagating upstream. The differing density between the cold reactant and hot product species results in different local sounds speeds, which makes for some interesting system acoustics which can be challenging for ROMs to reproduce.



2.2.3 Standing Flame w/ Artificial Forcing

This case is similar to the contact surface in the sense that it features a cold "reactant" species and a hot "product" species. However, the viscosity and reaction are turned **on** for this case, with a single-step irreversible reaction mechanism which simply converts "reactant" to "product". Additionally, the bulk velocity of the fluid is decreased to the point that the reaction and diffusion is perfectly balanced with the bulk velocity, resulting in an effectively stationary flame. Artificial pressure forcing is applied at the outlet, causing a single-frequency acoustic wave to propagate upstream.

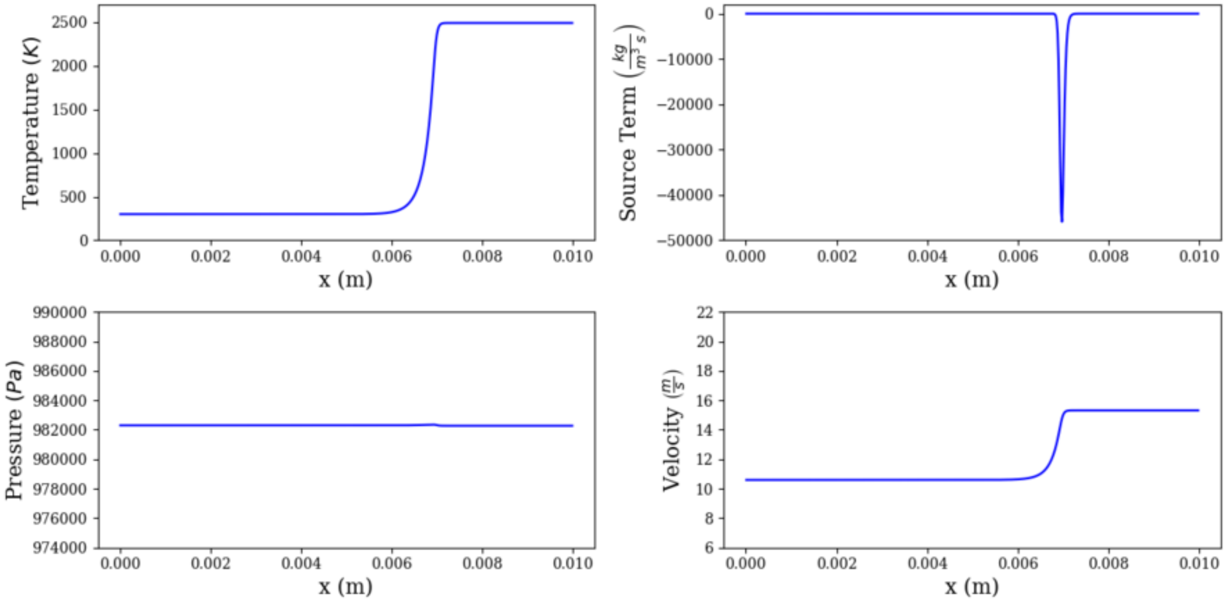


This is an incredibly simple reacting flow problem, one which linear subspace ROMs should nail with only a few trial basis modes. This is not surprising, as the flow is largely stationary, with all fluctuations exhibiting a fixed frequency and amplitude. However, we've observed that non-linear autoencoder projection ROMs may fail even for this simple case, and users should take care to check whether their methods succeed.

A `setup_sample_rom.sh` script is also provided to download input files for a linear MP-LSVT projection ROM. Simply execute the script and the necessary input files will be unpacked, after which the ROM may be executed normally.

2.2.4 Transient Flame

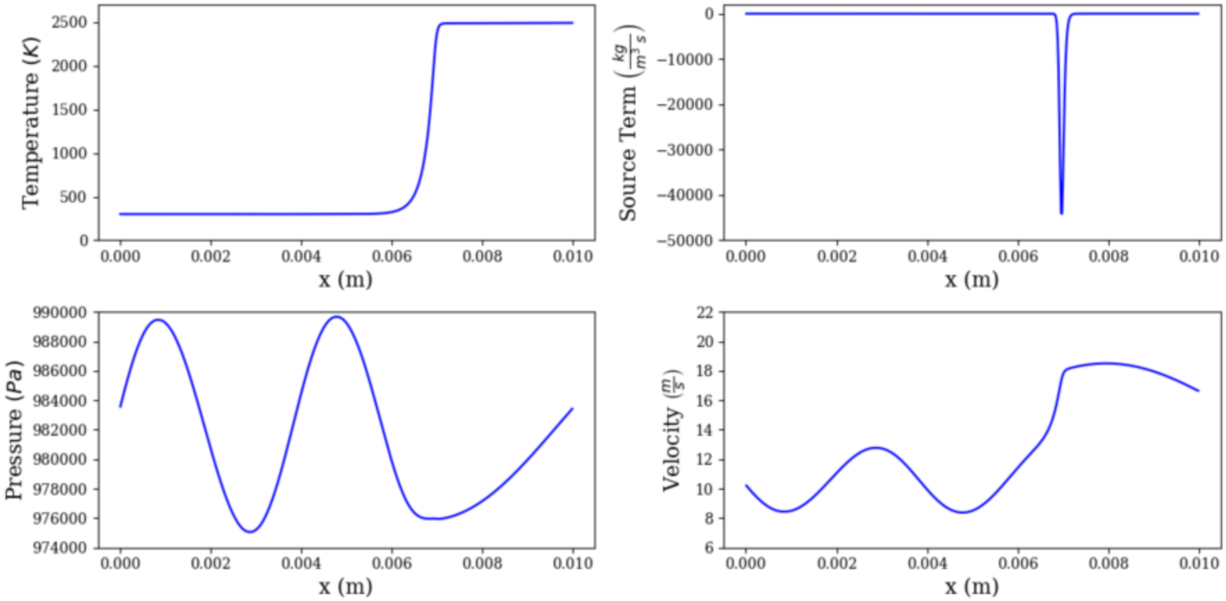
This case exhibits all of the features of the previous test cases: a cold "reactant" species diffusing into a hot "product" species, a single-step reaction mechanism, and a higher bulk fluid velocity to cause the flame to advect downstream. The sharp gradients in temperature and species mass fraction, the stiff reaction source term, and the bulk advection of the sharp gradients make for a fairly challenging problem.



A `setup_sample_rom.sh` script is also provided to download input files for a non-linear autoencoder MP-LSVT projection ROM via TensorFlow-Keras. Simply execute the script and the necessary input files will be unpacked, after which the ROM may be executed normally. Note that the execution of this ROM is expected to be much, much slower than that of the FOM, as deep autoencoder ROMs are notoriously computationally expensive.

Transient Flame w/ Artificial Forcing

The complexity of the transient flame problem may be further increased by applying artificial pressure forcing at the outlet, causing an acoustic wave to propagate upstream. As the amplitude and frequency of the forcing is increased, the interaction between the system acoustics and the flame becomes increasingly complex.



These highly non-linear interactions lie at the core of the problems the authors are working to tackle, namely the often-disastrous feedback loop between unsteady reaction heat release and system acoustics in modern combustion devices such as rocket or gas turbine combustors.

2.3 Inputs

This section outlines the various input files that are required to run **PERFORM**, as well as the input parameters that are used in text input files. If you are having issues running a case (in particular, experience a `KeyError` error) or observing strange solver behavior, please check that all of your input parameters are set correctly, and use this page as a reference. Examples of some of these files can be found in the example cases in `perform/examples/`.

All text input files are parsed using regular expressions. All input parameters must be formatted as `input_name = input_value`, with as much white space before and after the equals sign as desired. A single line may contain a single input parameter definition, denoted by a single equals sign. An input file may contain as many blank lines as desired, and any line without an equals sign will be ignored (useful for user comments). Examples of various input parameters are given below

```
# sample user comment
samp_string        = "example/string_input"
samp_int           = 3
samp_float_dec     = 3.14159
samp_float_sci     = 6.02214e23
samp_bool          = False
samp_list          = [1.0, 2.0, 3.0]
samp_list_of_lists = [["1_1", "1_2"],["2_1", "2_2"]]
```

As a rule, you should write input values as if you were writing them directly in Python code. As seen above, string values should be enclosed in double quotes (single quotes is also valid), boolean values should be written precisely as `False` or `True` (case sensitive), lists should be enclosed by brackets (e.g. `[val_1, val_2, val_3]`), and lists of lists should be formatted in kind (e.g. `[[val_11, val_12],[val_21, val_22]]`). Even if a list input only has one entry, it should be formatted as a list in the input file. Certain input parameters also accept a value of `None` (case sensitive). Each `input_name` is case sensitive, and string input parameters are also case sensitive. We are working on making these non-case sensitive where it's possible.

Below, the formats and input parameters for each input file are described. For text file inputs, tables containing all possible parameters are given, along with their expected data type, default value and expected units of measurement (where applicable). Note that expected types of `list` of `lists` is abbreviated as `lol` for brevity. For detailed explanations of each parameter, refer to *Input Parameter Index*.

2.3.1 solver_params.inp

The `solver_params.inp` file is a text file containing input parameters for running all simulations, FOM or ROM. It is the root input file from which the gas file, mesh file, and initial condition file are specified. Further, this file specifies all parameters related to the flux scheme, time discretization, robustness control, unsteady outputs, and visualizations. **It must be placed in the working directory, and must be named** `solver_params.inp`. Otherwise, the code will not function.

Table 1: `solver_params.inp` input parameters

Parameter	Type	Default	Units
`chem_file`	`str`	-	-
`mesh_file`	`str`	-	-
`init_file`	`str`	-	-
`ic_params_file`	`str`	-	-

continues on next page

Table 1 – continued from previous page

Parameter	Type	Default	Units
`dt`	`float`	-	s
`time_scheme`	`str`	-	-
`time_order`	`int`	-	-
`num_steps`	`int`	-	-
`subiter_max`	`int`	`50`	-
`res_tol`	`float`	`1e-12`	Unitless
`dual_time`	`bool`	`True`	-
`dtau`	`float`	`1e-5`	s
`adapt_dtau`	`bool`	`False`	-
`cfl`	`float`	`1.0`	Unitless
`vnn`	`float`	`20.0`	Unitless
`run_steady`	`bool`	`False`	-
`steady_tol`	`float`	`1e-12`	Unitless
`invisc_flux_scheme`	`str`	`"roe"`	-
`visc_flux_scheme`	`str`	`"invisc"`	-
`space_order`	`int`	`1`	-
`grad_limiter`	`str`	`"none"`	-
`bound_cond_inlet`	`str`	-	-
`press_inlet`	`float`	-	BC-dependent
`vel_inlet`	`float`	-	BC-dependent
`temp_inlet`	`float`	-	BC-dependent
`rho_inlet`	`float`	-	BC-dependent
`mass_fracs_inlet`	`list of float`	-	BC-dependent
`pert_type_inlet`	`str`	-	-
`pert_perc_inlet`	`float`	-	Unitless
`pert_freq_inlet`	`list of float`	-	1/s
`bound_cond_outlet`	`str`	-	-
`press_outlet`	`float`	-	BC-dependent
`vel_outlet`	`float`	-	BC-dependent
`temp_outlet`	`float`	-	BC-dependent
`rho_outlet`	`float`	-	BC-dependent
`mass_fracs_outlet`	`list of float`	-	BC-dependent
`pert_type_outlet`	`str`	-	-
`pert_perc_outlet`	`float`	-	Unitless
`pert_freq_outlet`	`list of float`	-	1/s
`vel_add`	`float`	`0.0`	m/s
`stdout`	`bool`	`True`	-
`res_norm_prim`	`list of float`	`[1e5, 10, 300, 1]`	[Pa, m/s, K, unitless]
`source_off`	`bool`	`False`	-
`save_restarts`	`bool`	`False`	-
`restart_interval`	`int`	`100`	-
`num_restarts`	`int`	`20`	-
`init_from_restart`	`bool`	`False`	-
`probe_locs`	`list of float`	`[None]`	m
`probe_vars`	`list of str`	`[None]`	-
`out_interval`	`int`	`1`	-
`prim_out`	`bool`	`True`	-
`cons_out`	`bool`	`False`	-
`source_out`	`bool`	`False`	-

continues on next page

Table 1 – continued from previous page

Parameter	Type	Default	Units
`hr_out`	`bool`	`False`	-
`rhs_out`	`bool`	`False`	-
`vis_interval`	`int`	`1`	-
`vis_show`	`bool`	`True`	-
`vis_save`	`bool`	`False`	-
`vis_type_X`	`str`	-	-
`vis_var_X`	`list` of `str`	-	-
`vis_x_bounds_X`	`lol` of `float`	`[[None,None]]`	plot-dependent
`vis_y_bounds_X`	`lol` of `float`	`[[None,None]]`	plot-dependent
`probe_num_X`	`int`	-	-
`calc_rom`	`bool`	`False`	-

2.3.2 Mesh File

The mesh file is a text file containing input parameters for defining the computational mesh. The name and location of the mesh file is arbitrary, and is referenced from the `mesh_file` input parameter in `solver_params.inp`.

