LA-UR-16-23260
Approved for public release; distribution is unlimited.

Title: ExactPack Documentation

Author(s): Singleton, Robert Jr.
Israel, Daniel M.
Doebling, Scott William
Woods, Charles Nathan
Kaul, Ann
Walter, John William Jr
Rogers, Michael Lloyd

Intended for: Documentation of the software product ExactPack.

Issued: 2017-08-25 (rev.1)

ExactPack Documentation

Release 1.6.0

Verification Team, LA-UR-16-23260

Aug 18, 2017

CONTENTS

CHAPTER

ONE

INTRODUCTION

For code verification, one compares the code output against known exact solutions. There are many standard test problems used in this capacity, such as the Noh and Sedov problems. ExactPack is a utility that integrates many of these exact solution codes into a common API (application program interface), and can be used as a stand-alone code or as a python package. ExactPack consists of python driver scripts that access a library of exact solutions written in Fortran or Python. The spatial profiles of the relevant physical quantities, such as the density, fluid velocity, sound speed, or internal energy, are returned at a time specified by the user. The solution profiles can be viewed and examined by a command line interface or a graphical user interface, and a number of analysis tools and unit tests are also provided. We have documented the physics of each problem in the solution library, and provided complete documentation on how to extend the library to include additional exact solutions. ExactPack's code architecture makes it easy to extend the solution-code library to include additional exact solutions in a robust, reliable, and maintainable manner.

1.1 How To Get Started

To get up and running with ExactPack quickly, you should start by reading the *Quickstart Guide* and trying out some of the examples. There are examples available for all of the solvers, as well as for the convergence analysis tools. Examples are located in the distribution directory under `exactpack/examples`, and can be run as standalone Python scripts, or as Python modules:

```
python -m exactpack.<example name>
```

If you want to use ExactPack for code verification or physics exploration, the *User's Guide* gives a complete explanation of how to use ExactPack. You will then want to look up the details of the specific solver you are working with in *Solvers*.

For those who want to add new solvers to ExactPack, read the *Developer's Guide*, with particular attention to *Coding Style* and *Adding a New Solver*. The *Reference Guide* is mostly intended for those wishing a more thorough understanding of ExactPack's internals, such as developers adding additional functionality for the code verification tools, GUI, etc.

Finally, the *Testing* section documents what internal self-tests ExactPack uses to verify the solutions it provides.

1.2 How To Get Help

The first place to look is in this manual, particularly the API documentation, as well as looking at the source code itself. If you need additional assistance, or wish to report a bug, please e-mail exactpack-support@lanl.gov. This software is provided free of charge, and we make no guarantee of support, but will be happy to look at your issue if we can.

CHAPTER

TWO

QUICKSTART GUIDE

There are three ways to use ExactPack: in Python as a module (*Using ExactPack as a Python Library*), using the GUI (*The ExactPack Graphical User Interface*), or using the command line utility (*Using ExactPack from the Command Line*). The first is intended as the primary interface, particularly for use in building more complete code verification tools. The other two are easy ways to quickly visualize exact solutions, or interactively explore data sets.

2.1 Installation

Installing ExactPack is simple. Just unpack the tar file and run the setup script:

```
tar xzf ExactPack-1.0.tar.gz
cd ExactPack-1.0
python setup.py install
```

This will install to the system Python library directories. If you do not have write permission for these directories, you can usually install to a user specific location by using the `--user` flag. All the other standard setup options should also be available (run `python setup.py --help` for more information).

The setup script also installs the GUI and command-line scripts described below.

2.2 The ExactPack Graphical User Interface

Warning: As of this writing, the GUI is undergoing active, early-stage development, so the interface may have major changes in subsequent releases.

ExactPack has a simple GUI, which can be started by running from the command line:

```
epgui
```

The interface is relatively straightforward. Choose a problem from the menu, input values for any parameters for which the defaults are not satisfactory, and click *Plot*. Multiple plots can be combined, or you can clear the canvas before plotting using *Clear and Plot*. *Save* writes a CSV data file containing the data from the last item plotted. To save the plot itself, use the controls in the plot window.

2.3 Using ExactPack from the Command Line

Warning: As of this writing, the command-line interface is undergoing active, early-stage development, so the interface may have major changes in subsequent releases. Users are advised not to develop any scripts or code that depend on the current behavior of the command-line tool.

Complete and current help information can be found by running:

```
exactpack --help
```

Also, running:

```
exactpack --doc
```

will open this manual (in either HTML or PDF format) in an appropriate viewer.

2.4 Using ExactPack as a Python Library

ExactPack is designed to be used to be used as a Python library in postprocessing scripts for code verification. It provides a number of tools to make it easy to compare exact solutions against the output of other codes, and for convergence analysis.

By default, importing ExactPack does not load in any specific solutions. Instead you must load in the particular problem you want. So, for example, you can access the default solver for the Noh problem by:

```
>>> from exactpack.solvers.noh import Noh
```

The following is an example of a script that computes a spherical Noh solution, with a specific-heat ratio of 1.4, and then plots it:

```
import numpy
from exactpack.solvers.noh import Noh

solver = Noh(gamma=1.4, geometry=3)
soln = solver(linspace(0, 1.0), 0.3)
soln.plot_all()
```

To get a complete list of the available solvers use *exactpack.utils.discover_solvers()*. Note that there may be several solvers for a particular problem. For example, by default *exactpack.solvers.noh* loads a pure Python implementation, *exactpack.solvers.noh.noh1*. If you want the Fortran implementation by Frank Timmes, you need to explicitly import *exactpack.solvers.noh.timmes*.

For documentation on specific solvers, including information on what parameters they accept and what solution methods are employed, see the API documentation (`exactpack`). For information about utility functions and analysis tools, see the APIs for those specific packages (*exactpack.utils* and `exactpack.analysis`).

CHAPTER

THREE

USER'S GUIDE

The User's Guide is intended to explain how to use ExactPack to build Python scripts for verification. It describes the basic layout of the libraries and the *API*.

3.1 Available Solvers

Importing `exactpack` loads some base code and utility functions, but no actual solvers. Individual solvers should be imported as needed. A solver sub-package is given by the name of the problem in lower case, preceded by its path. So the default Noh problem solver is imported using[1]:

```
import exactpack.solvers.noh
```

If there are multiple solvers available for a given problem, they will be in sub-sub-packages. The base package will load a default choice of the solver, which is the solver recommended for general use by the maintainers of ExactPack.

For example, the default Noh solver is really *exactpack.solvers.noh.noh1*, and this is what you get if you import *exactpack.solvers.noh*. If you want an interface to the Fortran implementation of Frank Timmes, then you can explicitly import *exactpack.solvers.noh.timmes*.

The solver itself is a Python class. Multiple solver variants may be available within a given package, but these will all use the same underlying solver. For example, the general Noh solver `exactpack.solvers.noh.Noh` can be used for the Noh problem in 1-, 2-, or 3-dimensions, or the wrapper classes `exactpack.solvers.noh.PlanarNoh`, `exactpack.solvers.noh.CylindricalNoh`, and `exactpack.solvers.noh.SphericalNoh` can be used.

For readability, it is often preferable to use a `from ... import ...` construct to load just the desired solvers, for example:

```
from exactpack.solvers.noh import SphericalNoh
```

To obtain a list of all available solvers, use the *exactpack.utils.discover_solvers()* function. The example scripts are also a good place to learn about importing solvers.

3.2 Basic Usage

3.2.1 The Solver Class

All solvers present a uniform *API*. The API needs to be sufficiently flexible to accommodate solvers for different problems, which may differ in input parameters and output variables, but uniform enough to make it simple to create

[1] Due to its simplicity, the Noh solver will be used as the example and reference implementation throughout this document.

re-usable scripts for analysis and plotting. This is accomplished by having all solvers derive from the *ExactSolver* base class.

The *ExactSolver* interface is quite simple. The constructor takes a series of keyword parameters which define the parameters and initial conditions for the problem, and returns a solver object. The attribute *ExactSolver.parameters* is a dictionary with keywords which are the available parameters, and values which are help strings for each parameter. If there are uninitialized parameters, or the constructor is passed parameters that it does not know, it will raise an exception. Relying on default parameters will generate a warning, to make sure the user knows exactly what parameters are being used.

For example, the *exactpack.solvers.noh.noh1.Noh* takes four parameters, the geometry, the gas constant, and the reference velocity and density. To instantiate a solver for the spherical Noh problem, with the value $\gamma = 5/3$ as in the original paper by Noh[#]_, we can use the following code

```
from exactpack.solvers.noh import Noh

solver = Noh(geometry=3, gamma=5.0/3.0)
```

The solver is a function-like object that can be called with two arguments. The first is a sequence listing the spatial points at which the solution is returned, and the second is the time. The points list can be a `numpy` array or a Python sequence (list or tuple) of points. Depending on the solver, each point may be 1-, 2-, or 3-dimensional, and may be in rectangular, cylindrical, or spherical coordinates. For 1-dimensional problems, the points are given by a rank-1 `numpy` array, or a list of scalar values. For 2- or 3-dimensional problems, the points are given by a `numpy` array of shape `(N,2)` or `(N,3)`, or by a list of 2- or 3-tuples. Check the documentation for a particular solver for details.

Attributes of a solver, such as `geometry` and `gamma`, should be treated as read-only once the solver has been instantiated.

Using the solver object defined above, the following call will return the solution at time $t = 0.6$ in the interval $x \in [0, 1]$:

```
from numpy import linspace

solution = solver(linspace(0, 1), 0.6)
```

(The `numpy.linspace()` function is a convenient way to generate an evenly spaced set of points. It has three arguments: the start point, the stop point, and an optional number of points to use.)

A Note on Dimensions

In general, the equations which are solved are dimensionally consistent, which means that as long as all the inputs are provided in consistent units (or consistently non-dimensionalized), the output will also be consistent. In any case, where a solver requires input in particular units, this will be specified in the solver documentation; if no units are specified, any consistent set may be employed.

3.2.2 The Solution Class

When a solver object is called to generate a solution, the return value is an *ExactSolution* instance. This is a structured array in which the fields (columns) can be accessed by variable name and the spatial points by indices. The first field is always the solution locations passed as the first argument to the solver. So, for example, using the solution computed in the previous section, one can do the following:

```
>>> solution[1]
(0.02040816326530612, 64.0, 21.333333333333332, 0.5, 0.0)

>>> solution.density[:8]
```

```
array([ 64.,  64.,  64.,  64.,  64.,  64.,  64.,  64.])

>>> solution['specific_internal_energy'][0]
0.5
```

The available variable names are standardized, and are listed in *Standardized Variable Names*. (For more details, on possible syntax, see the documentation for Numpy Structured arrays, of which *ExactSolution* is a subclass.)

Plotting

The methods *ExactSolution.plot()* and *ExactSolution.plot_all()* are convenient wrappers for matplotlib plotting. The syntax should be familiar to anyone familiar with that package.

To get a quick, auto-scaled, plot of all solution variables:

```
solution.plot_all()
```

Note: *ExactSolution.plot()* and *ExactSolution.plot_all()* do not issue a `matplotlib.pyplot.show()` internally, so it may be necessary to do this manually, depending on which backend you use. If you want to work with plots interactively from the command-line running IPython with the `matplotlib.pylab` environment:

```
ipython --pylab
```

will automatically take care of updating plots, as well as setting up your environment to include the most common Numpy and matplotlib functions.

For more control over your plots, the *ExactSolution.plot()* function takes one required argument, the name of the variable to plot, and also accepts all the keyword options understood by the `matplotlib.pyplot.plot()` function. So, for a solution object *solution*:

```
solution.plot('density', 'r--', marker='o')
```

will plot the density with a red dashed line and circles.