As of the writing of this section, **PERFORM** can solve on uniform meshes. Thus, the defining parameters are fairly simple.

Table 2: Mesh file inputs

Parameter	Type	Default	Units
`x_left`	`float`	-	m
`x_right`	`float`	-	m
`num_cells`	`int`	-	-

2.3.3 Chemistry File

The chemistry file is a text file containing input parameters for defining properties of the chemical species modeled in a given simulation, along with parameters which define the reactions between these species. The name and location of this file are arbitrary, and is referenced from the `chem_file` input parameter in `solver_params.inp`.

The set of parameters which is required for any gas or reaction model are given in *Universal Chemistry Inputs*. Those required for a calorically-perfect gas model (`gas_model = "cpg"`) are given in *CPG Inputs*. Those required for a finite-rate irreversible reaction model (`reaction_model = "fr_irrev"`) are given in *Finite Rate Irreversible Reaction Inputs*. To be abundantly clear, **these parameters should all be given in the same chemistry file**, but they are split into different sections here for clarity.

Universal Chemistry Inputs

The parameters described here are required for any combination of gas model and reaction model.

Table 3: Universal chemistry file inputs

Parameter	Type	Default	Units
`gas_model`	`str`	`"cpg"`	-
`reaction_model`	`str`	`"none"`	-
`num_species`	`int`	-	-
`species_names`	`list` of `str`	-	-
`mol_weights`	`list` of `float`	-	g/mol

CPG Inputs

The parameters described here are required when using a calorically-perfect gas model, i.e. when setting `gas_model = "cpg"`.

Table 4: CPG chemistry file inputs

Parameter	Type	Default	Units
`enth_ref`	`list` of `float`	-	J/kg
`cp`	`list` of `float`	-	J/K-kg
`pr`	`list` of `float`	-	Unitless
`sc`	`list` of `float`	-	Unitless
`temp_ref`	`list` of `float`	-	K
`mu_ref`	`list` of `float`	-	N-s/m^2

Finite Rate Irreversible Reaction Inputs

The parameters described here are required when using a finite-rate irreversible reaction model, i.e. when setting `reaction_model = "fr_irrev"`.

Table 5: Finite rate irreversible reaction model chemistry file inputs

Parameter	Type	Default	Units
`nu`	`lol` of `float`	-	Unitless
`nu_arr`	`lol` of `float`	-	Unitless
`act_energy`	`list` of `float`	-	kJ/mol
`pre_exp_fact`	`list` of `float`	-	Unitless
`temp_exp`	`list` of `float`	-	Unitless

2.3.4 Initial Condition Inputs

Unsteady solutions can be initialized in three different ways in **PERFORM**: piecewise uniform function parameters files (*Piecewise Uniform IC File*), full primitive state NumPy profiles (*NumPy Primitive IC File*), or restart files (*Restart Files*). If multiple restart methods are requested, the following priority hierarchy is followed: restart files first, then primitive state NumPy files, and finally a piecewise uniform function.

Piecewise Uniform IC File

The piecewise uniform initial condition file is a text file containing input parameters for initializing a simulation from a two-section piecewise uniform profile describing the full primitive state. This is done by specifying a “left” and “right” primitive state, and a spatial point on the computational mesh at which the two states are separated. This is ideal for initializing problems like the *Sod Shock Tube* or flames.

Table 6: Piecewise uniform IC inputs

Parameter	Type	Default (Units)	Units
`x_split`	`float`	-	m
`press_left`	`float`	-	Pa
`vel_left`	`float`	-	m/s
`temp_left`	`float`	-	K
`mass_fracs_left`	`list` of `float`	-	Unitless
`press_right`	`float`	-	Pa
`vel_right`	`float`	-	m/s
`temp_right`	`float`	-	K
`mass_fracs_right`	`list` of `float`	-	Unitless

NumPy Primitive IC File

Providing a complete primitive state profile is by far the simplest initialization method available. The `init_file` parameter in `solver_params.inp` provides the arbitrary location of a NumPy binary (`*.npy`) containing a single NumPy array. This NumPy array must be a two- or three-dimensional array, where the first dimension is the number of governing equations in the system (3 + `num_species` - 1) and the second dimension is the number of cells in the discretized spatial domain. The order of the first dimension *must* be ordered by pressure, velocity, temperature, and then chemical species mass fraction. The chemical species mass fractions must be ordered as they are in the chemistry file. The optional third dimension is the time step dimension; if a higher-order time integration method is requested, the initial condition profile may provide prior time steps to preserve this order of accuracy upon initialization. If only one time step or a two-dimensional profile is provided, the time integrator will attempt to “cold start” from a first-order scheme.

This file can be generated however you like, such as ripping it manually from the unsteady outputs of a past **PERFORM** run, or generating a more complex series of discontinuous steps than what the `ic_params_file` settings handle natively.

Restart Files

Restart files accomplish what the name implies: restarting the simulation from a previous point in the simulation. Restart files are saved to the `restart_files` directory in the working directory when `save_restarts = True` at an interval specified by `restart_interval` in `solver_params.inp`. Two files are saved to reference a restart solution: a `restart_iter.dat` file and a `restart_file_X.npz` file, where `X` is the *restart iteration number*. The latter file contains both the conservative and primitive solution saved at that restart iteration, as well as the physical solution time associated with that solution. The former file is an text file containing the restart iteration number of the most recently-written restart file, and thus points to which `restart_file_X.npz` should be read in to initialize the solution. It is overwritten every time a restart file is written. Similarly, the maximum number of `restart_file_X.npz` saved to disk is dictated by `num_restarts`. When this threshold is reached, the restart iteration number will loop back to 1 and begin overwriting old restart files.

Setting `init_from_restart = True` will initialize the solution from the restart file whose restart iteration number matches the one given in `restart_iter.dat`. Thus, without modification, the solution will restart from the most recently generated restart file. However, if you want to restart from a different iteration number, you can manually change the iteration number stored in `restart_iter.dat`.

2.3.5 rom_params.inp

The `rom_params.inp` file is a text file containing input parameters for running ROM simulations. **It must be placed in the working directory**, the same directory as its accompanying `solver_params.inp` file. Parameters in this file are detailed in *rom_params.inp*.

2.4 Outputs

2.4.1 Unsteady Solution Data

Field Data

Unsteady field data represents the time evolution of an unsteady field at the time step iteration interval specified by `out_interval` in `solver_params.inp`. All unsteady field data is written to `working_dir/unsteady_field_results`, and currently comes in five flavors: primitive state output, conservative state output, source term output, heat release output, and RHS term output. With the exception of the heat release output, all have the same general form: a NumPy binary file containing a single NumPy array with three dimensions. The first dimension is the number of variables, the second is the number of cells in the computational mesh, and the third is the number of time steps saved in the output file. The primitive state, conservative state, and RHS term output have the same number of variables, equal to the number of governing equations (3 + `num_species` - 1), while the source term only has (`num_species` - 1) variables. Heat release output only has two dimensions, the first for the number of cells and the second for the number of time steps.

Primitive state field data is saved if `prim_out = True` and has the prefix `sol_prim_*`. Conservative state field data is saved if `cons_out = True` and has the prefix `sol_cons_*`. Source term field data is saved if `source_out = True` and has the prefix `source_*`. RHS term field data is saved if `rhs_out = True` and has the prefix `rhs_*`. Heat release field data is saved if `hr_out = True` and has the prefix `heat_release_*`.

Unsteady field data may have three different main suffixes, depending on solver parameters: `*_FOM` for an unsteady full-order model simulation, `*_steady` for a "steady" full-order model simulation (see *Running in "Steady" Mode*) for details), or `*_ROM` for a reduced-order model simulation. An additional suffix, `*_FAILED`, is also appended if the solver fails (solution blowup). Thus, the conservative field results for a failed ROM run would have the name `sol_cons_ROM_FAILED.npy`, while a successful run would simply generate `sol_cons_ROM.npy`.

Probe Data

Probe/point monitor data represents the time evolution of an unsteady field variable at a single finite volume cell. Probe locations are specified by `probe_locs` in `solver_params.inp`, and the fields to be measured are specified by `probe_vars`. Valid options for `probe_vars` can be found in *Input Parameter Index*. Probe measurements are taken at every physical time step. Data for each probe is saved to a separate file in `working_dir/probe_results`. Each has the format of a NumPy binary file containing a single two-dimensional NumPy array. The first dimension is the number of variables listed in `probe_vars` plus one, as the physical solution time is also stored at each iteration. The second dimension is the number of physical time steps for which the simulation was run.

The name of each probe begins with `probe_*`, and is followed by a list of the variables stored in the probe data. Finally, the same suffixes mentioned above are applied depending on the solver settings: ``*_FOM`, ``*_steady``, and `*_ROM`. Again, if the solver fails, the suffix `*_FAILED` will also be appended. Finally, the 1-indexed number of a the probe will be appended to the end of the file. For example, the second probe monitoring the velocity and momentum of a "steady" solve which fails will have the file name `probe_velocity_momentum_2_steady_FAILED.npy`.

Restart Files

All restart file data is stored in `working_dir/restart_files`. Please refer to *Restart Files* for details on the formatting and contents of restart files.

2.4.2 Visualizations

During simulation runtime, **PERFORM** is capable of generating two types of plots via `matplotlib`: field plots and probe monitor plots. If `vis_show = True` in `solver_params.inp`, then these images are displayed on the user's monitor. If `vis_save = True`, they are saved to disk. The interval of displaying/saving the figures is given by `vis_interval`. Each figure corresponds to a single instance of `vis_type_X`, within which there may be several plots. Each probe figure corresponds to a single probe, from which multiple probed variables may be extracted.

All saved images are PNG images stored within `working_dir/image_results`.

Field Plots

Field Plots display instantaneous snapshots of the entire field with the field data plotted on the y-axis and cell center coordinates plotted on the x-axis.

Field plots save an instantaneous snapshot of the field plots at the interval set by `vis_interval`. These are stored within a subdirectory following the same pattern given to field data files, except the prefix of the directory is given by `working_dir/image_results/field_*`. Within this subdirectory, individual images have the prefix `fig_*`, followed by the number of the image in the series of expected image numbers to be generated by a given run. If a simulation terminates early any field plots that were expected to be generated will not be generated.

Probe Plots

Probe plots display the entire time history of the probed data up to the most recent plotting interval reached by the simulation, with the probed variable data plotted on the y-axis and time plotted on the x-axis.

A single figure is saved to disk for a given probe figure. It if first written after `vis_interval` time steps, after which the same file is overwritten at the interval specified by `vis_interval`. The names of probe plots follow the same pattern given to the probe data files (except with the file extension `*.png`, of course).

2.5 Input Parameter Index

This section provides a comprehensive index of all solver input parameters for the text input files detailed in *Inputs*.

2.5.1 solver_params.inp

See *solver_params.inp* for variable types, default values, and units (where applicable).

- `mesh_file`: Path to mesh file. Permits absolute path, or relative path from working directory.
- `chem_file`: Path to chemistry file. Permits absolute path, or relative path from working directory.
- `init_file`: Path to full initial primitive state profile (stored in a `*.npy` NumPy binary file) to initialize the unsteady solution from. Permits absolute path, or relative path from working directory. If `ic_params_file` is set, or `init_from_restart = True`, this parameter will be ignored.

- `ic_params_file`: Path to left/right (step function) primitive state parameters file to initialize the unsteady solution from. Permits absolute path, or relative path from working directory. If `init_from_restart = True`, this parameter will be ignored.
- `dt`: Fixed time step size for numerical time integration.
- `time_scheme`: Name of numerical time integration scheme to use. Please see the theory documentation for details on each scheme.
  - Valid options: `ssp_rk3`, `classic_rk4`, `jameson_low_store`, `bdf`
- `time_order`: Order of accuracy for the chosen time integrator. Some time integrators have a fixed order of accuracy, while others may accept several different values. If a time integrator has a fixed order of accuracy and you have entered a different order of accuracy, a warning will display but execution will continue. If a time integrator accepts several values and an invalid value is entered, the solver will terminate with an error.
- `num_steps`: Number of discrete physical time steps to run the solver through. If the solver fails before this number is reached (either through a code failure or solution blowup), any unsteady output files will be dumped to disk with the suffix `_FAILED"` appended to denote a failed solution.
- `subiter_max`: The maximum number of subiterations that the iterative solver for implicit time integration schemes may execute before concluding calculations for that physical time step.
- `res_tol`: The threshold of convergence of the ℓ^2 norm of the Newton iteration residual below which the subiteration loop will automatically conclude.
- `dual_time`: Boolean flag to specify whether dual time-stepping should be used for an implicit time integration scheme.
- `dtau`: Fixed value of $\Delta\tau$ to use for dual time integration. Ignored if `adapt_dtau = True`.
- `adapt_dtau`: Boolean flag to specify whether the value of $\Delta\tau$ should be adapted at each subiteration according to the below robustness control parameters.
- `cfl`: Dual time-stepping Courant–Friedrichs–Lewy number to adapt $\Delta\tau$ based on maximum wave speed in each cell. Smaller CFL numbers will result in smaller $\Delta\tau$, and greater regularization as a result.
- `vnn`: Dual time-stepping von Neumann number to adapt $\Delta\tau$ based on the mixture kinematic viscosity in each cell. Smaller VNN numbers will result in smaller $\Delta\tau$, and greater regularization as a result.
- `run_steady`: Boolean flag to specify whether to run the solver in “steady” mode. See *Running in “Steady” Mode* for more details.
- `steady_tol`: If `run_steady = True`, the threshold of convergence of the ℓ^2 norm of the change in the primitive solution below which the steady solve will automatically conclude.
- `invisc_flux_scheme`: Name of the numerical inviscid flux scheme to use. Please see the theory documentation for details on each scheme.
  - Valid options: `roe`
- `visc_flux_scheme`: Name of the numerical viscous flux scheme to use. Please see the theory documentation for details on each scheme.
  - Valid options: `inviscid`, `standard`
- `space_order`: Order of accuracy of the state reconstructions at the cell faces for flux calculations. Must be a positive integer. If `space_order = 1`, the cell-centered values are used. If `space_order > 1`, finite difference stencils are used to compute cell-centered gradients, from which higher-order face reconstructions are computed. If the gradient calculation for the value entered has not been implemented, the solver with terminate with an error.
- `grad_limiter`: Name of the gradient limiter to use when computing higher-order face reconstructions. Please see the solver documentation for details on each scheme.

  - Valid options: `barth_cell`, `barth_face`, `venkat`
- `bound_cond_inlet`: Name of the boundary condition to apply at the inlet. For details on each boundary condition, see the solver documentation. For required input parameters for a given boundary condition, see *Inlet BCs and Parameters*.
  - Valid options: `stagnation`, `fullstate`, `meanflow`
- `press_inlet`: Pressure-related value for inlet boundary condition calculations.
- `vel_inlet`: Velocity-related value for inlet boundary condition calculations.
- `temp_inlet`: Temperature-related value for inlet boundary condition calculations.
- `rho_inlet`: Density-related value for inlet boundary condition calculations.
- `mass_fracs_inlet`: Chemical composition-related value for inlet boundary condition calculations.
- `pert_type_inlet`: Type of value to be perturbed at the inlet. See *Inlet BCs and Parameters* for valid options for each `bound_cond_inlet` value.
- `pert_perc_inlet`: Percentage of the specified perturbed value, determining the amplitude of the inlet perturbation signal. Should be entered in decimal format, e.g. for a 10% perturbation, enter `pert_perc_inlet = 0.01`. See *Boundary Perturbations* for more details.
- `pert_freq_inlet`: List of superimposed frequencies of the inlet perturbation. See *Boundary Perturbations* for more details.
- `bound_cond_outlet`: Name of the boundary condition to apply at the outlet. For details on each boundary condition, see the solver documentation. For required input parameters for a given boundary condition, see *Outlet BCs and Parameters*.
- `press_outlet`: Pressure-related value for outlet boundary condition calculations.
- `vel_outlet`: Velocity-related value for outlet boundary condition calculations.
- `temp_outlet`: Temperature-related value for outlet boundary condition calculations.
- `rho_outlet`: Density-related value for outlet boundary condition calculations.
- `mass_fracs_outlet`: Chemical composition-related value for outlet boundary condition calculations.
- `pert_type_outlet`: Type of value to be perturbed at the outlet. See *Outlet BCs and Parameters* for valid options for each `bound_cond_outlet` value.
  - Valid options: `subsonic`, `meanflow`
- `pert_perc_outlet`: Percentage of the specified perturbed value, determining the amplitude of the outlet perturbation signal. Should be entered in decimal format, e.g. for a 10% perturbation, enter `pert_perc_outlet = 0.1`. See *Boundary Perturbations* for more details.
- `pert_freq_outlet`: List of superimposed frequencies of the outlet perturbation. See *Boundary Perturbations* for more details.
- `vel_add`: Velocity to be added to the entire initial condition velocity field. Accepts negative values.
- `stdout`: Boolean flag to specify whether to print iteration counts and residual norms to STDOUT.
- `res_norm_prim`: List of values by which to normalize each field of the ℓ^2 and ℓ^1 residual norms before averaging across all fields. They are order by pressure, velocity, temperature, and then all species mass fractions except the last. This ensures that the norms of each residual field contribute roughly equally to the average norm used to determine Newton's method convergence.
- `source_off`: Boolean flag to specify whether to apply the reaction source term. This is `False` by default; setting it manually to `True` turns off the source term. This can save computational cost for non-reactive cases.

- `save_restarts`: Boolean flag to specify whether to save restart files.
- `restart_interval`: Physical time step interval at which to save restart files.
- `num_restarts`: Maximum number of restart files to store. After this threshold has been reached, the count returns to 1 and the first restart file is overwritten by the next restart file (and so on).
- `init_from_restarts`: Boolean flag to determine whether to initialize the unsteady solution from
- `probe_locs`: List of locations in the spatial domain to place point monitors. The probe measures values at the cell center closest to the specified location. If a location is less than the inlet boundary location, the inlet ghost cell will be monitored. Likewise, if a location is greater than the outlet boundary location, the outlet ghost cell will be monitored. These probe monitors are recorded at every physical time iteration and the time history is written to disk. See *Probe Data* for more details on the output.
- `probe_vars`: A list of fields to be probed at each specified probe location.
  - Valid for all probes: `"pressure"`, `"velocity"`, `"temperature"`, `"density"`, `"momentum"`, `"energy"`, `"species_X"`, `"density-species_X"` (where `X` is replaced by the integer number of the desired chemical species to be probed, e.g. `"species_2"` for the second species specified in the chemistry file).
  - Valid options for interior probes only: `"source"`, `"heat-release"`
- `out_interval`: Physical time step interval at which to save unsteady field data.
- `prim_out`: Boolean flag to specify whether the unsteady primitive state should be saved.
- `cons_out`: Boolean flag to specify whether the unsteady conservative state should be saved.
- `source_out`: Boolean flag to specify whether the unsteady source term field should be saved.
- `hr_out`: Boolean flag to specify whether the unsteady heat release rate should be saved.
- `rhs_out`: Boolean flag to specify whether the unsteady right-hand-side field should be saved.
- `vis_interval`: Physical time step interval at which to draw any requested field/probe plots. If no plots are requested, this parameter is ignored.
- `vis_show`: Boolean flag to specify whether field/probe plots should be displayed on the user's monitor at the interval specified by `vis_interval`. If no plots are requested, this parameter is ignored.
- `vis_save`: Boolean flag to specify whether field/probe plots should be saved to disk at the interval specified by `vis_interval`. If no plots are requested, this parameter is ignored.
- `vis_type_X`: Type of data to visualize in the `X`th figure. For example, `vis_type_3` would specify the type of the third plot to be visualized. Values of `X` must start from 0 and progress by one for each subsequent plot. Any gap in these numbers will cause any plots after the break to be ignored (e.g. specifying `vis_type_0`, `vis_type_2`, and `vis_type_3` without specifying `vis_type_1` will automatically ignore the plots for `vis_type_2` and `vis_type_3`).
  - Valid options: `field`, `probe`
- `probe_num_X`: 0-indexed number of the point monitor to visualize in the `X`th figure if `vis_type_X = "probe"`. Must correspond to a valid probe number.
- `vis_var_X`: A list of fields to be plotted in the `X`th figure. Note that for `vis_type_X = "probe"` figures, if a specified field is not being monitored at the probe specified by `probe_num_X`, the solver will terminate with an error.
- `vis_x_bounds_X`: List of lists, where each sub-list corresponds to the plots specified in `vis_var_X`. Each sublist contains two entries corresponding the lower and upper x-axis bounds for visualization of `vis_var_X`.
- `vis_y_bounds_X`: List of lists, where each sub-list corresponds to the plots specified in `vis_var_X`. Each sublist contains two entries corresponding the lower and upper y-axis bounds for visualization of `vis_var_X`.

- `calc_rom`: Boolean flag to specify whether to run a ROM simulation. If set to `True`, a `rom_params.inp` file must also be placed in the working directory. See *rom_params.inp* for more details on this input file.

2.5.2 Mesh File

See *Mesh File* for variable types, default values, and units (where applicable).

- `x_left`: Left-most boundary coordinate of the spatial domain. This point will be the coordinate of theleft face of the left-most finite volume cell.
- `x_right`: Right-most boundary coordinate of the spatial domain. This point will be the coordinate of theright face of the right-most finite volume cell.
- `num_cells`: Total number of finite volume cells in the discretized spatial domain.

2.5.3 Chemistry File

We break down the sections of the chemistry file input file, as in *Inputs*.

Universal Chemistry Inputs

See *Universal Chemistry Inputs* for variable types, default values, and units (where applicable).

- `gas_model`: Name of the gas model to be used.
    - Valid options: `"cpg"`
- `reaction_model`: Name of the reaction model to be used.
    - Valid options: `"none"`, `"fr_irrev"`
- `num_species`: Total number of species participating in simulation.
- `species_names`: List of the names of the chemical species. These are only used for labeling plot axes, so they can be whatever you like (e.g. “methane”, “Carbon Dioxide”, “H2O”). If none are provided, these will default to `["Species 1", "Species 2", ...]`.
- `mol_weights`: Molecular weights of each species. Must have `num_species` entries.

CPG Inputs

See *CPG Inputs* for variable types, default values, and units (where applicable).

- `enth_ref`: Reference enthalpy at 0 K of each species. Must have `num_species` entries.
- `cp`: Constant specific heat capacity at constant pressure for each species. Must have `num_species` entries.
- `pr`: Prandtl number of each species. Must have `num_species` entries.
- `sc`: Schmidt number of each species. Must have `num_species` entries.
- `temp_ref`: Reference dynamic viscosity of each species for Sutherland’s law. Must have `num_species` entries.
- `mu_ref`: Reference temperature of each species for Sutherland’s law. If `temp_ref[i] = 0` for any species, it will be assumed that its dynamic viscosity is constant and equal to `mu_ref[i]`. Must have `num_species` entries.

Finite Rate Irreversible Reaction Inputs

See *Finite Rate Irreversible Reaction Inputs* for variable types, default values, and units (where applicable).

- `nu`: List of lists of irreversible reaction stoichiometric coefficients, where each sublist corresponds to a single reaction. Reactants should have positive values, while products should have negative values.
- `nu_arr`: List of lists of irreversible reaction molar concentration exponents for all chemical species, where each sublist corresponds to a single reaction. Those chemical species that don't participate in the reaction should just be assigned a value of `0.0`.
- `act_energy`: List of Arrhenius rate activation energies E_a for each reaction.
- `pre_exp_fact`: List of Arrhenius rate pre-exponential factors.
- `temp_exp`: List of Arrhenius rate temperature exponents.

2.5.4 Piecewise Uniform IC File

See *Piecewise Uniform IC File* for variable types, default values, and units (where applicable).