In addition to all the keywords supported by `matplotlib.pyplot.plot()`, *ExactSolution.plot()* understands one additional keyword, *scale*, which is an arbitrary scaling factor to apply to the data before plotting. For `scale='auto'`, *ExactSolution.plot()* will pick a scaling that results in a plotted value of order one.

If no *label* keyword is specified, *ExactSolution.plot()* will generate a label.

Other Functions

To dump the solution to a CSV file use *ExactSolution.dump()*:

```
solution.dump("filename.csv")
```

3.2.3 Jump Conditions

The *ExactSolution* object can optionally report the exact locations and left and right states for points at which the solution is discontinuous. These values are stored in the attribute *ExactSolution.jumps*. If this attribute is set to `None`, this means that the solver does not report discontinuities. Otherwise this will be set to a list, each element of

which will be of type *JumpCondition*. An empty list means that the solution is continuous (i.e., the list of jumps has no elements).

A *JumpCondition* has a location, and left and right values for each variable:

```
>>> solution.jumps
[JumpCondition(location=0.2,velocity=(0,-1),density=(64.0,16.0),
  specific_internal_energy=(0.5,0),pressure=(21.3333333333,0))]
>>> solution.jumps[0].location
0.19999999999999998
>>> solution.jumps[0].density
Jump(left=64.0, right=16.0)
>>> solution.jumps[0].density.left
64.0
>>> solution.jumps[0].density.right
16.0
```

3.3 Convergence Analysis

ExactPack is primarily designed as a tool for code verification. A comprehensive introduction to code verification is beyond the scope of this guide. In brief, the idea is to compare the output of a numerical solution produced by a PDE solver to an exact analytic or semi-analytic solution, such as those provided by ExactPack. The magnitude of the error in the solution is computed for a seriers of grid or timestep refinements, and one confirms the error converges to zero. Ideally, the rate of convergence is also compared to the rate predicted by theory, if one is available.

Although ExactPack can provide solutions for any user script for analyzing convergence, there are also some built-in tools for convergence analysis in `exactpack.analysis`. The basic object for doing verification analysis is `Study`. To perform a code verification study, instantiate a study object by passing it a list of data files, an ExactPack solver, and a data reader:

```
from exactpack.analysis import Study
from exactpack.analysis.readers import TextReader
from exactpack.solvers.riemann import Sod

study = Study(['coarse.dat', 'medium.dat', 'fine.dat'],
              reference=Sod(),
              reader=TextReader)
```

The user can then perform and display a convergence fit using one of several routines, for example:

```
from exactpack.analysis import FitConvergenceRate

fitstudy = FitConvergenceRate(study)

print fitstudy.report()

fitstudy.plot('pressure')
plt.show()
```

The above works for three convergence fit methods: `FitConvergenceRate` (uses scipy optimization fit), `RoacheConvergenceRate`, (uses Richardson-extrapolation based method as described in Roache's book), and `RegressionConvergenceRate` (uses linear regression).

CHAPTER

FOUR

DEVELOPER'S GUIDE

The Developer's Guide is intended to explain the internal architecture of ExactPack and how to add new solvers. It assumes you have already read and understood the *User's Guide*. In the description that follows, extensive reference will be made to the implementation of the solver `exactpack.solvers.noh`. This should be considered the reference implementation, and a good template for new solvers. Potential developers are advised to carefully look through the source code for `exactpack.solvers.noh` in conjunction with reading this chapter, particularly `exactpack.solvers.noh.noh1.Noh` and `exactpack.solvers.noh.timmes.Noh`. (In the HTML version of this manual, there is a hyperlink to the source code from the documentation for each class and function.)

4.1 Coding Style

4.1.1 General Guidelines

The following are some recommended guidelines for project coding style.

- Be pythonic. When in doubt about the best way to implement something, ask whether this is the most natural way to do it in Python. Follow **PEP 20**, also known as *The Zen of Python*.
- When in doubt, use Python standards for formatting code. General coding style is covered in **PEP 8**. For docstrings, see **PEP 257**, except use Sphinx compatible RST markup wherever possible, since the docstrings are used to generate this documentation. (A brief description of RST can be found in reStructuredText Primer, and Sphinx specific markup in Sphinx Markup Constructs and Sphinx Domains.)
- Unless absolutely necessary, use only libraries in the Anaconda Python Distribution. Installing packages with complex dependencies can be a pain, and we make it easier by at least assuring that ExactPack will always install under Anaconda.
- Good documentation takes time. Do it anyway, the time will be amply repaid in the eventual service life of the code. Others cannot read your mind!

4.1.2 Requirements for New Solvers

Before releasing a new solver, please make sure the following requirements have been met.

- The solver supports the ExactPack API as described in this chapter.
- The solver is fully documented including:
  - A complete description of the physics and geometry of the problem and the equations being solved.
  - Either a description of the solution method, or a pointer to an archived publication with such a description.
  - A description of the all input and output parameters, including the choices of any default values. (See the notes in *Documenting the Parameters* for the correct format to do this.)

- Unittests for the solver are provided.
  - Unittests should be documented to indicate what they do and why these tests are selected.
  - At a minimum, unittests should demonstrate that the solver reproduces the correct results in limiting cases and well established, previously published, data.
  - Additional tests, such as solution verification (checking that the output of the solver converges), are always useful. The more tests, the more confidence we can have in the output.
- A simple example script is also provided.

4.2 A Tour of the Package Source

The main ExactPack package consists of several sub-packages, some of which provide solvers, while others contain supporting code. The layout of the package directory is as follows (not every file or directory is shown):

```
ExactPack/
    doc/
    exactpack/
        __init__.py
        analysis/
        base.py
        contrib/
        solvers/
            noh/
                __init__.py
                noh1.py
                timmes.py
                _timmes.so
        tests/
            __init__.py
            test_noh.py
        utils.py
    examples/
        noh.py
    setup.py
    src/
        timmes/
            noh/
                noh.f
```

`doc/`

The Sphinx documentation (this document).

`exactpack/`

This is the main Python code for the ExactPack package. It includes several important subdirectories which are described below.

`analysis/`

The `analysis` module provides tools for *Convergence Analysis*, as well as plotting tools and data file readers.

`contrib/`

This is the location for third-party or other contributed solvers that are sufficiently developed to be redistributed as part of the standard distribution, but which are provided as-is, with no

support. This may be because the solver is not up to the requirements listed in *Coding Style*, or because support is provided by someone outside the development team.

If you add a solver, it should be initially added in the `contrib/` directory, and the documentation should include your contact information as the support point-of-contact, using the `codeauthor` directive. The ExactPack development team may decide to move your solver to the `solvers/` directory, if appropriate.

`solvers/`

The `solvers` directory contains the supported solver packages. For example, the `noh` directory in `solvers` is a package providing an exact solution code for the Noh problem. This includes two different implementations of the Noh problem, `noh1.py` is written in pure Python code using Numpy, and `timmes.py` is a wrapper for the Fortran implementation whose source code is in `src/timmes/noh/noh.f`. The dynamically loadable shared object library `_timmes.so` is not actually distributed with the source, as it is automatically compiled when the package is installed using the `setup.py` script, and it might have different names on different architectures. It is a Python importable object, but is not meant to be accessed directly, but rather, via the `timmes.py` wrapper (which is why it is named with a leading underscore).

`tests/`

ExactPack includes unit tests both for basic functionality of the package, as well as verification of the solvers. Verifying exact solution codes is a tricky issue (of the "who will watch the watchers" variety), and is discussed in *Testing*.

`examples/`

Several example files are provided as templates for how to write Python scripts that use the ExactPack package.

`setup.py`

This is the setuptools packaging script for building and installing ExactPack.

`src/`

Fortran or other non-Python source files for exact solution codes is stored here. The `src` directory is organized into sub-directories by author or contributor, and then by problem.

This description does not include every file that you will see if you browse the source tree. There will be many more package directories and corresponding source directories for the different solvers. Each problem has its own package, with each solver a separate module within that package. This allows the library to provide multiple solvers for each problem, which is useful for cases when the various solvers work in different regimes, or provide different functionality. Each module contains one or more solver classes. Although the classes in a module will all use the same solver, different classes may be used as interfaces for specific cases. For example, *exactpack.solvers.noh.noh1.Noh* is a general Noh solver useable in planar, cylindrical, or spherical geometries; whereas *exactpack.solvers.noh.noh1.SphericalNoh* is a derived convenience class with default settings for a solution in a spherical geometry.

4.3 Adding a New Solver

Let's take a closer look at the *exactpack.solvers.noh* package, and see how we would go about adding a new solver.

First, we need to create a sub-directory to hold the solver, in this case this is the `noh` directory. Next, we add an `__init__.py` file to tell Python that this new directory is a package. At the head of the `__init__.py` we put a

docstring describing the Noh problem. This docstring can be quite long, since it will be the complete documentation of the problem that will appear in the API section of this manual.

4.3.1 A Native Python Solver

Since the Noh problem has an analytic closed form solution, it is relatively simple to implement directly in Python using Numpy. An example is `noh1.py`.

The main part of the implementation is defining the class *exactpack.solvers.noh.noh1.Noh* (class names are to be capitalized by convention), which derives from the *ExactSolver* base class:

```
class Noh(ExactSolver):

    r"""Computes the solution to the general Noh problem.

    This is a base class with default values, which can be used to solve the Noh
    problem for a perfect-gas with any specific heat ratio, :math:`\gamma`.  It
    supports one-dimensional planar, two-dimensional cylindrical, and
    three-dimensional spherical shocks. The default values for the parameters
    are :math:`\rho_0=1` and :math:`u_0=1`, although the user is free to
    change these parameters. The default geometry is spherical, with
    :math:`\gamma=5/3`.

    The solver reports jumps in the
    :attr:`exactpack.base.ExactSolution.jumps` attribute of the return value.

    """

    parameters = {
        'geometry': "1=planar, 2=cylindrical, 3=spherical",
        'gamma': "specific heat ratio :math:`\gamma \equiv c_p/c_v`",
        'u0': "incident velocity (negative)",
        'rho0': "density"
        }

    geometry = 3
    gamma = 5.0 / 3.0
    u0 = -1.0
    rho0 = 1.0
```

The Python docstring is formatted using RST so that it will be rendered nicely by Sphinx as part of this document. The `r` at the beginning indicates a raw string, which means that backslashes and other control codes will not be processed by the Python interpreter. (For more information on formatting docstrings for Sphinx, see the Sphinx documentation contents.)

In order to insure a uniform API, the initialization of solver attributes is handled by *ExactSolver*, and the solver class itself should not set any parameters. Instead the class definition should set a `parameters` attribute, which is a dictionary with each key-value pair consisting of a parameter name and a corresponding help string. This allows the base class to provide uniform error checking, documentation, and other functionality. This attribute is also used by other functions to determine the valid parameters for the solver class. For example, the GUI uses this to build a dialog box that includes the appropriate parameters for a given solver class.

Providing a constructor is optional. The default constructor for *ExactSolver* takes an arbitrary list of keyword arguments, and adds them to the instance's dictionary. It also does error checking for missing or unknown parameters. In most cases, the solver class does not need to provide a constructor. If a constructor is provided, it *must* use the `super()` function to call the base class constructor, which takes care of the parameter initialization.

For the Noh problem, we want to raise an error if the `geometry` parameter is not a valid value, or the initial velocity of the gas is not moving in toward the origin. The constructor looks like:

```
    def __init__(self, **kwargs):

        super(Noh, self).__init__(**kwargs)

        if self.geometry not in [1, 2, 3]:
            raise ValueError("geometry must be 1, 2, or 3")

        if self.u0 >= 0:
            raise ValueError("Incident velocity must be negative")
```

The actual computation of the exact solution is done by the `_run` method. The `_run` method takes two arguments: a Numpy array containing the spatial points at which the solution should be computed, and a scalar time. The return value must be of type *`ExactSolution`*.

The Noh problem has a solution that can be expressed as a piecewise analytic function (e.g. *[Gehmeyr1997]*), which is easily expressed in Numpy:

```
    def _run(self, r, t):

        shock_location = abs(self.u0) * t * (self.gamma - 1) / 2

        density = np.where(r < shock_location,
                           self.rho0 * ((self.gamma + 1) / (self.gamma - 1)) ** \
                           self.geometry * np.ones(shape=r.shape),
                           self.rho0 * (1 + abs(self.u0) * t / r) ** (self.geometry -␣
↪1))
        pressure = np.where(r < shock_location,
                            (self.rho0 * self.u0 ** 2) * 4.0 ** self.geometry / 3.0 *␣
↪\
                            np.ones(shape=r.shape),
                            np.zeros(shape=r.shape))
        sie = np.where(r < shock_location,
                       (self.u0 ** 2) * (1.0 / 2.0) * np.ones(shape=r.shape),
                        np.zeros(shape=r.shape))
        velocity = np.where(r < shock_location,
                            np.zeros(shape=r.shape),
                            self.u0 * np.ones(shape=r.shape))

        return ExactSolution([r, density, pressure, sie, velocity],
                             names=['position',
                                    'density',
                                    'pressure',
                                    'sie',
                                    'velocity'],
                             jumps=[JumpCondition(shock_location,
                                    "Shock",
                                    density=(self.rho0 * ((self.gamma + 1) / \
                                    (self.gamma - 1)) ** self.geometry,
                                    self.rho0 * 4.0 ** (self.geometry - 1)),
                                    pressure=((self.rho0 * self.u0 ** 2) * \
                                              4.0 ** self.geometry / 3.0, 0),
                                    sie=( (self.u0 ** 2)* 1.0 / 2.0, 0),
                                    velocity=(0, -1))
                                    ]
        )
```

The *ExactSolution* constructor takes two required arguments. The first is an sequence of arrays, the first of which is the solution locations (this is the same as the first argument passed to _run), and the rest are solution values for each variable at the corresponding locations. (Note that a rank-2 Numpy array, with first index corresponding to the solution variable, and second index to the spatial position, will also work.) The second argument names is a sequence of strings giving the names of each of the corresponding solution variable. For variable names, the standard names found in table *Standardized Variable Names* should be used, if one is listed. A third optional argument jumps describing the discontinuities is also possible (see below).

Documenting the Parameters

The *ExactSolver* class uses some Python magic (in particular, meta-classes; for details see the source code) to simplify documenting the parameters. We want the parameters to be consistently documented in the manual, the help() function, and to have the help information available to tools such as the GUI. This is done in an automated fashion via the *ExactSolver.parameters* attribute. As described above, this attribute consists of a dict with all parameters of the solver class as keywords, and a short, one-line description as the associated values. At class creation, this help information is automatically added to the docstring.

The parameter list is also used to check the parameters used to instantiate a solver, and issue errors or warnings as necessary.

If a longer description of any of the parameters is needed, that should be incorporated into the documentation for the class. By default, any documentation of the __init__ method (if there is one) will be appended to the documentation for the class when generating the manual.

More About Jump Conditions

There is an optional third argument, jumps, which can be used to indicate the location and values of the solution at discontinuities. If this is omitted, the value of jumps defaults to None, which means that the solver does not provide information about jump conditions. Otherwise, this is a list of *JumpCondition*s, one for each discontinuity in the solution. To indicate a continuous solution use an empty list, jumps=[].

Note: jump=None indicates that the solver does not provide information regarding jumps. The empty list jump=[] means that the solution is continuous and that there are no jumps. Since analysis tools may use the jumps for comparison with computed solution, it is important distinguish between these values.

The first argument to *JumpCondition* is the spatial location of the discontinuity. The second is a short string describing the type of discontinuity. The remaining arguments are keyword arguments, the keyword giving the variable name (which should be the same as used in the names argument of the *ExactSolution* constructor), and the values at the left and right states. These are converted to *Jump* objects before being stored. In the Noh example, above, the values are given as 2-tuples, which are interpreted as (left, right) states.

Setting the Default Solver

For the *exactpack.solvers.noh* package, there are two available implementations of the solver. The __init__.py file loads the default solver, in this case:

```
from noh1 import Noh
```

Even in cases with a single solver, it is preferable to put it in a separate file, and import it in __init__.py. This is consistent with the Python convention of avoiding code in __init__.py and also keeps the package names from changing if an additional solver is added in the future.

Convenience Wrappers

In some cases, it may be useful to have multiple interface classes for the same solver. For example, the general Noh solver, *Noh* can be used to set up the Noh problem in planar, cylindrical, or spherical geometries, but it is convenient to have specialized classes for each of these three cases.

We can derive a *SphericalNoh* class from the general solver described above:

```
class SphericalNoh(Noh):
    """The standard spherical Noh problem.

    The spherical Noh problem as defined in [Noh1987]_, with a default
    value of :math:`\gamma=5/3`.
    """

    parameters = {'gamma': Noh.parameters['gamma']}
    geometry = 3
    gamma = 5.0 / 3.0
```

In this case, the class definition does not include any methods; they are all defined in the parent class. The `geometry` attribute defaults to 3, corresponding to spherical geometry. There is no `geometry` attribute in the `parameters` list, since we don't want users to change it: `geometry` will neither be a legal parameter to the constructor, nor will it show up as a parameter in the GUI.

4.3.2 A Fortran Solver

Often we already have a solver in another language, usually some variant of Fortran, and we wish to incorporate it into the ExactPack framework. Converting the solver to pure Python may not be advisable: there may be to high a risk of introducing new bugs, it may run too slowly, or the time involved may just be too great to justify. In this case we can use the F2PY package to wrap the existing Fortran code for easy access from Python.

The first step is to decorate the original Fortran code with F2PY directives. These are specially formatted Fortran comments that are used by the F2PY package to determine the calling interface.

For our Noh problem example, we started with the code from Frank Timmes verification website. Some modifications to the source code are needed, such as removing the main driver (this will be provided on the Python side) and unused support routines, removing the input and output statements (this will also by handled in Python), and moving the loop over spatial points into the main driver routine (for efficiency). Once this is done, the driver routine is decorated with F2PY directives:

```
      subroutine noh_1d(time,xpos,nstep,
     1                  rho1,u1,gamma,xgeom,
     2                  den,ener,pres,vel,jumps)
      implicit none
      save

cf2py intent(out) :: den, ener, pres, vel, jumps
cf2py intent(hide) :: nstep
cf2py double precision :: time, rho1, u1, gamma, xgeom
cf2py double precision :: xpos(nstep)
cf2py double precision :: den(nstep), ener(nstep), pres(nstep), vel(nstep)
cf2py double precision :: jumps(9)
```

The `cf2py` prefix is a Fortran comment and identifies an F2PY directive. `intent(out)` is used to identify which variable will be passed to Python as return values. `intent(hide)` is used to hide an argument so it doesn't appear explicitly in the Python interface. In this case the variable in question, `nstep`, does not need to be explicitly passed

from Python because it can be determined by the wrapper based on the length of `xpos`. The type declarations are used by the Python wrapper to do type checking and conversion on the arguments.

With the addition of these directives, this Fortran source can now be compiled into a shared library, which is loadable by Python. The signature for this function is:

`exactpack.solvers.noh._timmes.`**`noh_1d`**(*time*, *xpos*, *rho1*, *u1*, *gamma*, *xgeom*) -> (*den*, *ener*, *pres*, *vel*, *jumps*)

It takes six positional or keyword arguments, which map to the arguments of the Fortran function with the exception of those tagged either `intent(out)` or the `intent(hide)`. It returns a tuple of Numpy arrays, which are the Fortran arrays with `intent(out)`, in the order given. F2PY takes care of creating arrays that will persist after the Fortran subroutine exits.

Compilation can be done from the command line using the `f2py` utility, but we automate it through ExactPack's `setup.py` file. This allows us to make a source distribution that compiles automatically when installing using Python's packaging tools. This is done by adding a `numpy.distutils.core.Extension` object to the `ext_modules` list of the `setup` class:

```
    ext_modules = [ Extension(name = 'exactpack.solvers.noh._timmes',
                              sources = ['src/timmes/noh/noh.f'] ),
```

where `name` is the name that will be used for the compiled package. On most systems, for this example, building ExactPack will generate a file `exactpack/noh/_timmmes.so`, which is the loadable library. We choose a name with a leading underscore to indicate that this module is not meant to be called directly by the user (as we shall see). `sources` is a list of source files to compile. By convention, ExactPack groups source files in problem specific directories, organized first by source code author, then by problem.

Sometimes a source file includes multiple routines, only some of which need to be exposed in the Python interface. In that case, an `f2py_options` argument should be passed to `numpy.distutils.core.Extension`'s constructor, with an `only:` flag. The syntax is:

```
f2py_options = ['only:'] + [ 'noh_1d' ] + [':']
```

In our Noh example, this is not necessary, since the source file contains only one routine.

At this point we are only half finished. We now have a Fortran library that can be accessed by importing:

```
import exactpack.solvers.noh._timmes
```

but the interface is hard to use and does not conform to the ExactPack API. We next must create a solver class to wrap the Fortran library. This we put in the file `exactpack/noh/timmes.py`. The class header is almost identical to the one for the Python solver:

```
class Noh(ExactSolver):
    r"""Computes the solution to the general Noh problem.

    The functionality of this class is completely superseded by
    :class:`exactpack.solvers.noh1.Noh`, the only difference being
    that this class is a wrapper to a Fortran library.  This version
    is provided primarily as a template example to developers to
    demonstrate how to add an exact solution that is computed by an
    external Fortran code. The default geometry is spherical, with
    :math:`\gamma=5/3`. The parameters values for the density and
    gas velocity are are :math:`\rho_0=1` and :math:`u_0=1`.
    """

    parameters = {
        'geometry': "1=planar, 2=cylindrical, 3=spherical",
```

```
        'gamma': "specific heat ratio :math:`\gamma \equiv c_p/c_v`",
        }

    geometry = 3
    gamma = 5.0/3.0
```

The constructor in this case also looks identical:

```
    def __init__(self, **kwargs):

        if 'geometry' in kwargs and not kwargs['geometry'] in [1, 2, 3]:
            raise ValueError("geometry must be 1, 2, or 3")

        super(Noh, self).__init__(**kwargs)
```

For more complex problems, we might want to call some Fortran code immediately in the constructor to compute intermediate values that will not change between calls to the `_run` method.

The `_run` method now is simply a wrapper to the Fortran function:

```
    def _run(self, r, t):

        density, sie, pressure, velocity, jumps = _timmes.noh_1d(
            t, r, 1.0, -1.0, self.gamma, self.geometry)

        return ExactSolution([r, density, sie, pressure, velocity],
                             names=['position',
                                    'density',
                                    'sie',
                                    'pressure',
                                    'velocity'],
                             jumps=[JumpCondition(jumps[0],
                                    "Shock",
                                    density=(jumps[1], jumps[2]),
                                    sie=(jumps[3], jumps[4]),
                                    pressure=(jumps[5], jumps[6]),
                                    velocity=(jumps[7], jumps[8]))
                                    ]
        )
```

Note that we do not need to create the arrays `density`, etc., since that is taken care of by the F2PY wrapper. They are then passed to the *`exactpack.base.ExactSolution`* constructor. The original Fortran source was modified to return an additional array, `jumps`, into which is packed the location of the shock, and pre- and post-shock states. These could equally well have been returned as nine separate scalars. The result would have been a more complex call signature, but would have eliminated the need to make sure the array is packed and unpacked consistently on the Fortran and Python sides, respectively.