- `x_split`: Location in spatial domain at which the piecewise uniform solution will be split. All cell centers with coordinates less than this value will be assigned to the "left" state, and those with coordinates greater than this value will be assigned to the "right" state.
- `press_left`: Static pressure in "left" state.
- `vel_left`: Velocity in "left" state.
- `temp_left`: Temperature in "left" state.
- `mass_fracs_left`: Species mass fractions in "left" state. Must contain `num_species` elements, and they must sum to 1.0.
- `press_right`: Static pressure in "right" state.
- `vel_right`: Velocity in "right" state.
- `temp_right`: Temperature in "right" state.
- `mass_fracs_right`: Species mass fractions in "right" state. Must contain `num_species_full` elements, and they must sum to 1.0.

2.5.5 rom_params.inp

See *rom_params.inp* for variable types, default values, and units (where applicable). We again break down some distinct sections of the file.

- `rom_method`: Name of the ROM method to use.
  - Valid options: `galerkin`, `lspg`, `mplsvt`
- `var_mapping`: Name of the state variable mapping which the ROM models employ.
  - Valid options: `conservative`, `primitive`
- `space_mapping`: Name of the mapping type which maps from the latent space to the full-order space.
  - Valid options: `linear`, `autoencoder`
- `num_models`: Number of distinct models used to make predictions for the full physical state. For example, if there is one model to predict the pressure and velocity fields, and another to predict the temperature and mass fraction fields, then `num_models = 2`

- `latent_dims`: A list containing the latent dimension for each model. If using a model with a fixed latent dimension (e.g. autoencoders), this will be checked against the model object and the code will terminate with an error if the values do not match
- `model_var_idxs`: A list of lists where each sublist contains the zero-indexed state variable numbers to which each model maps. The variable numbers are ordered by density/pressure, momentum/velocity, energy/temperature, and density-weighted mass fraction/mass fraction (as ordered in the `chem_file`). For example, in a ROM with two models, if the first model maps to velocity and mass fraction, and the second model maps to pressure and temperature, then `model_var_idxs = [[1,3],[0,2]]`.
- `model_dir`: Absolute path of the base under which model files and feature scaling profiles are stored.
- `cent_ic`: Boolean flag to set `cent_cons`/`cent_prim` (depending on the ROM method) to the provided initial condition profile. This is simply a convenience parameter that is useful when performing parametric predictions and don't want to repeatedly change the centering profile address.
- `norm_sub_cons`: List of paths relative to `model_dir` to the subtractive normalization NumPy binary profiles for feature scaling of the conservative state variables with which each model is associated. For example, if a model is associated with density/pressure and energy/temperature, then the corresponding entry in `norm_sub_cons` should be for the subtractive normalization profiles for the density and energy fields.
- `norm_fac_cons`: List of paths relative to `model_dir` to the factor normalization NumPy binary profiles for feature scaling of the conservative state variables with which each model is associated. For example, if a model is associated with density/pressure and energy/temperature, then the corresponding entry in `norm_fac_cons` should be for the factor normalization profiles for the density and energy fields.
- `cent_cons`: List of paths relative to `model_dir` to the centering NumPy binary profiles for feature scaling of the conservative state variables with which each model is associated. For example, if a model is associated with density/pressure and energy/temperature, then the corresponding entry in `cent_cons` should be for the centering profile for the density and energy fields.
- `norm_sub_prim`: List of paths relative to `model_dir` to the subtractive normalization NumPy binary profiles for feature scaling of the primitive state variables with which each model is associated. For example, if a model is associated with pressure and temperature, then the corresponding entry in `norm_sub_prim` should be for the subtractive normalization profile for the pressure and temperature fields.
- `norm_fac_prim`: List of paths relative to `model_dir` to the factor normalization NumPy binary profiles for feature scaling of the primitive state variables with which each model is associated. For example, if a model is associated with pressure and temperature, then the corresponding entry in `norm_fac_prim` should be for the factor normalization profile for the pressure and temperature fields.
- `cent_prim`: List of paths relative to `model_dir` to the centering NumPy binary profiles for feature scaling of the primitive state variables with which each model is associated. For example, if a model is associated with pressure and temperature, then the corresponding entry in `cent_prim` should be for the centering profile for the pressure and temperature fields.

Linear Space Mapping Inputs

See *Linear Space Mapping Inputs* for variable types, default values, and units (where applicable).

- `basis_files`: List of paths relative to `model_dir` to the linear trial basis NumPy binary (`*.npy`) files for each model.

Autoencoder Space Mapping Inputs

See *Autoencoder Space Mapping Inputs* for variable types, default values, and units (where applicable).

- `decoder_files`: List of paths relative to `model_dir` to the decoder model objects for each model.
- `encoder_files`: List of paths relative to `model_dir` to the encoder model objects for each model.
- `decoder_isconv`: Boolean flag indicating whether the output of the decoder is a convolutional layer. If this is `True`, then `decoder_io_format` must be specified.
- `decoder_io_format`: The expected array axis ordering of the state profiles on which the decoder operates, if `decoder_isconv = True`. See *Neural Networks* for more details.
  - Valid options: `"channels_first"`, `"channels_last"`
- `encoder_isconv`: Boolean flag indicating whether the output of the encoder is a convolutional layer. If this is `True`, then `encoder_io_format` must be specified.
- `encoder_io_format`: The expected array axis ordering of the state profiles on which the encoder operates, if `encoder_isconv = True`. See *Neural Networks* for more details.
  - Valid options: `"channels_first"`, `"channels_last"`

Machine Learning Library Inputs

See *Machine Learning Library Inputs* for variable types, default values, and units (where applicable).

- `ml_library`: Name of the machine learning library which was used to train and serialize any machine learning models to be used in the ROM.
  - Valid options: `tfkeras`
- `run_gpu`: Boolean flag to determine whether to run machine learning model inference on the GPU. Please note that running on the CPU is often faster than running on the GPU for these small 1D problems, as memory movement between the host and device can be extremely slow and all memory movement operations are blocking.

2.6 Miscellanea

Below are details on **PERFORM** features that don't fit neatly into other documentation sections.

2.6.1 Running in "Steady" Mode

Setting the Boolean flag `run_steady = True` in `solver_params.inp` slightly alters the solver behavior to run in a sort of "steady-state solver" mode. To be completely clear, **PERFORM** is an unsteady solver and there are no true steady solutions for the types of problems it is designed to simulate. However, this "steady" mode is designed specifically for solving flame simulations in which bulk advection is carefully balanced with chemical diffusion and reaction forces, resulting in a roughly stationary flame, which is as close to a steady solution as one can expect for these cases. This stationary flame acts as a good "mean" flow for stationary flame problems with external forcing, or as an initial condition for transient flame problems (combined with a non-zero `vel_add`).

The exact changes in behavior are as follows:

- The ℓ^2 and ℓ^1 norms displayed in the terminal is the norm of the change in the primitive state between physical time steps. This is opposed to no residual output for explicit time integration schemes, or the linear solve residual norm for implicit time integration schemes.

- The time history of the above residual norms will be written to the file `working_dir/unsteady_field_results/steady_convergence.dat`.
- The solver will terminate early if the ℓ^2 norm of the solution change converges below the tolerance set by `steady_tol` in `solver_params.inp`.

This "steady" solver can be run for both explicit and implicit time integration schemes. The procedure for obtaining "steady" flame solutions is incredibly tedious, generally requiring carefully manually tuning the boundary conditions to achieve a certain inlet velocity until a point at which the advection downstream is balanced with the diffusion and reaction moving upstream. During this tuning procedure, the user often must visually confirm that the flame is not moving by watching the field plots closely. Again, this process is incredibly tedious, but the "steady" solver helps facilitate this by providing at least one quantitative metric for determining if a steady flame solution has been achieved.

2.7 Issues and Contributing

If you experience errors or unexpected solver behavior when running **PERFORM**, please first double-check your input parameters and use this documentation as a reference for proper input file formatting. If problems persist, please create a new issue on the GitHub repository, and we'll do our best to resolve it. However, if a submitted issue is related to a significant *expansion* of code capabilities (e.g. adding a new ROM model or flux scheme), we probably won't work on it.

On the other hand, if you would like to personally work to expand the code and contribute to **PERFORM**, first of all thank you! Please fork the repository under your own GitHub account and implement your contributions. When you're finished, create a pull request against the main repository and your changes will be reviewed before being merged. You can manually check beforehand that your changes do not break code by running Bash script `perform/tests/run_tests.sh`, which will execute unit, integration, and regression tests. You can also let GitHub Actions automatically run these tests when you push changes to the remote repo or submit a pull request.

2.8 Governing Equations

PERFORM solves the 1D Navier-Stokes equations with chemical species transport and a chemical reaction source term. This can be formulated as

$$\frac{\partial \mathbf{q}}{\partial t} + \frac{\partial}{\partial x}(\mathbf{f} - \mathbf{f}_v) = \mathbf{s}$$

$$\mathbf{q} = \begin{bmatrix} \rho \\ \rho u \\ \rho h^0 - p \\ \rho Y_l \end{bmatrix}, \quad \mathbf{f} = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho h^0 u \\ \rho Y_l \end{bmatrix}, \quad \mathbf{f}_v = \begin{bmatrix} 0 \\ \tau \\ u\tau - q \\ -\rho V_l Y_l \end{bmatrix}, \quad \mathbf{s} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \dot{\omega}_l \end{bmatrix}$$

where $\mathbf{q}$ is the conservative state, $\mathbf{f}$ is the inviscid flux vector, $\mathbf{f}_v$ is the viscous flux vector, and $\mathbf{s}$ is the source term. Additionally, ρ is density, u is velocity, h^0 is stagnation enthalpy, p is static pressure, and Y_l is the mass fraction of the lth chemical species. For a system with N_Y chemical species, only $N_Y - 1$ species transport equations are solved, as the final species mass fraction Y_{N_Y} can be computed from the fact that all mass fractions must sum to unity.

The spatial domain is discretized by the finite volume method. Brief notes on available flux calculations schemes are given in *Flux Schemes*. Descriptions of gradient limiters for higher-order schemes are given in *Gradient Limiters*. Available boundary conditions are described in *Boundary Conditions*. Available numerical time integrators are detailed in *Time Integrators*.

Some details on available gas models for calculating relevant thermodynamic and transport properties are given in *Gas Models*. Models for calculating the source term $\dot{\omega}_l$ are detailed in *Reaction Models*.

Details on each of these topics are provided in the theory documentation. Additionally, those interested in how this theory may be extended to higher dimensions and to more complex gas/reaction models are directed to Matthew Harvazinski's thesis [Har12]. This details the inner mechanics of **GEMS**, the high-fidelity 3D combusting flow solver which **PERFORM**'s baseline solver is based off of.

2.9 Flux Schemes

This section outlines the various schemes available for computing the inviscid and viscous fluxes of the 1D Navier-Stokes equations with species transport. For details on the mathematics of each scheme, please refer to the theory documentation.

2.9.1 Inviscid Flux Schemes

Roe Scheme

This inviscid flux scheme is activated by setting `invisc_flux_scheme = "roe"` in `solver_params.inp`. The Roe scheme follows the approximate Riemann solver of Philip Roe [Roe81]. As of the writing of this section, the code is not capable of applying an entropy fix for locally-sonic flows, but will be available in a forthcoming release.

2.9.2 Viscous Flux Schemes

Inviscid Scheme

This viscous flux scheme is activated by setting `visc_flux_scheme = "invisc"` in `solver_params.inp`. This scheme simply neglects all contributions from the viscous flux terms.

Standard Viscous Scheme

This viscous flux scheme is activated by setting `visc_flux_scheme = "standard"` in `solver_params.inp`. This scheme computes the contribution from the viscous fluxes terms from the average state at each cell face (in the case of the Roe flux, the Roe average) and face gradients computed by a second-order accurate finite difference stencil. The viscous fluxes are then computed directly, with the sole approximation of the diffusion velocity term given by

$$V_l Y_l = -D_{l,M} \frac{\partial Y_l}{\partial x}$$

where $D_{l,M}$ is the mass diffusion coefficient for the lth species diffusing into the mixture. This approximation is inserted into both the heat flux and species transport viscous flux term. Please see the theory documentation for calculation of the diffusion coefficients

The calculation of individual species diffusion velocities is incredibly expensive, necessitating this approximation. However, this may lead to violations of mass conservation in solving the species transport equations. A correction velocity term is included which helps mitigate this, and is described in detail in the theory documentation.