For more information about F2PY and using it to interface with Numpy arrays, see the F2PY User's Guide and Reference Manual and Using Python as Glue.

4.3.3 Unit Tests

Strictly speaking, real code verification for exact solution codes is difficult, since there is often nothing to compare the numerical soluton with. On the other hand, if users are to have confidence in the results, there should be some testing. Depending on the problem, this could consist of solution verification (convergence here being in some internal quadrature tolerance, not grid points), comparison to limiting cases with known analytic solutions, or comparison to

well reviewed published results. At a minimum, any self tests provided by the original stand-alone solver should be integrated into the ExactPack testing framework.

Tests for solvers are found in the `tests` directory, with file names `test_<solver>.py`. For examples of testing, see the files in that directory. The tests are built using the standard Python `unittest` framework.

A complete description of all the unit tests for ExactPack, as well as a discussion of the testing strategy, can be found in *Testing*.

CHAPTER

FIVE

REFERENCE GUIDE

This chapter provides a complete description of the ExactPack API including documentation of all of the available solvers.

5.1 Core Functionality

This section covers the core functionality of the package, including the base clases for solvers and solutions, and utility routines. For more user friendly introduction to the library, see the *User's Guide*.

By default the contents of the following modules are loaded into the global namespace when `exactpack` is imported.

5.1.1 `exactpack.base`

exception `exactpack.base.`**`UsingDefaultWarning`**
Warning category for use of default parameter settings.

Sometimes users of pre-packaged codes can be unaware of default values that those codes are assuming, and that can lead to confusion. The default behavior in ExactPack is to issue a *`UsingDefaultWarning`* in such cases to notify users what is happening. This behavior can be over-ridden using the `warnings` package.

class `exactpack.base.`**`Jump`**(*left*, *right=None*)
A class to hold values at jump points.

A *`Jump`* is a container for left and right states used by *`JumpCondition`* to hold the two limiting values at discontinuous points in an exact solution.

Parameters

- **`left`** – the left state
- **`right`** – the right state

There are three ways of constructing a *`Jump`* object.

1. If two arguments are provided (*left* and *right*), these are used for the left and right states.
2. Alternately, one tuple or another sequence argument can be provided, and the first two elements will be used for the left and right states.
3. Finally, providing a single argument of type *`Jump`* will act as a copy constructor, creating a new object that is a copy of the original.

`left` = None
The left state, or the limiting value at the point from below [$\lim_{x \to a^-} f(x)$].

right = None
The right state, or the limiting value at the point from above [$\lim_{x \to a^+} f(x)$].

class exactpack.base.**JumpCondition**(*location*, *description=''*, ***kwargs*)
A class for jump conditions.

By definition, weak solutions of differential equations may have discontinuities. These are points at which the solution, considered as a function, has no value, but for which the left and right limits on the function value are different. Numerically computed discrete solutions cannot directly capture discontinuities. The *JumpCondition* class is used to provide a numerical description of the mathematical properties of the jump.

Each solution discontinuity has a location and a set of variables for which the left and right states are provided. In addition to the attributes described below, a *JumpCondition* object will have an attribute for each problem variable, with a value of type *Jump* giving the left and right states.

For example, the jump in the Heaviside step function could be described by the following:

```
JumpCondition(location=0,
              description="Mathematical Discontinuity",
              H=(0, 1))
```

For more information see *More About Jump Conditions*.

Parameters

- **location** (*Number*) – the location of the jump point
- **description** (*str*) – a short description of the type of discontinuity (e.g., 'Shock', 'Material Interface')
- **kwargs** – the remaining keywords arguments set the jump conditions: the keywords are the variable names, and the values are either of type *Jump*, or a 2-tuple to be converted

location = None
The location of the jump point

description = None
A short description of the type of discontinuity

class exactpack.base.**ExactSolver**(***params*)
A virtual base class for ExactPack solvers.

Solvers are Python callable objects which can be used to generate solutions at specific points in time and space. A solver instance can be invoked as a function with two arguments. The first is a list of points at which to generate the exact solution, and the second is the time at which the solution is required. The points list can be a numpy array or a Python sequence (list or tuple) of points. Depending on the solver, each point may be 1-, 2-, or 3-dimensional, and may be in rectangular, cylindrical, or spherical coordinates. For 1-dimensional problems, the points are given by a rank-1 numpy array, or a list of scalar values. For 2- or 3-dimensional problems, the points are given by a numpy array of shape `(N,2)` or `(N,3)`, or by a list of 2- or 3-tuples. Check the documentation for a particular solver for details.

For an example of how to write an ExactSolver child class, see *Adding a New Solver*.

parameters = {}
A list of parameters which can be used as keyword arguments to the solver's constructor.

class exactpack.base.**ExactSolution**
A class for solutions returned by ExactPack solvers.

Parameters **data** – a sequence of numpy.ndarrays (including a rank-2 numpy.ndarray) to use as the fields of the *ExactSolution*.

exactpack.base.ExactSolution is derived from the numpy.ndarray, specifically a Numpy record array (Structured arrays). There are some special things that need to be done to take care of the way a numpy.ndarray is created. The code used here is based on the example in Subclassing ndarray.

For compatibility, the field names must follow a standardized naming convention. For any variable that has an entry in the following table, the listed field name will be used. Other problem specific variables, which are not listed in the table, should be fully described in the documentation for the specific solver.

Table 5.1: Standardized Variable Names

Field Name	Description
density	Density
pressure	Pressure
specific_internal_energy	Specific internal energy
velocity	Velocity along the problem direction
position	Generic position variable r or x along the problem direction
position_x	Cartesian position variable for x in 1D, 2D or 3D
position_y	Cartesian position variable for y in 2D or 3D
position_z	Cartesian position variable for z in 3D

Internally, numpy.core.records.fromarrays() is used to map the *data* to a structured array.

jumps = None
A list of *JumpCondition*s. Note the following important distinction: An empty list means a continuous solution (the solver is reporting there are no jumps), whereas a value of None means the solver is not reporting any information about jumps (that is, there may or may not be jumps in the analytic solution).

plot(*name*, ***kwargs*)
Plot one solution variable using matplotlib.

name is the name of the variable to plot.

The optional argument *scale* gives a scaling factor to be applied to the data before plotting. If scale='auto', then *plot()* will determine a scale factor so that the plotted data will be of order one.

If a *label* argument is given, it is used with no changes. If no *label* is given (or label=None), then the label defaults to *name* (with the optional *scale* indicated in the label if one is used).

All other keywords are passed directly to the matplotlib.pyplot.plot() command.

Note: ExactPack doesn't load matplotlib until the first time *plot()* is called. Scripts that need to select a different matplotlib backend can do this any time before *plot()* is first called.

---

plot_all()
Plot all variables.

Plot all the variables against radial distance, using auto scaling.

dump(*filename*)
Dump the solution variables to a CSV file.

5.1.2 exactpack.analysis

This module provides tools to perform code verification analysis.

class exactpack.analysis.code_verification.**PointNorm**(*ord=1*)
Pointwise error norm.

Returns a callable which computes the pointwise error norm according to the formula:

$$L_n(d) = \left[\frac{1}{N}\sum_i |d_i|^n\right]^{1/n}$$

where N is the number of points, and n is the order of the norm, *ord*.

Note: The norm is normalized by the number of points. This makes this norm suitable for intrinsic properties, like density or velocity, and not for extrinsic properties, like mass or momentum.

---

ord is the order of the norm.

class exactpack.analysis.code_verification.**CellNorm**(*weights='cell_volume'*, *ord=1*)
Volume weighted cell error norm.

Returns a callable which computes the volume weighted cell error norm according to the formula:

$$L_n(d) = \left[\sum_i |w_i d_i|^n\right]^{1/n}$$

where n is the order of the norm, *ord*. The weights w_i are chosen based on the dimensionality of the problem: For 1D Cartesian, they are the zone lengths, for 2D Cartesian, they are the zone areas, and for 2D axisymmetric and 3D they are the zone volumes.

Note: The default weights are the cell volumes. This is only valid intrinsic properties, like density or velocity, and not for extrinsic properties, like mass or momentum.

In general, if the weights are extrinsic properties, such as mass or volume, then the norm should be used on intrinsic variables. If the weights are intrinsic properties, then the norm should be used on extrinsic variables.

---

ord is the order of the norm and *weights* is the name of the variable to use for the weights.

class exactpack.analysis.code_verification.**Study**(*datasets*, *study_parameters*, *reference=None*, *time=None*, *reader=None*, *abscissa=None*)
Container object for building parameter studies.

The *Study* is a base class which loads data from multiple runs to be used in a parameter study, typically a solution convergence study. Child classes can be defined to implement specific types of analysis.

Create a study object.

If the *reader* option is not None, then *datasets* is a list of filenames to read data from. Otherwise, it is a list of actual data objects. The *study_parameters* are the numerical values which are used as the independent variable in the *Study*, typically the cell size or grid spacing.

The reference solution to compare to is supplied in the form of an ExactSolver instance passed to *reference*. The exact solution will be compared at *time*.

Note: If a file glob() is used to create the list of filenames, it is advisable that it be sorted(), so that the order of the cases is predictable for consistency with the *study_parameters* list.

---

datasets = None
A sequence of numpy.recarrays, one for each data set.

abscissa = None
The name of the variable which is the independent space variable for comparisons.

plot(*var*, *maxpts=10000*)
Plot the study.

Plot the spatial profiles of the variable *var* from each data set in the study and the exact reference solution (if one is defined). If the data contains more than *maxpts* points to plot, sample the data at a fixed interval to plot no more than *maxpts* points.

class exactpack.analysis.code_verification.**FilterStudy**(*source*)
A study built from another study using a filter.

A filter study is a class derived from *FilterStudy*, which takes one study and transforms it by applying a filter function to every data set in that study, to create a new study. To create a filter study, necessary to subclass *FilterStudy* and override the *filter()* method with a function which takes a single data set as an argument and returns a new data set.

Create a filtered study.

Filter the data in the *source* study using the *filter()* method and construct a new study that contains the filtered data fields.

filter(*dataset*)
Data set filter.

The filter function is applied to each data set when constructing the new filtered study. The default *filter()* method raises an exception, and must be over-ridden in child classes.

class exactpack.analysis.code_verification.**ExtractRegion**(*source*, *domain=(-inf*, *inf)*)
A study that extracts data in a particular region.

This filter creates a new study which only includes data within a specific region of space.

class exactpack.analysis.code_verification.**ComputeExact**(*source*)
A study that adds exact solution data.

This filter creates a new study which contains additional fields in each data set corresponding to the exact solution on the grid from that data set. Exact solution fields are added for each variable which is defined in both the source data set and the exact solution. The variable names in the new data set are the same as in the source, except that the string "Exact:" is pre-pended to each variable name.

Create a filtered study.

Filter the data in the *source* study using the *filter()* method and construct a new study that contains the filtered data fields.

filter(*dataset*)
Compute the exact solution corresponding to a data set.

Compute the exact solution on the same grid as the data set. Then, return a new data set containing the exact solution on that grid for each variable which is defined in both the source data set and the exact solution. The variable names in the new data set are the same as in the source, except that the string "Exact:" is prepended to each variable name.

class exactpack.analysis.code_verification.**ComputeErrors**(*source*)
A study object that adds error fields.

This filter creates a new study which contains additional fields in each data set corresponding to the error field. *ComputeErrors* assumes that the source study contains both code output data and exact solution data for one

or more variables, the latter being indicated by a variable name pre-pended with the string "Exact:". Typically this is obtained as the output from a *ComputeExact* study. The error is just the difference between the two fields, and is indicated in the output by the string "Error:" being prepended to the variable name.

Create a filtered study.

Filter the data in the *source* study using the `filter()` method and construct a new study that contains the filtered data fields.

class exactpack.analysis.code_verification.**ComputeNorms**(*study*, *norm=<exactpack.analysis.code_verification.PointNor object>*)

A study that computes error norms.

This filter creates a new study whose datasets have an `errors` attribute, which is a dictionary with keys that are variable names, and values that are the norms of the corresponding error variable.

Create a *ComputeNorms* object.

Create a new instance of *study* with the error norms added to each data set. The specific norm to use can be chosen with the *norm* parameter.

Note: It is extremely important to use the correct type of norm: a *PointNorm* for intrinsic quantities, and a *CellNorm* for extrinsic ones.

---

plot(*var*, **args*, ***kwargs*)

Plot the norms.

Plot the error norms of *var* versus the `study_parameters`. The remaining arguments are passed to `matplotlib.pyplot.plot()`.

class exactpack.analysis.code_verification.**GlobalConvergenceRate**(*study*, *norm=<exactpack.analysis.code_verifica object>*, *domain=(-inf, inf)*, *fiducials={}*)

Base class for convergence analaysis studies.

To perform convergence analysis in ExactPack, it is necessary first to pipe the data through a series of filters, and then to compute the convergence rate. This class is intended as a base class for convergence analysis; it performs all the requisite filtering, but does not compute the convergence rate. That computation should be done in a child class.

p(*var*)

The convergence rate for *var*.

goodness(*var*)

The goodness of fit for *var*.

plot_fit(**args*, ***kwargs*)

Plot the fit.

Since *GlobalConvergenceRate* is a base class, there is nothing to plot.

plot_fiducial(*var*, **args*, ***kwargs*)

Plot the fiducual curve.

Plot a fiducial line with slope (in log-log coordinates) specified in the `fiducials` dictionary, for comparision to the fitted data.

plot(*var*, **args*, ***kwargs*)
Plot the convergence study.

Plot the convergence study. A fiducial line is plotted if a fiducial `p` is available. If a symbol is specified, it is used for the data; the fit is plotted with a line only in the same color as the data. If more control over the plotting parameters is desired, *`plot()`* method of the `norms` attribute and the *`plot_fiducial()`* and *`plot_fit()`* methods can be called directly.

class exactpack.analysis.code_verification.**FitConvergenceRate**(*study*, *fitting_function=<function <lambda>>*, **args*, ***kwargs*)

A class which performs a convergence analysis using a curve fit.

When initialized with a *`Study`* instance, the *`FitConvergenceRate`* performs a convergence analysis using a optimization procedure to fit the study data to a specified fitting function.

Create a *`FitConvergenceRate`* object.

Use the data sets in *study* to compute the convergence rate, by fitting the data to the *fitting_function*. Any other arguments are passed to *`GlobalConvergenceRate`*.

fitting_function = None
The function to fit the convergence data to.

fits = None
A dictionary whose keys are the variables for which convergence analysis is avaiable, and values are the fit parameters. Each fit value is a tuple (*popt*, *pcov*) returned by `scipy.optimize.curve_fit()`, where *popt* is the optimal fit values for *`fitting_function`*, and *pcov* is the estimated covarience of *popt*. For more information, see the documentation for `scipy.optimize.curve_fit()`.

plot_fit(*var*, **args*, ***kwargs*)
Create a standard convergence rate plot.

This method uses `matplotlib.pyplot.plot()` to plot the error norms and the best fit converegence rate. It is necessary to do an explicit `show()` to display the plot. *var* is the variable to plot, and *label* is a label string to for labeling the plot. The string will be formatted with the variable name `var` and `popt` and `pcov` returned from `scipy.optimize.curve_fit()` passed as formatting values.

p(*var*)
The convergence rate for *var*.

goodness(*var*)
The goodness of fit for *var*.

class exactpack.analysis.code_verification.**RoacheConvergenceRate**(**args*, ***kwargs*)

A class which performs a convergence analysis using the generalized Richardson extrapolation.

When initialized with a *`Study`* instance, the *`RoacheConvergenceRate`* performs a convergence analysis using the generalized Richardson extrapolation, also known as Roache's formula.

Note: Convergence values are evaluated using pairs of data sets in the order in which they are passed to the study routine. If the grids are not in order, unexpected results may be obtained.

Create a *`RoacheConvergenceRate`* object.

fits = None
A dictionary whose keys are the variables for which convergence analysis is avaiable, and values are the

fit parameters. Since the Roache formula is applied to each pair of grids in sequence, this is a list of tuples, each containing a convergence rate and a slope.

static roache(*e1*, *e2*, *h1*, *h2*)
Return rate of convergence and pre-factor.

p(*var*)
An array of convergence rates for *var*.

As the Roache formula operates on solutions on exactly two grids, this function returns an array of p values.

plot_fit(*var*, **args*, ***kwargs*)
Create a standard convergence rate plot.

This method uses `matplotlib.pyplot.plot()` to plot the error norms and the best fit converegence rate. It is necessary to do an explicit `show()` to display the plot. *var* is the variable to plot, and *label* is a label string to for labeling the plot. The string will be formatted with the variable name `var` and the minimum and maximum *p* values `pmin` and `pmax` passed as formatting values.

class `exactpack.analysis.code_verification.`**RegressionConvergenceRate**(**args*, ***kwargs*)
A class which performs a convergence analysis using linear regression.

When initialized with a *Study* instance, the *RegressionConvergenceRate* performs a convergence analysis based on the error ansatz

$$\epsilon = Ah^p$$

by applying linear regression to the logarithm of the errors and deltas.

Create a *RegressionConvergenceRate* object.

fits = None
A dictionary whose keys are the variables for which convergence analysis is avaiable, and values are the fit parameters. Each fit value is a tuple (*slope*, *intercept*, *rvalue*, *pvalue*, *stderr*) returned by `scipy.stats.linregress()`, where *slope* is the convergence rate. For more information, see the documentation for `scipy.stats.linregress()`.

plot_fit(*var*, **args*, ***kwargs*)
Create a standard convergence rate plot.

This method uses `matplotlib.pyplot.plot()` to plot the error norms and the best fit converegence rate. It is necessary to do an explicit `show()` to display the plot. *var* is the variable to plot, and *label* is a label string to for labeling the plot. The string will be formatted with the variable name `var` and `popt` and `pcov` returned from `scipy.optimize.curve_fit()` passed as formatting values.

p(*var*)
The convergence rate for *var*.

goodness(*var*)
The goodness of fit for *var*.

5.1.3 exactpack.utils

`exactpack.utils.`**discover_solvers**()
Return a list of available solvers.

Solver discovery is performed by walking the directory tree containing the `exactpack` module, and looking for classes that inherit from *exactpack.base.ExactSolver*. The return value is a list of names of solver classes.

Since the directory walk is somewhat inefficient, it is suggested that the results be cached by the caller, if they are likely to be needed again.

Todo

Add a search directory list to *exactpack.utils.discover_solvers()*, so that users can develop other solvers outside the library.

5.2 Interfaces

In addition to using ExactPack as a Python package, it provides rudimentary command-line and graphical user interfaces. The internal implementation of these interfaces are documented here. For instructions on using the interfaces, see *Using ExactPack from the Command Line* and *The ExactPack Graphical User Interface*.

5.2.1 exactpack.gui

5.2.2 exactpack.cmdline

exactpack.cmdline.**main**()

A command line interface for ExactPack

For usage information, run:

```
exactpack --help
```

CHAPTER

SIX

SOLVERS

This chapter documents all the solver classes provided by ExactPack.

6.1 The Blake Problem

The spherically symmetric, isotropic, linear elastic Blake problem.

The Blake problem concerns the spherically symmetric propagation of radial, longitudinal waves from a cavity of radius, a, in a homogeneous, isotropic, linear elastic whole space, whose surface is loaded by a time-dependent normal traction or pressure. When the pressure history, $p(t)$, is a step or rapid increase in time followed by a suitable decay, the solution has application in modeling the behavior of an embedded explosive energy source. Indeed, this was the motivation for early analyses of the problem: see Jeffries[7], Sharpe[9], Blake[2], Denny and Johnson[4] and references therein.

6.1.1 Governing equations

Following Aldridge[1] (among others) we use spherical coordinates with origin at the cavity center for our analysis. Symmetry considerations require that the particle displacement field be purely radial so that,

$$\mathbf{u}(\mathbf{r}, t) = u(r, t)\, \hat{\mathbf{e}}_r \,, \tag{6.1}$$

where $\mathbf{r}$ is the position vector and $\hat{\mathbf{e}}_r$ is the radial unit vector. Thus, $u(r, t)$ is the *physical* radial component of displacement. The elastodynamic equation (6.4) for $\mathbf{u}$ is obtained by substituting the isotropic, linear elastic constitutive model (6.2) into the momentum equation (6.3) (in its body force- and body moment-free form),

$$\boldsymbol{T} = \lambda \,\mathrm{tr}\,(\boldsymbol{E})\, \mathbf{1} + 2\mu\, \boldsymbol{E} \,, \tag{6.2}$$

$$\boldsymbol{\nabla} \bullet \boldsymbol{T} = \frac{\partial^2 \boldsymbol{u}}{\partial t^2} \,, \tag{6.3}$$

[7] Jeffries, Harold. *On the Cause of Oscillatory Movement in Seismograms*. Geophysical Journal International (Oxford University Press) **2**, pp. 407-416 (1931).

[9] Sharpe, Joseph A. *The production of elastic waves by explosion pressures I, Theory and empirical field observations*. Geophysics (Society of Exploration Geophysicists) **7**, no. 2, pp. 144-154 (1942).

[2] Blake, F. G. *Spherial Wave Propagation in Solid Media*. The Journal of the Acoustical Society of America (Acoustical Society of America) **24**, no. 2, p. 211 (1952).

[4] Denny, Marvin D. and Lane R. Johnson. *The Explosion Seismic Source Function: Models and Scaling Laws Reviewed*. In "Explosion Source Phenomenology", by Steven R Taylor, Patton Howard J and Paul G Richards, pp. 1-24. Washington D.C.: American Geophysical Union (1991).