2.10 Gradient Limiters

For higher-order face reconstructions with a fixed stencil, gradient limiters are required to prevent excessive oscillations near sharp gradients. When initializing simulations from a strong step function (`ic_params_file`), a gradient limiter is practically required to ensure simulation stability (at least for the first few time steps).

2.10.1 Barth-Jespersen Limiter

The Barth-Jespersen gradient limiter [BJ89] is activated by setting either `grad_limiter = "barth_cell"` or `grad_limiter = "barth_face"` in `solver_params.inp`. The former computes the limiter based on unconstrained state reconstructions at neighboring cell centers, while the latter does so at the cell faces. This limiter guarantees that no new local maxima or minima are created by the higher-order face reconstructions. However, the gradient limiter calculation is non-differentiable (due to a minimum function), which can negatively affect convergence of the solver.

2.10.2 Venkatakrishnan Limiter

The Venkatakrishnan gradient limiter [Ven93] is activated by setting `grad_limiter = "venkat"` in `solver_params.inp`. The Venkatakrishnan limiter improves on the Barth-Jespersen limiter by replacing the non-differentiable minimum function with a smooth polynomial function. This has the effect of improving solver convergence, but has the negative consequence of limiting the solution in smooth regions and more aggressively smoothing discontinuities.

2.11 Boundary Conditions

Boundary conditions in **PERFORM** are enforced by an explicit ghost cell formulation. As noted in *solver_params.inp*, the requirements and interpretations of the `*_inlet` and `*_outlet` input parameters depend on the boundary condition specified by `bound_cond_inlet` / `bound_cond_outlet`. Valid options of `pert_type` for each boundary condition are also specified. If an invalid entry for `pert_type` is supplied, it will simply be ignored. For the mathematical definitions of these boundary conditions, please refer to the solver theory documentation.

In the following sections, we provide which additional input parameters are required and how they are interpreted for each valid entry of `bound_cond_inlet/bound_cond_outlet`. Additionally, the acceptable values to be artificially perturbed for each boundary condition are given under `pert_type_inlet/pert_type_outlet`.

2.11.1 Inlet BCs and Parameters

Fixed Stagnation Temperature and Pressure Inlet

This boundary condition is activated by setting `bound_cond_inlet = "stagnation"` in `solver_params.inp`. This boundary specifies the upstream stagnation temperature and stagnation pressure, i.e. the temperature and pressure of the fluid when its velocity is brought to zero. Tho boundary condition results in reflections of acoustic waves and should not be used with unsteady calculations with significant system acoustics.

The applicable boundary condition input parameters are as follows:

- `press_inlet`: Specified stagnation pressure at the inlet.
- `temp_inlet`: Specified stagnation temperature at the inlet.
- `mass_fracs_inlet`: Fixed mixture composition at the inlet.

Full State Specification Inlet

This boundary condition is activated by setting `bound_cond_inlet = "fullstate"` in `solver_params.inp`. This boundary conditions overspecifies the boundary condition by fixing the inlet ghost cell primitive state. This is not a useful boundary condition for unsteady calculations (unless the flow is supersonic), but is mostly useful for testing how an outlet boundary condition responds to perturbations propagating downstream.

The applicable boundary condition input parameters are as follows:

- `press_inlet`: Fixed static pressure at the inlet.
- `vel_inlet`: Fixed velocity at the inlet.
- `temp_inlet`: Fixed static temperature at the inlet.
- `mass_fracs_inlet`: Fixed mixture composition at the inlet
- `pert_type_inlet` (optional): Accepts `"pressure"`, `"velocity"`, or `"temperature"` to perturb the values of the appropriate fixed quantity.

Mean Flow Inlet

This boundary condition is activated by setting `bound_cond_inlet = "meanflow"` in `solver_params.inp`. This boundary condition provides a non-reflective inlet that requires some sense of a mean flow (or the flow infinitely far upstream) about which the unsteady flow is simply a perturbation. It effectively fixes the incoming characteristics while allowing the outgoing characteristics to be transmitted outside the domain without acoustic reflections.

The applicable boundary condition input parameters are as follows:

- `press_inlet`: Specified mean upstream static pressure.
- `temp_inlet`: Specified mean upstream static temperature.
- `mass_fracs_inlet`: Fixed mixture composition at the inlet.
- `vel_inlet`: Specified mean upstream value of ρc, where c is the sound speed.
- `rho_inlet`: Specified mean upstream value of ρc_p, where c_p is the specific heat capacity at constant pressure.
- `pert_type_inlet` (optional): Accepts `"pressure"` to perturb the mean upstream pressure.

2.11.2 Outlet BCs and Parameters

Fixed Static Pressure Outlet

This boundary condition is activated by setting `bound_cond_outlet = "subsonic"` in `solver_params.inp`. This boundary condition fixes the static pressure at the outlet. As with the stagnation temperature and pressure inlet, this boundary condition produces acoustic reflections at the outlet.

The applicable boundary condition input parameters are as follows:

- `press_outlet`: Specified static pressure at the outlet.
- `mass_frac_outlet`: Fixed mixture composition at the outlet.
- `pert_type_outlet` (optional): Accepts `"pressure"` to perturb the pressure at the outlet.

Mean Flow Outlet

This boundary condition is activated by setting `bound_cond_outlet = "meanflow"` in `solver_params.inp`. This boundary condition, as with the mean flow inlet boundary condition, fixes the incoming characteristic while transmitting the outgoing characteristics without reflections. Again, it requires some sense of a mean flow (or the flow infinitely far downstream) about which the unsteady flow is simply a perturbation.

The applicable boundary condition input parameters are as follows:

- `press_outlet`: Specified mean downstream static pressure.
- `vel_outlet`: Specified mean downstream value of ρc, where c is the sound speed.
- `rho_outlet`: Specified mean downstream value of ρc_p, where c_p is the specific heat capacity at constant pressure.
- `pert_type_outlet` (optional): Accepts `"pressure"` to perturb the mean downstream pressure.

2.11.3 Boundary Perturbations

Setting valid values for `pert_type_inlet` or `pert_type_outlet`, as well as non-zero values of `pert_perc_inlet` / `pert_freq_inlet` or `pert_perc_outlet` / `pert_freq_outlet`, initiates external forcing at the appropriate boundary. The perturbation signal is a simple sinusoid, given for a given perturbed quantity α in the boundary ghost cell as

$$\alpha(t) = \overline{\alpha} \left(1 + A \sum_{i=1}^{N_f} \sin(2\pi f_i t) \right)$$

where $\overline{\alpha}$ is the relevant reference quantity given in `solver_params.inp`, f_i are the signal frequencies in `pert_freq_inlet/pert_freq_outlet`, and A is the amplitude percentage `pert_perc_inlet/pert_perc_outlet`.

For example, if the user sets (among other required parameters)

```
bound_cond_outlet = "meanflow"
press_outlet      = 1.0e6
pert_type_outlet  = "pressure"
pert_perc_outlet  = 0.05
pert_freq         = [2000.0, 5000.0]
```

this will result in two perturbation signals (one of 2 kHz, another of 5 kHz) of the mean downstream static pressure with amplitude of 50 kPa.

2.12 Time Integrators

This section briefly describes the various numerical time integrators that are available in **PERFORM**. For details on each scheme, please refer to the theory documentation.

2.12.1 Explicit Integrators

Explicit time integrators depend only on prior time steps, and thus no not require the solution of a linear system. As there is no concept of a linear solve residual as in the iterative solution of implicit schemes, **PERFORM** will simply report the time step iteration number in the terminal. Explicit schemes are generally less stable than implicit schemes and require smaller time steps, especially for well-resolved combustion problems. However, given the relative cost of the implicit solve Jacobian calculations and linear system solution in **PERFORM**, you can sometimes achieve a cheaper solution with an explicit scheme with a smaller time step over an implicit scheme with a larger time step.

Classic RK4

The classic RK4 scheme is activated by setting `time_scheme = "classic_rk4"` in `solver_params.inp`. This is the classic fourth-order accurate explicit Runge-Kutta scheme, originally proposed by Martin Kutta in 1901. The Butcher tableau for this scheme is given below.

$$
\begin{array}{c|cccc}
0 & 0 & 0 & 0 & 0 \\
1/2 & 1/2 & 0 & 0 & 0 \\
1/2 & 0 & 1/2 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 \\
\hline
 & 1/6 & 1/3 & 1/3 & 1/6
\end{array}
$$

Strong Stability-preserving RK3

The strong stability-preserving RK3 (SSPRK3) scheme is activated by setting `time_scheme = "ssp_rk3"` in `solver_params.inp`. Strong stability-preserving methods are named in reference to preserving the strong stability properties of the forward Euler scheme while providing higher orders of accuracy. This scheme provides such a third-order accurate scheme. The Butcher tableau for this scheme is given below.

$$
\begin{array}{c|ccc}
0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 \\
1/2 & 1/4 & 1/4 & 0 \\
\hline
 & 1/6 & 1/6 & 2/3
\end{array}
$$

Jameson Low-storage Scheme

The Jameson low-storage scheme by setting `time_scheme = "jameson_low_store"` in `solver_params.inp`. What we refer to as the “Jameson low-storage scheme” is simply a scheme presented by Jameson which, while only appropriate for steady calculations, is a vast simplification over of the typical Runge-Kutta methods and extensible to arbitrary orders of accuracy. For an s-stage RK scheme, the ith stage calculation is given by

$$q^i = q^n - \frac{\Delta t}{s + 1 - i} f(q^{i-1})$$

Thus, the scheme only requires the solution at the previous time step and the RHS evaluation at the previous time step, greatly decreasing the storage cost of the scheme.

2.12.2 Implicit Integrators

Implicit time integrators depend on the system solution at future time steps, and thus can only be solved approximately. **PERFORM** uses Newton's method to iteratively solve the fully-discrete system. Newton's method is repeatedly applied until the ℓ^2 norm of the linear solve residual converges below the threshold given by `res_tol`, or `subiter_max` iterations are computed. Implicit time integrators generally exhibit excellent stability properties at relatively large time steps. As such, they are well-suited to combustion problems, which are typically extremely stiff due to the strong exponential non-linearity arising from the reaction source term. However, the cost of computing the Jacobian of the RHS function and solving the stiff linear system can be quite expensive relative to the cost of computing the RHS side for simple 1D problems.

Backwards Differentiation Formula

Backwards differentiation formula (BDF) schemes are activated by setting `time_scheme = "bdf"` in `solver_params.inp`. BDF schemes are a particular class of linear multi-step schemes. As of the writing of this section, **PERFORM** is capable of computing the first-order, second-order, third-order, and fourth-order accurate BDF schemes. However, it is not advised to use anything higher than the second-order accurate scheme (`time_order = 2`), as these higher-order accurate schemes can sometimes be unstable.

2.12.3 Dual Time-stepping

Dual time-stepping [VM95] is activated by using implicit time integration scheme and setting `dual_time = True` in `solver_params.inp`. This method is a time integration method which adds a pseudo-time derivative to the governing equations,

$$\Gamma \frac{\partial \mathbf{q}_p}{\partial \tau} + \frac{\partial \mathbf{q}}{\partial t} + \frac{\partial}{\partial x}(\mathbf{f} - \mathbf{f}_v) = \mathbf{s}$$

where τ is the pseudo-time variable, $\mathbf{q}_p = [p \ u \ T \ Y_l]^\top$ are the primitive variables, and $\Gamma = \partial \mathbf{q} / \partial \mathbf{q}_p$. Numerical integration of these equations with an implicit solver has two beneficial effects: the pseudo-time term has the effect of regularizing the linear solve, improving its stability, and the primitive state $\mathbf{q}_p$ can be solved for directly, instead of the conservative state $\mathbf{q}$. This latter point is particularly key for reacting systems, as computing the primitive state from the conservative state can be extremely challenging when using a thermally-perfect or real gas model.

2.13 Gas Models

This section describes the models available for describing the thermodynamic and transport properties of gases. As of the writing of this section, there are no plans to include real gas models, and so all models will make the perfect gas assumption. This means that all intermolecular forces are neglected, and the gas is governed by the ideal gas law

$$p = \rho R T$$

The two gas models which are planned to be included in **PERFORM** differ in how thermodynamic properties (e.g. enthalpy, entropy) and transport properties (e.g. dynamic viscosity, diffusion coefficients) are calculated.