[1] Aldridge, David F. *Elastic Wave Radiation from a Pressurized Spherical Cavity*. Report SAND2002-1882, Sandia National Laboratories (SNL), Albuquerque, NM 87185: SNL (2002).

$$(\lambda+\mu)\,\boldsymbol{\nabla}(\boldsymbol{\nabla}\bullet\boldsymbol{u})+\mu\,\boldsymbol{\nabla}^2\boldsymbol{u}=\rho_0\frac{\partial^2\boldsymbol{u}}{\partial t^2}\,. \tag{6.4}$$

Here $\boldsymbol{T}$ is the (Cauchy) stress tensor, $\mathbf{1}$ the unit tensor, $\mathrm{tr}\,(\cdot)$ the trace of a 2-tensor, $\boldsymbol{\nabla}(\cdot)$ the gradient operator, and $\boldsymbol{\nabla}\bullet(\cdot)$ the divergence. The infinitesimal strain tensor, $\boldsymbol{E}$, is the symmetric part of $\boldsymbol{\nabla u}$, λ and μ are the (constant) Lame moduli and ρ_0 is the initial mass density. With use of the identity,

$$\boldsymbol{\nabla}^2\boldsymbol{u}=\boldsymbol{\nabla}(\boldsymbol{\nabla}\bullet\boldsymbol{u})-\boldsymbol{\nabla}\times(\boldsymbol{\nabla}\times\boldsymbol{u})\,, \tag{6.5}$$

(6.4) may be written as,

$$(\lambda+2\mu)\,\boldsymbol{\nabla}^2\boldsymbol{u}+(\lambda+\mu)\,\boldsymbol{\nabla}\times\,(\boldsymbol{\nabla}\times\boldsymbol{u})=\rho_0\frac{\partial^2\boldsymbol{u}}{\partial t^2}\,. \tag{6.6}$$

The motion described by (6.1) is curl-free so that (6.6) reduces to the vector wave equation,

$$\boldsymbol{\nabla}^2\boldsymbol{u}=\frac{1}{{c_L}^2}\frac{\partial^2\boldsymbol{u}}{\partial t^2} \tag{6.7}$$

where ${c_L}^2$ is the (squared) longitudinal or compressional wave speed of the material, see equation (6.13). Introducing (6.1) into (6.7) and expanding in spherical coordinates, we find that the radial component $u(r,t)$ satisfies the *scalar* wave equation in spherical coordinates,

$$\frac{\partial^2 u}{\partial r^2}+\frac{2}{r}\frac{\partial u}{\partial r}-\frac{2u}{r^2}=\frac{1}{{c_L}^2}\frac{\partial^2 u}{\partial t^2}. \tag{6.8}$$

As noted by (Hutchens 2005), further simplification obtains by introduction of a displacement potential, ϕ, so that (6.8) may be written as,

$$\frac{\partial}{\partial r}\left[\frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial\phi}{\partial r}\right)-\frac{1}{{c_L}^2}\frac{\partial^2\phi}{\partial t^2}\right]=0\,;\quad u(r,t)=\frac{\partial\phi}{\partial r}. \tag{6.9}$$

The outermost derivative in (6.9) is eliminated by equating the expression inside the square brackets to an arbitrary function of time, $C(t)$. Because we need only $u=\partial\phi/\partial r$ and its derivatives (but not ϕ itself) we may, without loss of generality, set $C(t)=0$. Finally, by introducing $(r\,\phi)$ as the field to be solved for, the governing equation reduces to a 1D Cartesian wave equation,

$$\frac{\partial^2(r\,\phi)}{\partial r^2}=\frac{1}{{c_L}^2}\frac{\partial^2(r\,\phi)}{\partial t^2}\,. \tag{6.10}$$

6.1.2 Formulation and solution

The formulation is completed by augmenting (6.10) with suitable boundary and initial conditions (BC and IC). At the cavity surface we require that the radial normal stress component equal the imposed normal traction or pressure history: $T_{rr}|_{r=a}=p(t)$, while at infinity $\boldsymbol{u}$ must vanish for all time. Substituting $\boldsymbol{E}=sym(\boldsymbol{\nabla u})$ into (6.2) and replacing u with $\partial\phi/\partial r$ these BCs may be written,

$$\left((\lambda+2\mu)\frac{\partial^2\phi}{\partial r^2}+\frac{2\lambda}{r}\frac{\partial\phi}{\partial r}\right)\Bigg|_{(r=a,\,t)}=-p(t);\quad \phi(r\to\infty,\,t)=0\,. \tag{6.11}$$

At the initial time the material is undistorted and quiescent: $u(r,t=0)=(\partial u/\partial t)(r,t=0)=0$.

In terms of ϕ the ICs become,

$$\phi(r,\,t=0)=0\,;\quad \left.\frac{\partial\phi}{\partial t}\right|_{t=0}=0\,, \tag{6.12}$$

where again we have set several arbitrary functions of time to zero.

A general solution to (6.10), (6.11), (6.12), may be constructed conveniently by use of the Laplace transform[10], see Hutchens[6] for details of the calculation. We introduce a rescaled time, $t'(r,t) = t - (r-a)/c_L$, and the definitions,

$$\alpha = \frac{2{c_T}^2}{a c_L}, \quad \beta^2 = \alpha^2 \left(\frac{{c_L}^2}{{c_T}^2} - 1 \right), \quad {c_L}^2 = \frac{\lambda + 2\mu}{\rho_0}, \quad {c_T}^2 = \frac{\mu}{\rho_0}, \tag{6.13}$$

where c_L and c_T are the longitudinal and shear wave speeds. Then, for an arbitrary $p(t)$, the time-domain solution may be written formally as,

$$\phi(r,t) = -\frac{a}{\rho}\frac{1}{\beta r} \int_0^{t'(r,t)} p\left(t'(r,t) - \tau\right) e^{-\alpha \tau} \sin(\beta \tau)\, d\tau \tag{6.14}$$

Provided that this integral is calculable in terms of elementary or tabulated special functions or a convergent series, the solution can be computed at high accuracy and can be useful for verification. All fields relevant for comparison with hydrocode results are obtained by calculating $\mathbf{E}$ from $\mathbf{u} = r\,\phi(r,t)\,\hat{\mathbf{e}}_r$ and then substituting into (6.2) with specific values of the Lame and other parameters. For the specific loading history, $p(t) = P_0\,H(t)$, where H is the Heaviside or unit-step function and P_0 the imposed pressure scale, (6.14) takes the form,

$$\phi(r,t') = -\frac{a}{\rho}\frac{P_0\,\mathrm{H}(t')}{(\alpha^2+\beta^2)\,r}\left[1 - e^{-\alpha t'}\left\{\cos(\beta t') + \frac{\alpha}{\beta}\sin(\beta t')\right\}\right]. \tag{6.15}$$

This is the case implemented in this initial version of the Blake problem solver in *ExactPack*.

By default, `exactpack.solvers.blake` loads `exactpack.solvers.blake.blake`.

6.1.3 References

6.1.4 exactpack.solvers.blake.blake

A python solver for the spherical, isotropic, linear elastic Blake problem.

This Blake version handles only spherical geometry with a constant (for $t > 0$) pressure history as in *Formulation and solution* above.

class exactpack.solvers.blake.blake.**Blake**(***kwargs*)

Compute a solution "snapshot" for the spherical Blake problem.

In the spherical Blake problem for an isotropic, linear-elastic material, solution discontinuities do not develop from smooth initial data because the governing equations and BCs are linear (6.10), (6.11). This class calculates the solution (the fundamental field being the displacement) in terms of the radial coordinate in the $t = 0$ *reference configuration* of the elastic space. In this domain, the solution is smooth because the pressure history is applied to the boundary (cavity surface). To assist with hydrocode comparison, we calculate the "curr_posn" array as the sum of the (input) referential radial coordinate and the displacement at the snapshot time.

Elastic Parameters

There are six parameters which are commonly used to characterize an isotropic, linear-elastic solid.

[10] Zwillinger, Daniel. CRC Standard Mathematical Tables and Formulae (32nd Edition), Taylor & Francis (2012).

[6] Hutchens, Gregory J. *An Analysis of the Blake Problem*. Report LA-UR-05-8737, Los Alamos National Laboratory (LANL), Los Alamos, NM 87545: LANL (2005).

Parameter Name	Symbol
First Lame Modulus	λ
Shear Modulus (Second Lame Modulus)	G
Young's Modulus	E
Poisson's Ratio	ν
Bulk Modulus	K
Longitudinal Modulus	M

The material is determined by specifying any **two** of these six; the other four are calculated internally. Consequently, class Blake requires any two of these to create a solver instance. For further detail, see the documentation for `set_elastic_params()` in module `set_check_elastic_params`.

Parameters

- **`cavity_radius`** – initial radius of cavity surface to which the pressure history is applied (m)
- **`shear_mod`** – Shear Modulus (Pa)
- **`bulk_mod`** – Bulk modulus (Pa)
- **`geometry`** – 3 = spherical (dimensionless)
- **`lame_mod`** – Lame modulus (Pa)
- **`pressure_scale`** – scale of pressure history imposed on cavity surface (Pa). In this version of the solver the pressure is constant for $t > 0$.
- **`youngs_mod`** – Young's modulus (Pa)
- **`poisson_ratio`** – Poisson's Ratio (dimensionless)
- **`long_mod`** – Longitudinal modulus (Pa)
- **`blake_debug`** – True/False flag to turn on available debugging code. Useful mainly for a developer/maintainer.
- **`ref_density`** – initial (uniform) mass density (kg/m**3)

Blake Class Snapshot Solver: Constructor

This function does the material and problem setup to create an instance of the spherical Blake class and returns the solver object. The solver produces an `ExactSolution` "snapshot" on the argument radial grid 'r' and at time 't', when invoked. When the constructor is called with *no arguments*, the solver for the *default* Blake problem is instantiated.

For the non-material problem parameters, only those for which a non-default value is desired need be specified. If *no* elastic (material) parameters are specified, the default material obtains. No other defaulting of the elastic parameters is performed.

Default Problem Definition

As our default for spherical Blake, we use the Los Alamos standard setup as defined by Brock[3]. This problem definition was also used by Kamm and Ankeny[8] in a hydrocode evaluation using their own Blake solver.

Spherical Snapshot Solver Output Fields

[3] Brock, Jerry S. *Blake Test Problem Parameters*. Report LA-UR-08-3005, Los Alamos National Laboratory (LANL), Los Alamos, NM 87545: LANL (2008).

[8] Kamm, James R., and Lee A. Ankeny. *Analysis of the Blake Problem with RAGE*. Report LA-UR-09-01255, Los Alamos National Laboratory (LANL), Los Alamos, NM 87545: LANL (2009).

Field Name	Description (physical components not tensor cmpts)
position	reference radial coordinate
curr_posn	current (snapshot time) radial coordinate
displacement	radial component of displacement vector
strain_rr	radial component of (infinitesimal) strain
strain_qq	circumfrential component of strain
strain_vol	volume strain = trace(strain tensor)
density	current mass density
stress_rr	radial component of stress
stress_qq	circumfrential component of stress
pressure	-(1/3) trace(stress tensor)
stress_dev_rr	radial component of stress deviator
stress_dev_qq	circumfrential component of stress deviator
stress_diff	abs(stress_rr - stress_qq)

Spherical Coordinates

We use a common physics convention.

(r, theta, phi) = (radius, co-latitiude, longitude), pos. phi measured from +x-axis to +y-axis, so that the basis vectors in this order form a right-handed triad. Indices 'q' and 'f' are used to indicate the theta and phi coorindates, respectively. In the current problem, by symmetry, the phi and theta components are equal, so only theta components are output.

6.1.5 exactpack.solvers.blake.set_check_elastic_params

Set and check values of a complete set of isotropic, linear elastic parameters.

exactpack.solvers.blake.set_check_elastic_params.**check_ii**(*prmcase*, *lbda_name*, *lbda*, *G_name*, *G*, *blk_dbg_prm*)

Terminate if the strain energy function is **not** PD.

Apply Gurtin's condition (ii)[5] to the elasticity parmeters to determine if the isotropic, linear elastic strain energy is positive definite (PD).

exactpack.solvers.blake.set_check_elastic_params.**check_iii**(*prmcase*, *G_name*, *G*, *K_name*, *K*, *blk_dbg_prm*)

Terminate if the strain energy function is **not** PD.

Apply Gurtin's condition (iii)[5] to the elasticity parmeters to determine if the isotropic, linear elastic strain energy is positive definite (PD).

exactpack.solvers.blake.set_check_elastic_params.**check_iv**(*prmcase*, *G_name*, *G*, *nu_name*, *nu*, *blk_dbg_prm*)

Terminate if the strain energy function is **not** PD.

Apply Gurtin's condition (iv)[5] to the elasticity parmeters to determine if the isotropic, linear elastic strain energy is positive definite (PD).

exactpack.solvers.blake.set_check_elastic_params.**check_v**(*prmcase*, *E_name*, *E*, *nu_name*, *nu*, *blk_dbg_prm*)

Terminate if the strain energy function is **not** PD.

[5] Morton E. Gurtin, "Linear Theory of Elasticity" in Mechanics of Solids, ed. C. Truesdell, Springer, 1983. Reprinted from "Handbuch der Physik", vol. VIa/2, ed. S. Flügge, Springer, 1972.

Apply Gurtin's condition (v)[5] to the elasticity parmeters to determine if the isotropic, linear elastic strain energy is positive definite (PD).

exactpack.solvers.blake.set_check_elastic_params.**warn_negative_poisson**(*prmcase*, *nu*, *pnm*, *pv*, *blk_dbg_prm*)

Issue a warning if Poisson's Ratio is non-positive.

This function should be invoked whenever the isotropic linear elastic material defined by two positive moduli *could* have a non-positive Poisson's Ratio. If the value *is* non-positive, an informational warning is issued as such materials are uncommon and the negative sign might be due to an input error. To reliably specify a material with negative Poisson's Ratio, the poisson_ratio parameters should be used explicitly.

exactpack.solvers.blake.set_check_elastic_params.**term_cmplx_poisson**(*prmcase*, *pnm*, *pv*, *blk_dbg_prm*)

Raise a value error if Poisson's Ratio will be complex.

Certain combinations of two positive moduli *can* produce a complex value for Poisson's Ratio.

exactpack.solvers.blake.set_check_elastic_params.**term_nan_poisson**(*prmcase*, *pnm*, *pv*, *blk_dbg_prm*)

Raise a value error if Poisson's Ratio is a NaN.

Certain combinations of two positive moduli *can* produce a NaN for Poisson's Ratio, due to divide by zero.

exactpack.solvers.blake.set_check_elastic_params.**term_nan_lame**(*prmcase*, *pnm*, *pv*, *blk_dbg_prm*)

Raise a value error if the first Lame modulus is a Nan.

Certain combinations of two positive moduli *can* produce a NaN for the Lame modulus, due to divide by zero.

exactpack.solvers.blake.set_check_elastic_params.**set_elastic_params**(*elas_prm_names*, *elas_prm_dflt_vals*, *elas_prm_order*, *defaulted*, *blk_dbg_prm*, ***kwargs*)

Calculate and check a full set of isotropic, linear elastic parameters.

An isotropic, linear elastic solid is defined by specifying, via the kwargs dict, any **two** of the **six** elastic parameters defined in class Blake. This function computes the remaining four elastic parameters and performs some sanity checking. In particular we require that,

1. Each user-specified modulus parameter is positive.
2. Each pair of user-specified parameters define a material which has a positive-definite (PD) strain energy function.
3. If Poisson's Ratio is negative when calculated from user-specified moduli, a non-fatal warning message to that effect is issued. A negative value is uncommon for materials in their linear elastic range. Moreover, if the user *intends* that Poisson's Ratio be negative, then it should be passed to the Blake constructor *explicitly*. When Poisson's Ratio is inferred from moduli values, a square root sign ambiguity may prevent the correct sign being set.

If conditions 1 or 2 aren't satisfied, an error message is issued and *ExactPack* terminates. Otherwise, the full parameter set is returned as a dictionary of the material parameter names and values. When True is passed as

the value of argument "defaulted", a dictionary of default values is constructed from the first two arguments and returned. This is the *only* default case accepted by this function.

NOTE : It is the caller's responsibility to ensure that the order of elements in elas_prm_names and elas_prm_dflt_vals agrees with each other and with the ordinality specified in elas_prm_order. Even if defaulted is *False*, the elas_prm_names and elas_prm_order arguments must still agree. In particular, the elas_prm_names list *must* contain the same six strings as does the corresponding attribute of `Blake`.

The blk_dbg_prm argument is True/False and enables available debugging code throughout this module when True.

6.2 Coggeshall Problems

The Coggeshall problems.

The Coggeshall *[Coggeshall1991]* problems are a collection of exact solutions to the one-dimensional Euler equations with heat conduction and no viscosity. The fluid field variables are the mass density $\rho(r,t)$, the fluid velocity $u(r,t)$, the fluid temperature $T(r,t)$, and the specific internal energy $e(r,t)$ of the fluid material, where r is the spatial coordinate in planar, cylindrical, or spherical geometry. The γ-law Equation of State (EOS) for the gas is written

$$P = \Gamma\,\rho T$$
$$e = \frac{\Gamma\,T}{\gamma-1}\,,$$

where Γ is the Gruneisen gas constant and $\gamma \equiv c_{\rm p}/c_{\rm v}$ is the adiabatic exponent. Dividing these equations gives the standard adiabatic γ-law EOS,

$$P = (\gamma-1)\,\rho e\,.$$

Upon taking ρ, u, and T as the independent variables, the conservation of mass, momentum, and energy imply the following balance equations:

$$\frac{\partial\rho}{\partial t} + u\frac{\partial\rho}{\partial r} + \rho\frac{\partial u}{\partial r} + \frac{k\rho u}{r} = 0$$
$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial r} + \frac{\Gamma T}{\rho}\frac{\partial\rho}{\partial r} + \Gamma\frac{\partial T}{\partial r} = 0$$
$$\frac{T}{\gamma-1}\left[\frac{\partial T}{\partial t} + u\frac{\partial T}{\partial r}\right] + \Gamma T\frac{\partial u}{\partial r} + \Gamma T\frac{ku}{r} + \frac{1}{\rho}\left[\frac{\partial F}{\partial r} + \frac{kF}{r}\right] = 0\,,$$

where $k = 0, 1, 2$ is the geometry factor specifying planar, cylindrical, or spherical coordinates, respectively. Note that the geometry option in ExactPack is different, with $\mathbf{geometry} = k+1 = 1, 2, 3$. The quantity F in the energy equation is the magnitude of the heat flux vector, which, in the diffusion approximation, takes the form

$$\vec{F} = -\frac{c\,\lambda}{3}\,\vec{\nabla}\,aT^4\,,$$

with c being the speed of light, a the radiation constant, and λ the radiation mean-free-path. To obtain a semi-analytic solution, we parameter λ by

$$\lambda(\rho,T) = \lambda_0\,\rho^\alpha T^\beta, \tag{6.16}$$

where λ_0 is a dimensionfull constant, and α and β are dimensionless parameters in the ranges $-1 \le \alpha \le 2$ and $1 \le \beta \le 3$. The mean-free-path λ is related to the Rosseland mean opacity κ by $\kappa = 1/\lambda\rho$. The heat flux can also be written in term of the heat conductivity K, defined by

$$\vec{F} = \frac{4c\lambda aT^3}{3}\,\vec{\nabla}T \equiv K(\rho,T)\vec{\nabla}T.$$

6.2.1 exactpack.solvers.cog.cog1

A Cog1 solver in Python.

This is a pure Python implementation of the Cog1 solution using Numpy. The exact solution takes the form,

$$\rho(r,t) = \rho_0\, r^b\, t^{-b-k-1}$$
$$u(r,t) = \frac{r}{t}$$
$$T(r,t) = T_0\, r^{-b}\, t^{b-(\gamma-1)(k+1)}$$

Free parameters: b, k, ρ_0, T_0, and γ.

class exactpack.solvers.cog.cog1.**Cog1**(***kwargs*)
Computes the solution to the Cog1 problem.

Parameters

- **b** – free param
- **geometry** – 1=planar, 2=cylindrical, 3=spherical
- **rho0** – density coefficient
- **temp0** – temperature coefficient
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$
- **Gamma** – Gruneisen gas parameter

class exactpack.solvers.cog.cog1.**PlanarCog1**(***kwargs*)
The planar Cog1.

Parameters

- **b** – free param
- **rho0** – density coefficient
- **temp0** – temperature coefficient
- **Gamma** – Gruneisen gas parameter
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog1.**CylindricalCog1**(***kwargs*)
The cylindrical Cog1.

Parameters

- **b** – free param
- **rho0** – density coefficient
- **temp0** – temperature coefficient
- **Gamma** – Gruneisen gas parameter
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog1.**SphericalCog1**(***kwargs*)
The spherical Cog1.

Parameters

- **b** – free param

- **rho0** – density coefficient
- **temp0** – temperature coefficient
- **Gamma** – Gruneisen gas parameter
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

6.2.2 exactpack.solvers.cog.cog2

A Cog2 solver in Python.

This is a pure Python implementation of the Cog2 solution using Numpy. The exact solution takes the form,

$$\rho(r,t) = \rho_0\, r^b\, t^{-2(b+k+1)/[2+(\gamma-1)(k+1)]}$$

$$u(r,t) = \frac{2}{2+(\gamma-1)(k+1)}\,\frac{r}{t}$$

$$T(r,t) = \frac{2(\gamma-1)(k+1)}{\Gamma(b+2)[2+(\gamma-1)(k+1)]^2}\,\left(\frac{r}{t}\right)^2$$

Free parameters: b, k, ρ_0, γ, and Γ.

class exactpack.solvers.cog.cog2.**Cog2**(***kwargs*)
Computes the solution to the Cog2 problem.

Parameters

- **geometry** – 1=planar, 2=cylindrical, 3=spherical
- **rho0** – density coefficient
- **b** – free dimensionless parameter
- **Gamma** – Gruneisen gas parameter
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog2.**PlanarCog2**(***kwargs*)
The planar Cog2.

Parameters

- **rho0** – density coefficient
- **b** – free dimensionless parameter
- **Gamma** – Gruneisen gas parameter
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog2.**CylindricalCog2**(***kwargs*)
The cylindrical Cog2.

Parameters

- **rho0** – density coefficient
- **b** – free dimensionless parameter
- **Gamma** – Gruneisen gas parameter
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog2.**SphericalCog2**(***kwargs*)
The spherical Cog2.

Parameters

- **rho0** – density coefficient
- **b** – free dimensionless parameter
- **Gamma** – Gruneisen gas parameter
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

6.2.3 exactpack.solvers.cog.cog3

A Cog3 solver in Python.

This is a pure Python implementation of the Cog3 solution using Numpy. The exact solution takes the form,

$$\rho(r,t) = \rho_0\, r^{b-k-1}\, e^{bt}$$
$$u(r,t) = -\frac{b}{v}\cdot r$$
$$T(r,t) = \frac{b^2}{v^2\,\Gamma(k-v-1)}\cdot r^2$$
$$\gamma = \frac{k-1}{k+1}$$

Free parameters: v, b, k, ρ_0, and Γ. Note that $\gamma < 1$.

class exactpack.solvers.cog.cog3.**Cog3**(***kwargs*)
Computes the solution to the Cog3 problem.

Parameters

- **geometry** – 1=planar, 2=cylindrical, 3=spherical
- **rho0** – density coefficient
- **b** – free dimensionless parameter
- **Gamma** – Gruneisen gas parameter
- **v** – free parameter with dimensions of velocity

class exactpack.solvers.cog.cog3.**PlanarCog3**(***kwargs*)
The planar Cog3 problem.

Parameters

- **rho0** – density coefficient
- **b** – free dimensionless parameter
- **Gamma** – Gruneisen gas parameter
- **v** – free parameter with dimensions of velocity

class exactpack.solvers.cog.cog3.**CylindricalCog3**(***kwargs*)
The cylindrical Cog3 problem.