2.13.1 Calorically-perfect Gas

The calorically perfect gas model is activated by setting `gas_model = "cpg"` in the `chem_file`. The CPG model assumes that the heat capacity at constant pressure for each species is constant, i.e. $c_{p,l}(T) = c_{p,l}$. These values are given in the `chem_file` via `cp`. The species enthalpies are thus given by

$$h_l = h_{ref,l} + c_{p,l} T$$

where the reference enthalpies at 0 K are given in the `chem_file` via `enth_ref`.

Species dynamic viscosities are computed via Sutherland's law,

$$\mu_l(T) = \mu_{ref,l} \left(\frac{T}{T_{ref,l}} \right)^{3/2} \left(\frac{T_{ref,l} + S}{T + S} \right)$$

where the species reference temperatures are given in the `chem_file` via `temp_ref`, and the reference viscosities via `mu_ref`. The Sutherland temperature is given as $S = 110.4$ K. If $T = 0$ K, then $\mu_l = \mu_{ref,l}$.

The species thermal conductivities (required for calculating the heat flux) are given by

$$K_l = \frac{\mu_l c_{p,l}}{\mathrm{Pr}_l}$$

Where the species Prandtl numbers Pr_l are given in the `chem_file` via `pr`. The binary diffusion coefficients of each species into the mixture is given by

$$D_{l,M} = \frac{\mu_l}{\rho \mathrm{Sc}_l}$$

where the species Schmidt number Sc_l are given in the `chem_file` via `sc`.

Additional details on the CPG gas model, particularly on computing the mixture thermodynamic and transport properties, can be found in the theory documentation.

2.13.2 Thermally-perfect Gas

Coming soon!

2.14 Reaction Models

This section briefly describes the reaction models available in **PERFORM**. For a comprehensive description of combustion mechanics well beyond the reaction models covered here (though also including them), we direct the reader to *Combustion* by Glassman and Yetter [GY08].

2.14.1 Finite-rate Mechanisms

Finite-rate reactions generally describe mixtures in chemical non-equilibrium, opposed to infinitely fast chemistry in which reactions are assumed to proceed to completion instantaneously. Finite-rate mechanisms must, as the name implies, compute finite reaction rates.

To preface this, we begin with the general form of the mth reaction (in a set of N_r reactions) between N_Y chemical species, given by

$$\sum_{l=1}^{N_Y} \nu'_{l,m} \chi_l \underset{k_{f,m}}{\overset{k_{r,m}}{\leftrightarrows}} \sum_{l=1}^{N_Y} \nu''_{l,m} \chi_l$$

where χ_l is the chemical formula for the lth chemical species, and $\nu'_{l,m}$ and $\nu''_{l,m}$ are the stoichiometric coefficients of the reactants and products, respectively. These stoichiometric coefficients are input into **PERFORM** as the difference between the reactant coefficient and the product coefficient, i.e. $\nu_{l,m} = \nu'_{l,m} - \nu''_{l,m}$, via `nu` in the `chem_file`.

The coefficients $k_{f,m}$ and $k_{r,m}$ are the forward and reverse reaction rates for the mth reaction. The forward reaction rate is computed as an Arrhenius rate, given by the formula

$$k_{f,m} = A_m T^{b_m} \exp\left(\frac{-E_{a,m}}{R_u T}\right)$$

where the coefficients A_m, b_m, and $E_{a,m}$ are tabulated constants given by the reaction mechanism, given by `pre_exp_fact`, `temp_exp`, and `act_energy`, respectively, in the `chem_file`.

The reaction source term $\dot{\omega}_l$ introduced in *Governing Equations* is computed as a function of reaction "rates of progress" w_m

$$\dot{\omega}_l = W_l \sum_{m=1}^{N_r} (\nu''_{l,m} - \nu'_{l,m}) w_m$$

where W_l is the molecular weight of the lth species. The following methods are concerned with the calculation of these rates of progress.

Irreversible Mechanisms

The irreversible reaction mechanism model is activated by setting `reaction_model = "fr_irrev"` in the `chem_file`. An irreversible finite-rate mechanism assumes that reactions only proceed in the forward direction, i.e. converting reactants to products and neglecting the reverse reaction rate $k_{r,m}$. The rate of progress for the mth reaction is given by

$$w_m = k_{f,m} \prod_{l=1}^{N_Y} [X_l]^{\tilde{\nu}_{l,m}}$$

where $[X_l]$ is the molar concentration of the lth species. Additionally, $\tilde{\nu}_{l,m}$ are tabulated constants for each species and reaction which are input in the `chem_file` via `nu_arr`.

Irreversible reactions vastly simplify the calculation of the reaction source term, at the expense of accuracy. The exponential constants $\tilde{\nu}_{l,m}$ are empirically-determined and may not be accurate under all flow and reaction regimes. The reduced cost of these mechanisms is often extremely attractive, and errors incurred by their approximations may be within acceptable limits.

Reversible Mechanisms

Coming soon!

2.15 Reduced-order Modeling

Reduced-order modeling, in a broad sense, aims to decrease the computational cost and complexity of numerical solutions of systems governed by ordinary differential equations by vastly reducing the number of degrees of freedom being solved for. For practical engineering systems, the number of degrees of freedom arising from complex governing equations and well-refined computational meshes can easily reach the tens or hundreds of millions. The solution of these systems require powerful supercomputers which consume vast amounts of electricity, money, and manpower to operate and maintain. A means of reducing this cost of computing the full-order model (FOM) by reducing the number of degrees of freedom, with minimal loss of accuracy, is thus highly sought after.

Practical combustion systems (e.g. gas turbines, rocket engines) are particularly challenging, as the characteristic spatio-temporal scales of flames are orders of magnitude smaller than those of non-reacting fluid flows. Simulations of such systems thus require highly-refined meshes and tiny time step sizes, resulting in exorbitant computational costs to simulate mere milliseconds of physical time. As such, CFD has yet to play a significant role in the design and control of practical combustion systems. The goal of the US Air Force Center of Excellence funding this project is to explore suitable methods of model order reduction for rocket combustor applications, but applies broadly to practical combustion systems.

Reduced-order models (ROMs) generally achieve order reduction by manipulating the full-order governing equations or solving a surrogate model for which there is an analytical mapping from a non-physical low-dimensional surrogate state to the physical full-dimensional state. This is opposed to “reduced-fidelity” models which reduce the number of degrees of freedom in a more empirical fashion. Such methods might include using a reduced reaction mechanism, or simply reducing the computational mesh refinement.

We can broadly classify ROMs into two categories: intrusive and non-intrusive methods.

2.15.1 Intrusive ROMs

Intrusive ROMs require direct access in some part to the FOM numerical solver routines. These methods generally manipulate the FOM governing equations, e.g. projecting the system onto a low-dimensional subspace. Obviously, intrusive ROMs can be time-consuming to develop and run, as they require intimate knowledge of the FOM solver and the ability to alter its source code. Furthermore, for general non-linear systems, intrusive ROMs may not even achieve any cost reduction, and require additional “hyper-reduction” methods to effectively reduce the computational cost. Despite these drawbacks, intrusive ROMs do allow tighter control over the ROM solution, as the solver obviously has access to vast amounts of information on the system physics and numerics. This allows the method to steer and control of the solution more actively, and to more easily apply physics-informed constraints to the system.

Many auxillary methods are being actively researched to improve the cost, accuracy, and robustness of intrusive ROMs. A multitude of mappings from the low-dimensional representation to the full-dimensional state seek to improve ROM accuracy. Alternative projection methods for projection-based ROMs have been proposed. Closure models attempt to model the information that has been lost the order-reduction approximation, much like turbulence closure models. Filtering methods filter out unstable dynamics of ROM systems, while artificial viscosity methods damp the unstable dynamics.

2.15.2 Non-intrusive ROMs

Non-intrusive ROMs, on the other hand, do **not** require access to the FOM numerical solver. These methods generally learn a surrogate model from FOM data, and may not even require a numerical time integration scheme to make predictions for the evolution of the full-dimensional state. The allure of these models are twofold. First, they are often extremely simple to develop and deploy, as they do not require solver routines for complex numerical schemes. Furthermore, these models typically fit into the memory of a single computational node, avoiding the need for complicated distributed-memory code. Second, they usually incur much lower computational cost than intrusive ROMs, as they do not require the costly evaluation of fluxes, source terms, or large linear solves. These models may be capable of running in seconds on a single CPU core on a laptop, which makes them far more attractive for many-query applications such as parametric design or uncertainty quantification. However, without any knowledge of the physical system it is modeling, a non-intrusive ROM may easily generate non-physical solutions and lack any means of controlling such deviations.

2.15.3 ROMs in PERFORM

PERFORM is specifically designed to allow for the rapid prototyping of new ROM methods with minimal effort on the part of the developer, with the intent of providing a general framework for both intrusive and non-instrusive ROMs. The hope is that this code can be used by members of the ROM community to quickly implement their ROM methods and test them for a suite of interesting multi-species and reacting 1D flow problems.

As of the writing of this section, **PERFORM** is capable of computing linear and non-linear projection-based ROMs using several projection methods. Several non-intrusive ROMs and closure models will be added in the near future.

If you have questions on how to best implement your ROM method in **PERFORM**, please feel free to start a new issue on the Github page with a brief description of the method and a paper on the method. We can give you some suggestions on how it might be most seamlessly integrated into the current class hierarchy, and you're welcome to make a pull request once you're finished coding it up.

2.16 ROM Input Files

This section outlines the various input files that are required to run ROMs in **PERFORM**, as well as the input parameters that are used in text input files. If you are having issues running a case (in particular, experiencing a `KeyError` error), please check that all of your input parameters are set correctly, and use this page as a reference. Text files inputs should be formatted exactly as described in *Inputs*.

Below, the formats and input parameters for each input file are described. For text file inputs, tables containing all possible parameters are given, along with their expected data type, default value and expected units of measurement (where applicable). Note that expected types of `list` of `lists` is abbreviated as `lol` for brevity. For detailed explanations of each parameter, refer to *Input Parameter Index*.

2.16.1 rom_params.inp

The `rom_params.inp` file is a text file containing input parameters for running ROM simulations. It specifies all parameters related to the ROM model and number of models, the latent dimension of each model, and the paths to model input files and standardization profiles. **It must be placed in the working directory alongside `solver_params.inp` , and must be named `rom_params.inp`**. Otherwise, the code will not function.

The table below provides input parameters which may be required by any ROM method. Input parameters which are specific to the neural network autoencoder ROMs are given in *Autoencoder Space Mapping Inputs*.

Table 7: `rom_params.inp` input parameters

Parameter	Type	Default	Units
`rom_method`	`str`	-	-
`var_mapping`	`str`	-	-
`space_mapping`	`str`	-	-
`num_models`	`int`	-	-
`latent_dims`	`list of int`	`[0]`	-
`model_var_idxs`	`lol of int`	`[[-1]]`	-
`model_dir`	`str`	-	-
`cent_ic`	`bool`	`False`	-
`norm_sub_cons`	`list of str`	`[""]`	-
`norm_fac_cons`	`list of str`	`[""]`	-
`cent_cons`	`list of str`	`[""]`	-
`norm_sub_prim`	`list of str`	`[""]`	-
`norm_fac_prim`	`list of str`	`[""]`	-
`cent_prim`	`list of str`	`[""]`	-

Linear Space Mapping Inputs

The parameters described here may be used in `rom_params.inp` when applying a linear space mapping.

Table 8: Linear space mapping input parameters

Parameter	Type	Default	Units
`basis_files`	list of `str`	-	-

Autoencoder Space Mapping Inputs

The parameters described here may be used in `rom_params.inp` when applying an autoencoder space mapping.

Table 9: Autoencoder space mapping input parameters

Parameter	Type	Default	Units
`decoder_files`	list of `str`	-	-
`encoder_files`	list of `str`	-	-
`decoder_isconv`	`bool`	`False`	-
`decoder_io_format`	`str`	`None`	-
`encoder_isconv`	`bool`	`False`	-
`encoder_io_format`	`str`	`None`	-

Machine Learning Library Inputs

The parameters described here may be used in `rom_params.inp` when using any ROM method which requires machine learning models.

Table 10: Machine learning library input parameters

Parameter	Type	Default	Units
`ml_library`	`str`	-	-
`run_gpu`	`bool`	`False`	-

2.16.2 Feature Scaling Profiles

Feature scaling is a routine procedure in data science for ensuring that the datasets used to train a model are normalized and no specific feature is given an inordinate amount of weight in the training procedure. This addresses the wide range of magnitudes seeing in flow field data: pressure can be $\mathcal{O}(1e6)$, temperature can be $\mathcal{O}(1e3)$, velocity can be $\mathcal{O}(10)$, and species mass fraction is $\mathcal{O}(1)$. When data is ingested by the model or the model makes a prediction during the inference stage (e.g. ROM runtime), the same scaling procedure must be applied.

In **PERFORM**, ROM models are generally trained on and operate on snapshots of the conservative or primitive state profile. Data standardization of a solution profile (given here generally by $\mathbf{u}$) is computed as

$$\mathbf{u}' = \frac{\mathbf{u} - \mathbf{u}_{cent} - \mathbf{u}_{sub}}{\mathbf{u}_{fac}}$$

We refer to $\mathbf{u}_{cent}$ as the "centering" profile, $\mathbf{u}_{sub}$ as the "subtractive" normalization profile, and $\mathbf{u}_{fac}$ as the "factor" normalization profile. The reverse procedure, de-scaling, is simply given by

$$\mathbf{u} = \mathbf{u}' \odot \mathbf{u}_{fac} + \mathbf{u}_{cent} + \mathbf{u}_{sub}$$

Conservative and primitive state centering profiles are input via `cent_cons` and `cent_prim` in `rom_params.inp`, respectively. Conservative and primitive subtractive normalization profiles are input via `norm_sub_cons` and `norm_sub_prim` in `rom_params.inp`, respectively. Finally, the conservative and primitive factor normalization profiles are input via `norm_fac_cons` and `norm_fac_prim` in `rom_params.inp`, respectively.

It may seem strange to separate $\mathbf{u}_{cent}$ and $\mathbf{u}_{sub}$, as their repeated summation would simply be wasted FLOPS. Indeed, under the hood these profiles are summed and treated as a single profile at runtime. However, during the pre-processing stage it is generally easier for the user to treat these separately. For example, the centering profile may be the time-averaged mean profile or initial condition profile, while the normalization profiles may come from min-max scaling of the centered data. We thus allow the user this flexibility in deciding how to express these profiles.

2.16.3 Model Objects

We operate under the assumption that every ROM method provides some mapping from a low-dimensional representation of the state to the physical full-dimensional state, sometimes referred to as a “decoder.” We generalize this mapping to allow for multiple decoders which may map to a subset of the state variables, each with their own low-dimensional state. For example, a ROM method may provide two decoders, one which predicts the pressure and velocity fields, and another which predicts the temperature and species mass fraction fields. In various contexts this has been referred to as a “scalar” or “separate” ROM. The more traditional method of using a single decoder for the entire full-dimensional state, with only one low-dimensional state vector, is sometimes referred to as a “vector” or “coupled” ROM.

The total number of models is given by the `num_models` parameter in `rom_params.inp`, and the dimension of each model’s low-dimensional state is given by each entry in `latent_dims`. The zero-indexed state variables to which each model maps is given by each sublist in `model_var_idxs`. The model object(s) required for this decoding procedure are specified by mapping-specific input parameters (e.g. `basis_files` for a linear mapping, and `decoder_files` for an autoencoder mapping).

Linear Bases

For ROM models which require a linear basis representation (such as those described in *Linear Subspace Projection ROMs*), each model object located by `basis_files` in `rom_params.inp` is a three-dimensional NumPy binary (`*.npy`) containing the linear trial basis for that model. The first dimension is the number of state variables that the trial basis represents, the second dimension is the number of cells in the computational domain, and the third dimension is the number of trial modes generated by the basis calculation procedure. This final dimension is the *maximum* number of trial modes which may be requested via the corresponding entry in `latent_dims`.

Neural Networks

The model objects for neural network-based ROMs are specific to each network training framework (e.g. Keras, PyTorch). In general, they are serialized as a single file when saved to disk and can be deserialized at runtime.

The expected format in which an input neural network model interacts with field data is given by `*_isconv` and `*_io_format` in `rom_params.inp`. If `*_isconv = True`, it is assumed that the network layers which input/output state data are convolutional layers, which require that the field data have separated spatial and variable dimensions. The order of these dimensions in the neural network are given by `*_io_format`. As of the writing of this section, the only valid options are `"channels_first"` and `"channels_last"`. The former indicates that the neural network operates with field data arrays whose first dimension is the batch size, the second dimension is the number of state variables (“channels”), and the final channel is the spatial dimension. The latter swaps the channel dimension and spatial dimension ordering. If `*_isconv = False`, it is assumed that field data is in “flattened” format when input/output to the neural network model.

TensorFlow-Keras Autoencoders

TensorFloat-Keras autoencoders must be serialized separately as an encoder and a decoder via the `model.save()` function. As of the writing of this section, only the older Keras HDF5 format (`*.h5`) can be loaded by **PERFORM**. The decoder files are located via `decoder_files` in `rom_params.inp`, while the encoder files (which are only required when initializing the low-dimensional solution from the full-state solution or when `encoder_jacob = True`) are located via `encoder_files`.

NOTE: if running with `run_gpu = False` (making model inferences on the CPU), note that TensorFlow convolutional layers cannot handle a `channels_first` format. If your network format conforms to `*_io_format = "channels_first"`, the code will terminate with an error. This issue could theoretically be fixed by the user by including a permute layer to change the layer input ordering to `channels_last` before any convolutional layers, but we err on the side of caution here.

2.17 Linear Subspace Projection ROMs

We begin describing linear projection ROMs by defining a general non-linear ODE which governs our dynamical system, given by

$$\frac{d\mathbf{q}}{dt} = \mathbf{R}(\mathbf{q})$$

where for ODEs describing conservation laws, $\mathbf{q} \in \mathbb{R}^N$ is the conservative state, and the non-linear right-hand side (RHS) term $\mathbf{R}(\mathbf{q})$ is the spatial discretization of fluxes, source terms, and body forces. For linear subspace ROMs, we make an approximate representation of the system state via a linear combination of basis vectors,

$$\mathbf{q} \approx \widetilde{\mathbf{q}} = \overline{\mathbf{q}} + \mathbf{P} \sum_{i=1}^{K} \mathbf{v}_i \widehat{q}_i = \overline{\mathbf{q}} + \mathbf{P}\mathbf{V}\widehat{\mathbf{q}}$$

The basis $\mathbf{V} \in \mathbb{R}^{N \times K}$ is referred to as the “trial basis”, and the vector $\widehat{\mathbf{q}} \in \mathbb{R}^K$ are the generalized coordinates. The matrix $\mathbf{P}$ is simply a constant diagonal matrix which scales the model prediction. K, sometimes referred to as the “latent dimension”, is chosen such that $K \ll N$. By far the most popular means of computing the trial basis is the proper orthogonal decomposition method.

Inserting this approximation into the FOM ODE, projecting the governing equations via the “test” basis $\mathbf{W} \in \mathbb{R}^{N \times K}$, and rearranging terms arrives at

$$\frac{d\widehat{\mathbf{q}}}{dt} = \left[\mathbf{W}^T \mathbf{V}\right]^{-1} \mathbf{W}^T \mathbf{P}^{-1} \mathbf{R}\left(\widetilde{\mathbf{q}}\right)$$

This is now a K-dimensional ODE which may be evolved with any desired time integration scheme. However, for general non-linear ODEs, it is unlikely that any cost reduction is actually achieved, as the majority of the computational cost for sufficiently complex.

The following sections provide brief details on how various linear subspace projection ROMs are formulated in relation to the above ROM formulation.

2.17.1 Galerkin Projection

The linear Galerkin projection ROM [RCM04] is activated by setting `rom_method = "linear_galerkin_proj"`. As the name implies, this method applies Galerkin projection by selecting $\mathbf{W} = \mathbf{V}$. If $\mathbf{V}$ is an orthonormal basis, the ROM formulation simplifies to

$$\frac{d\widehat{\mathbf{q}}}{dt} = \mathbf{V}^T \mathbf{P}^{-1} \mathbf{R}\left(\widetilde{\mathbf{q}}\right)$$

Although Galerkin projection ROMs have been extensively studied and can be successful when applied to fairly simple fluid flow problems, they exhibit very poor accuracy and stability for practical flows.

This method requires setting the `cent_cons`, `norm_sub_cons`, and `norm_fac_cons` feature scaling profiles in `rom_params.inp`.

NOTE: Galerkin ROMs target the conservative variables, and the trial bases input via `model_files` should be trained as such. If you attempt to run the simulation with dual time-stepping (`dual_time = True`) it will terminate with an error.

2.17.2 LSPG Projection

The linear least-squares Petrov-Galerkin (LSPG) projection ROM [CBA17] is activated by setting `rom_method = "linear_lspg_proj"`. This method is so named because it is derived by solving the non-linear least-square problem

$$\widehat{\mathbf{q}} = \underset{\mathbf{a} \in \mathbb{R}^K}{argmin} || \mathbf{P}^{-1} \mathbf{r}\left(\overline{\mathbf{q}} + \mathbf{P}\mathbf{V}\mathbf{a}\right) ||_2^2$$

where $\mathbf{r}()$ is the fully-discrete residual, i.e. the set of equations arising from discretizing the FOM ODE in time. Solving this problem via Gauss-Newton, the sth subiteration is given by

$$\left(\mathbf{W}^s\right)^T \mathbf{W}^s (\widehat{\mathbf{q}}^{s+1} - \widehat{\mathbf{q}}^s) = -\left(\mathbf{W}^s\right)^T \mathbf{P}^{-1} \mathbf{r}\left(\widetilde{\mathbf{q}}^s\right)$$

where

$$\mathbf{W}^s = \mathbf{P}^{-1} \frac{\partial \mathbf{r}\left(\widetilde{\mathbf{q}}^s\right)}{\partial \widetilde{\mathbf{q}}} \mathbf{P}\mathbf{V}$$

In general, LSPG has been shown to produce more stable and accurate ROMs than Galerkin ROMs for a given number of trial modes. However, LSPG ROMs are significantly more computationally expensive (requiring the calculation of a time-variant test basis which involves the residual Jacobian). Further, LSPG ROMs deteriorate to a Galerkin projection ROM when using an explicit time integrator or as $\Delta t \rightarrow 0$. If you attempt to run an LSPG ROM with an explicit time integrator, the code will terminate with an error.

This method requires setting the `cent_cons`, `norm_sub_cons`, and `norm_fac_cons` feature scaling profiles in `rom_params.inp`.

NOTE: LSPG ROMs, as with Galerkin ROMs, target the conservative variables, and the trial bases input via `model_files` should be trained as such. If you attempt to run the simulation with dual time-stepping (`dual_time = True`) it will terminate with an error.

2.17.3 SP-LSVT Projection

The linear structure-preserving least-squares with variable transformations (SP-LSVT) projection ROM [HWDM20] is activated by setting `rom_method = "linear_splsvt_proj"`. This method leverages *Dual Time-stepping* to allow the trial bases to target an arbitrary (but complete) set of solution variables, instead of the conservative variables. This is particularly useful for combustion problems, where we would like to work with the primitive variables. To begin, the method proposes a similar representation of the primitive state as a linear combination of basis vectors

$$\mathbf{q}_p \approx \widetilde{\mathbf{q}}_p = \overline{\mathbf{q}}_p + \mathbf{H} \sum_{i=1}^{K} \mathbf{v}_{p,i} \widehat{q}_{p,i} = \overline{\mathbf{q}}_p + \mathbf{H} \mathbf{V}_p \widehat{\mathbf{q}}_p$$

where $\mathbf{V}_p$ and $\widehat{\mathbf{q}}_p$ are the trial basis and generalized coordinates for the primitive variable representation. Here, $\mathbf{H}$ is a constant diagonal scaling matrix for the primitive state. Similar to LSPG, SP-LSVT solves the non-linear least-squares problem

$$\widehat{\mathbf{q}}_p = \underset{\mathbf{a} \in \mathbb{R}^K}{argmin} || \mathbf{P}^{-1} \mathbf{r}_\tau \left(\overline{\mathbf{q}}_p + \mathbf{H} \mathbf{V}_p \mathbf{a} \right) ||_2^2$$

where $\mathbf{r}_\tau()$ is the fully-discrete *dual-time* residual. Solving this problem via Gauss-Newton, the sth subiteration is given by

$$\left(\mathbf{W}^s\right)^T \mathbf{W}^s (\widehat{\mathbf{q}}_p^{s+1} - \widehat{\mathbf{q}}_p^s) = - \left(\mathbf{W}^s\right)^T \mathbf{P}^{-1} \mathbf{r}_\tau \left(\widetilde{\mathbf{q}}_p^s \right)$$

where

$$\mathbf{W}^s = \mathbf{P}^{-1} \frac{\partial \mathbf{r}_\tau \left(\widetilde{\mathbf{q}}_p^s \right)}{\partial \widetilde{\mathbf{q}}_p} \mathbf{H} \mathbf{V}_p$$

SP-LSVT is quite similar to LSPG, but has shown exceptional accuracy and stability improvements over LSPG for combustion problems.

This method requires setting the `cent_prim`, `norm_sub_prim`, `norm_fac_prim`, and `norm_fac_cons` feature scaling profiles in `rom_params.inp`.

NOTE: SP-LSVT ROMs target the primitive variables, and the trial bases input via `model_files` should be trained as such. If you attempt to run the simulation without dual time-stepping (`dual_time = False`) it will terminate with an error.

2.18 Non-linear Subspace Projection ROMs

Over the past decade, it has become increasingly clear that linear subspace ROMs, i.e. those that represent the solution as a linear combination of trial basis vectors, are severely lacking when applied to practical fluid flow problems. Their difficulty in reconstructing sharp gradients and their inability to generalize well beyond the training dataset call into question whether they can be a useful tool for parametric or future-state prediction. This idea is synthesized by the concept of the Kolmogorov n-width [Pin85],

$$d_n(\mathcal{A}, \mathcal{X}) \triangleq \inf_{\mathcal{X}_n} \sup_{x \in \mathcal{A}} \inf_{y \in \mathcal{X}_n} ||x - y||_{\mathcal{X}}$$

which measures how well a subset $\mathcal{A}$ of a space $\mathcal{X}$ can be represented by an n-dimensional subspace $\mathcal{X}_n$. Those subsets for which an increase in n does not improve the representation much are said to have a “slowly-decaying” n-width. The solution of advection-dominated flows, which characterize most practical engineering systems, have a slowly-decaying n-width, and as such a linear representation of the solution may be quite poor.

Non-linear representations of the solution, and the ROM methods which arise from them, seek to overcome this problem. The solution approximation can be recast in a more general form as

$$\mathbf{q} \approx \widetilde{\mathbf{q}} = \overline{\mathbf{q}} + \mathbf{P} \mathbf{g} \left(\widehat{\mathbf{q}} \right)$$

where $\mathbf{g} : \mathbb{R}^K \rightarrow \mathbb{R}^N$ is some non-linear mapping from the low-dimensional state to the physical full-dimensional state. In theory, the non-linear solution manifold to which the decoder maps can more accurately represent the governing ODE solution manifold.

A particularly attractive option for developing this non-linear mapping is from autoencoders, an unsupervised learning neural network architecture. This class of neural networks attempts to learn the identity mapping by ingesting full-dimensional state data, "encoding" this to a low-dimensional state (the "code"), and then attempting to "decode" this back to the original full-dimensional state data. After the network is trained, the "decoder" half of the network is used as the non-linear mapping $\mathbf{g}$ in the ROM.

This approach has seen exceptional success for fairly simple advection-dominated problems, but is still in its infancy and has yet to be tested for any practical problems. However, it is not without its drawbacks. The cost of evaluating the neural network decoder (and its Jacobian, as will be seen later) greatly exceeds the cost of computing the linear "decoding" $\mathbf{V}\widehat{\mathbf{q}}$. The decoder predictions are also prone to noisy predictions even in regions of smooth flow. Although work is being done in developing graph neural networks, the traditional convolutional autoencoders can only be applied to solutions defined on Cartesian meshes. Further, the neural network is a black box model with no true sense of optimality besides "low" training and validation error.

The implementation of these non-linear autoencoder ROMs is dependent on the software library used to train the neural network, e.g. TensorFlow-Keras [AAB+15, C+15] or PyTorch. Some details on how these neural networks should be formatted and input to **PERFORM** are given in *TensorFlow-Keras Autoencoders*.

2.18.1 Manifold Galerkin Projection

The non-linear autoencoder manifold Galerkin projection ROM [LC20] with TensorFlow-Keras neural networks is activated by setting `rom_method = "autoencoder_galerkin_proj_tfkeras"`. After inserting the approximate state into the FOM ODE, this method leverages the chain rule and rearranges terms to arrive at

$$\frac{d\widehat{\mathbf{q}}}{dt} = \mathbf{J}_d^+ \left(\widehat{\mathbf{q}}\right) \mathbf{P}^{-1} \mathbf{R}(\widetilde{\mathbf{q}})$$

where $\mathbf{J}_d \triangleq \partial \mathbf{g} / \partial \widehat{\mathbf{q}} : \mathbb{R}^K \rightarrow \mathbb{R}^{N \times K}$ is the Jacobian of the decoder.

This method has been shown to greatly outperform linear Galerkin projection ROMs for simple advection-dominated flow problems, immensely improving shock resolution and parametric prediction at extremely low K. Unsurprisingly though, the computational cost of this method is much, much greater than its linear subspace counterpart.

This method requires setting the `cent_cons`, `norm_sub_cons`, and `norm_fac_cons` feature scaling profiles in `rom_params.inp`.

NOTE: Manifold Galerkin ROMs target the conservative variables, and the encoders/decoders input via `encoder_files`/`model_files`, respectively, should be trained as such. If you attempt to run the simulation with dual time-stepping (`dual_time = True`) it will terminate with an error.

Encoder Jacobian Form

Due to the exceptional cost of this method, an approximate method has been proposed to at least circumvent the cost of computing the pseudo-inverse of the decoder Jacobian. Under some generous assumptions of negligible error in the autoencoding procedure, we can approximate

$$\mathbf{J}_d^+ \left(\widehat{\mathbf{q}}\right) \approx \mathbf{J}_e \left(\widetilde{\mathbf{q}}\right)$$

where $\mathbf{J}_e \triangleq \partial \mathbf{h} / \partial \widetilde{\mathbf{q}} : \mathbb{R}^N \rightarrow \mathbb{R}^{K \times N}$ is the Jacobian of the *encoder* half of the autoencoder, $\mathbf{h}(\widetilde{\mathbf{q}})$. Unfortunately, this method is only applicable to Galerkin projection ROMs using *explicit* time integrators, as implicit time integration requires both the decoder Jacobian and its pseudo-inverse, eliminating the usefulness of this substitution. This method has yet to be demonstrated successfully on practical problems, but has had some success for the parametrized 1D Burgers' equation.

The encoder Jacobian form for manifold Galerkin ROMs with explicit time integrators is activated by setting `encoder_jacob = True` in `rom_params.inp`. Of course, the encoders must be provided via `encoder_files`.

2.18.2 Manifold LSPG Projection

The non-linear autoencoder manifold least-squares Petrov-Galerkin (LSPG) projection ROM [LC20] with TensorFlow-Keras neural networks is activated by setting `rom_method = "autoencoder_lspg_proj_tfkeras"`. The method follows the same procedure as the linear equivalent, but the resulting test basis takes the form

$$\mathbf{W}^s = \mathbf{P}^{-1} \frac{\partial \mathbf{r}\left(\widetilde{\mathbf{q}}^s\right)}{\partial \widetilde{\mathbf{q}}} \mathbf{P} \mathbf{J}_d(\widehat{\mathbf{q}}^s)$$

Some results indicate that manifold LSPG ROMs are more accurate than manifold Galerkin ROMs for a given number of trial modes. However, as with the linear ROMs, manifold LSPG is significantly more computationally expensive and still deteriorates to manifold Galerkin projection when using an explicit time integrator or as $\Delta t \to 0$. If you attempt to run an LSPG ROM with an explicit time integrator, the code will terminate with an error.

This method requires setting the `cent_cons`, `norm_sub_cons`, and `norm_fac_cons` feature scaling profiles in `rom_params.inp`.

NOTE: Manifold LSPG ROMs, as with manifold Galerkin ROMs, target the conservative variables, and the encoders/decoders input via `encoder_files/model_files`, respectively, should be trained as such. If you attempt to run the simulation with dual time-stepping (`dual_time = True`) it will terminate with an error.

2.18.3 SP-LSVT Projection

The non-linear autoencoder manifold structure-preserving least-squares with variable transformations (SP-LSVT) projection ROM with TensorFlow-Keras neural networks is activated by setting `rom_method = "autoencoder_splsvt_proj_tfkeras"`. As with its linear counterpart, the manifold SP-LSVT begins by providing a representation of the *primitive* state

$$\mathbf{q}_p \approx \widetilde{\mathbf{q}}_p = \overline{\mathbf{q}}_p + \mathbf{H} \mathbf{g}_p\left(\widehat{\mathbf{q}}_p\right)$$

Again following the same dual-time residual minimization procedure arrives at a similar test basis of the form

$$\mathbf{W}^s = \mathbf{P}^{-1} \frac{\partial \mathbf{r}_\tau\left(\widetilde{\mathbf{q}}_p^s\right)}{\partial \widetilde{\mathbf{q}}_p} \mathbf{H} \mathbf{J}_{d,p}\left(\widehat{\mathbf{q}}_p^s\right)$$

Again, although manifold SP-LSVT is quite similar to manifold LSPG, early results indicate that it is much more accurate and stable than manifold LSPG for combustion problems.

This method requires setting the `cent_prim`, `norm_sub_prim`, `norm_fac_prim`, and `norm_fac_cons` feature scaling profiles in `rom_params.inp`.

NOTE: Manifold SP-LSVT ROMs target the primitive variables, and the encoders/decoders input via `encoder_files/model_files`, respectively, should be trained as such. If you attempt to run the simulation without dual time-stepping (`dual_time = False`) it will terminate with an error.

2.19 License

```
MIT License

Copyright (c) 2020 Christopher R. Wentland

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

2.20 References

BIBLIOGRAPHY

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: large-scale machine learning on heterogeneous systems. 2015. Software available from tensorflow.org. URL: https://www.tensorflow.org/.

[BJ89] Timothy Barth and Dennis Jespersen. The design and application of upwind schemes on unstructured meshes. In *27th Aerospace Sciences Meeting*. American Institute of Aeronautics and Astronautics, 1989. doi:10.2514/6.1989-366.

[CBA17] Kevin Carlberg, Matthew Barone, and Habir Antil. Galerkin v. least-squares Petrov–Galerkin projection in nonlinear model reduction. *Journal of Computational Physics*, 330:693–734, 2017. doi:10.1016/j.jcp.2016.10.033.

[C+15] François Chollet and others. Keras. 2015. URL: https://keras.io.

[DXSM04] Li Ding, Guoping Xia, Venkateswaran Sankaran, and Charles L. Merkle. Computational framework for complex fluid physics applications. *Third International Conference on Computational Fluid Dynamics*, pages 619–624, 2004. doi:10.1007/3-540-31801-1_89.

[GY08] Irvin Glassman and Richard A Yetter. *Combustion*. Academic Press, 2008.

[Har12] Matthew E. Harvazinski. *Modeling self-excited combustion instabilities using a combination of two- and three-dimensional simulations*. PhD thesis, Purdue University, 2012.

[HWDM20] Cheng Huang, Christopher R. Wentland, Karthik Duraisamy, and Charles Merkle. Model reduction for multi-scale transport problems using structure-preserving least-squares projections with variable transformation. *arXiv:2011.02072v3*, 2020.

[LC20] Kookjin Lee and Kevin T. Carlberg. Model reduction of dynamical systems on nonlinear manifolds using deep convolutional autoencoders. *Journal of Computational Physics*, 2020. doi:10.1016/j.jcp.2019.108973.

[Pin85] Allan Pinkus. *n-Widths in Approximation Theory*. Springer, 1985.

[Roe81] Philip L. Roe. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics*, 43(2):357–372, 1981. doi:10.1016/0021-9991(81)90128-5.

[RCM04] Clarence W. Rowley, Tim Colonius, and Richard M. Murray. Model reduction for compressible flows using pod and galerkin projection. *Physica D*, 189:115–129, 2004. doi:10.1016/j.physd.2003.03.001.

[Ven93] V. Venkatakrishnan. On the accuracy of limiters and convergence to steady state solutions. In *31st Aerospace Sciences Meeting*. American Institute of Aeronautics and Astronautics, 1993. doi:10.2514/6.1993-880.

[VM95] Sankaran Venkateswaran and Charles L. Merkle. Dual time-stepping and preconditioning for unsteady computations. *33rd Aerospace Sciences Meeting and Exhibit*, 1995. doi:10.2514/6.1995-78.