Parameters

- **rho0** – density coefficient
- **b** – free dimensionless parameter
- **Gamma** – Gruneisen gas parameter

- **v** – free parameter with dimensions of velocity

class exactpack.solvers.cog.cog3.**SphericalCog3**(***kwargs*)
The spherical Cog3 problem.

Parameters

- **rho0** – density coefficient
- **b** – free dimensionless parameter
- **Gamma** – Gruneisen gas parameter
- **v** – free parameter with dimensions of velocity

6.2.4 exactpack.solvers.cog.cog4

A Cog4 solver in Python.

This is a pure Python implementation of the Cog4 solution using Numpy. The exact solution takes the form,

$$\rho(r,t) = \rho_0\, r^{-2k/(\gamma+1)}$$
$$u(r,t) = u_0\, r^{-k(\gamma-1)/(\gamma+1)}$$
$$T(r,t) = \frac{u_0^2(1-\gamma)}{2\gamma\Gamma}\, r^{-2k(\gamma-1)/(\gamma+1)}$$

Free parameters: k, u_0, ρ_0, γ, and Γ. The only physical solutions for which $T > 0$ are for $\gamma < 1$.

class exactpack.solvers.cog.cog4.**Cog4**(***kwargs*)
Computes the solution to the Cog4 problem.

Parameters

- **geometry** – 1=planar, 2=cylindrical, 3=spherical
- **rho0** – density coefficient
- **Gamma** – Gruneisen gas parameter
- **u0** – velocity coefficient
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$ (must be < 1)

class exactpack.solvers.cog.cog4.**PlanarCog4**(***kwargs*)
The planar Cog4.

Parameters

- **Gamma** – Gruneisen gas parameter
- **rho0** – density coefficient
- **u0** – velocity coefficient
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$ (must be < 1)

class exactpack.solvers.cog.cog4.**CylindricalCog4**(***kwargs*)
The cylindrical Cog4.

Parameters

- **Gamma** – Gruneisen gas parameter
- **rho0** – density coefficient

- **u0** – velocity coefficient
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$ (must be < 1)

class exactpack.solvers.cog.cog4.**SphericalCog4**(***kwargs*)
The spherical Cog4.

Parameters

- **Gamma** – Gruneisen gas parameter
- **rho0** – density coefficient
- **u0** – velocity coefficient
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$ (must be < 1)

6.2.5 exactpack.solvers.cog.cog5

A Cog5 solver in Python.

This is a pure Python implementation of the Cog5 solution using Numpy. The exact solution takes the form,

$$\rho(r,t) = \rho_0\, r^{-2}$$
$$u(r,t) = u_0\, t$$
$$T(r,t) = \frac{u_0}{\Gamma} \cdot r$$
$$k = 2 \text{ and } \gamma = \frac{1}{2}$$

Free parameters: u_0, ρ_0, and Γ.

class exactpack.solvers.cog.cog5.**Cog5**(***kwargs*)
Computes the solution to the Cog5 problem.

Parameters

- **rho0** – density coefficient
- **u0** – velocity coefficient
- **Gamma** – Gruneisen gas parameter

6.2.6 exactpack.solvers.cog.cog6

A Cog6 solver in Python.

This is a pure Python implementation of the Cog6 solution using Numpy. The exact solution takes the form,

$$\rho(r,t) = \rho_0 \frac{r^b}{(\tau^2 - t^2)^{(k+1+b)/2}}$$
$$u(r,t) = -\frac{rt}{\tau^2 - t^2}$$
$$T(r,t) = \frac{\tau^2}{\Gamma(b+2)} \cdot \frac{r^2}{(\tau^2 - t^2)^2}$$
$$\gamma = \frac{k+3}{k+1}$$

Free parameters: b, k, ρ_0, τ, and Γ. For $b = 3$, $k = 2$ (spherical with $\gamma = 5/3$), this becomes the 1974 Kidder solution [R.E. Kidder, Nucl. Fusion **14** (1974) 53]

class exactpack.solvers.cog.cog6.**Cog6**(***kwargs*)
Computes the solution to the Cog6 problem.

Parameters

- **geometry** – 1=planar, 2=cylindrical, 3=spherical
- **tau** – free parameter with dimensions of time
- **rho0** – density coefficient
- **b** – free dimensionless parameter
- **Gamma** – Gruneisen gas parameter

class exactpack.solvers.cog.cog6.**PlanarCog6**(***kwargs*)
The planar Cog6 problem.

Parameters

- **tau** – free parameter with dimensions of time
- **rho0** – density coefficient
- **b** – free dimensionless parameter
- **Gamma** – Gruneisen gas parameter

class exactpack.solvers.cog.cog6.**CylindricalCog6**(***kwargs*)
The cylindrical Cog6 problem.

Parameters

- **tau** – free parameter with dimensions of time
- **rho0** – density coefficient
- **b** – free dimensionless parameter
- **Gamma** – Gruneisen gas parameter

class exactpack.solvers.cog.cog6.**SphericalCog6**(***kwargs*)
The spherical Cog6 problem.

Parameters

- **tau** – free parameter with dimensions of time
- **rho0** – density coefficient
- **b** – free dimensionless parameter
- **Gamma** – Gruneisen gas parameter

class exactpack.solvers.cog.cog6.**Kidder74**(***kwargs*)
Cog6 reduces to the 1974 Kidder solution for geometry=3, b=3, and $\gamma = 5/3$.

Parameters

- **tau** – free parameter with dimensions of time
- **rho0** – density coefficient
- **Gamma** – Gruneisen gas parameter

6.2.7 exactpack.solvers.cog.cog7

A Cog7 solver in Python.

This is a pure Python implementation of the Cog7 solution using Numpy. The exact solution takes the form,

$$\rho(r,t) = \frac{R_0^{b/\gamma}\,\tau^{[(k+1)\gamma-1-b]/(\gamma-1)}}{\left(R_0^{2-b/\gamma} - R_i^{2-b/\gamma}\right)^{1/(\gamma-1)}} \cdot r^{-k-1} \cdot \left(\frac{r}{(\tau^2-t^2)^{1/2}}\right)^{k+1-b/\gamma} \cdot \left[\left(\frac{r}{(\tau^2-t^2)^{1/2}}\right)^{2-b/\gamma} - \left(\frac{R_i}{\tau}\right)^{2-b/\gamma}\right]^{1/(\gamma-1)}$$

$$u(r,t) = -\frac{rt}{\tau^2-t^2}$$

$$T(r,t) = \frac{\tau^2(\gamma-1)}{\Gamma(2\gamma-b)} \cdot r^{-2} \left(\frac{r}{(\tau^2-t^2)^{1/2}}\right)^{2+b/\gamma} \cdot \left[\left(\frac{r}{(\tau^2-t^2)^{1/2}}\right)^{2-b/\gamma} - \left(\frac{R_i}{\tau}\right)^{2-b/\gamma}\right]$$

$$\gamma = \frac{k+3}{k+1}$$

Free parameters: b, k, τ, R_0, R_i, and Γ. For $b = 0$, $k = 2$ (spherical with $\gamma = 5/3$), this becomes Kidder's 1976 solution [R.E. Kidder, Nucl. Fusion **16** (1976) 33].

class exactpack.solvers.cog.cog7.**Cog7**(***kwargs*)

Computes the solution to the Cog7 problem.

Parameters

- **tau** – free parameter
- **b** – free dimensionless parameter
- **R0** – free parameter with dimensions of length
- **geometry** – 1=planar, 2=cylindrical, 3=spherical
- **Ri** – free parameter with dimensions of length
- **Gamma** – Gruneisen gas parameter

class exactpack.solvers.cog.cog7.**PlanarCog7**(***kwargs*)

The planar Cog7.

Parameters

- **tau** – free parameter
- **b** – free dimensionless parameter
- **R0** – free parameter with dimensions of length
- **Ri** – free parameter with dimensions of length
- **Gamma** – Gruneisen gas parameter

class exactpack.solvers.cog.cog7.**CylindricalCog7**(***kwargs*)

The cylindrical Cog7.

Parameters

- **tau** – free parameter
- **b** – free dimensionless parameter
- **R0** – free parameter with dimensions of length
- **Ri** – free parameter with dimensions of length
- **Gamma** – Gruneisen gas parameter

class exactpack.solvers.cog.cog7.**SphericalCog7**(***kwargs*)
The spherical Cog7.

Parameters

- **tau** – free parameter
- **b** – free dimensionless parameter
- **R0** – free parameter with dimensions of length
- **Ri** – free parameter with dimensions of length
- **Gamma** – Gruneisen gas parameter

class exactpack.solvers.cog.cog7.**Kidder76**(***kwargs*)
Cog7 reduces to the 1976 Kidder solution for geometry=3, b=0.

Parameters

- **tau** – free parameter
- **R0** – free parameter with dimensions of length
- **Ri** – free parameter with dimensions of length
- **Gamma** – Gruneisen gas parameter

6.2.8 exactpack.solvers.cog.cog8

A Cog8 solver in Python.

This is a pure Python implementation of the Cog8 solution using Numpy. The solution takes a particularly simple analytic form,

$$\begin{aligned}
\rho(r,t) &= \rho_0\, r^{(k-1)/(\beta-\alpha+4)} t^{-(k+1)-(k-1)/(\beta-\alpha+4)} \\
u(r,t) &= \frac{r}{t} \\
T(r,t) &= T_0\, r^{(1-k)/(\beta-\alpha+4)} t^{(1-\gamma)(k+1)+(k-1)/(\beta-\alpha+4)} \,.
\end{aligned}$$

Free parameters: α, β, k, ρ_0, T_0, and γ.

For the values $\alpha = -1$, $\beta = 2$, $\gamma = 5/3$, and $k = 2$ (spherical), the solution takes the form:

$$\begin{aligned}
\rho(r,t) &= \rho_0\, r^{1/7} t^{-22/7} \\
u(r,t) &= r t^{-1} \\
T(r,t) &= \rho_0\, r^{-1/7} t^{-13/7} \,.
\end{aligned}$$

class exactpack.solvers.cog.cog8.**Cog8**(***kwargs*)
Computes the solution to the Cog8 problem.

Parameters

- **geometry** – 1=planar, 2=cylindrical, 3=spherical
- **rho0** – density coefficient
- **beta** – dimensionless constant β in Eq. (6.16)
- **temp0** – temperature coefficient
- **Gamma** – Gruneisen gas parameter
- **alpha** – dimensionless constant α in Eq. (6.16)
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog8.**PlanarCog8**(***kwargs*)
The planar Cog8 problem.

Parameters

- **rho0** – density coefficient
- **beta** – dimensionless constant β in Eq. (6.16)
- **temp0** – temperature coefficient
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$
- **alpha** – dimensionless constant α in Eq. (6.16)
- **Gamma** – Gruneisen gas parameter

class exactpack.solvers.cog.cog8.**CylindricalCog8**(***kwargs*)
The cylindrical Cog8 problem.

Parameters

- **rho0** – density coefficient
- **beta** – dimensionless constant β in Eq. (6.16)
- **temp0** – temperature coefficient
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$
- **alpha** – dimensionless constant α in Eq. (6.16)
- **Gamma** – Gruneisen gas parameter

class exactpack.solvers.cog.cog8.**SphericalCog8**(***kwargs*)
The spherical Cog8 problem.

Parameters

- **rho0** – density coefficient
- **beta** – dimensionless constant β in Eq. (6.16)
- **temp0** – temperature coefficient
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$
- **alpha** – dimensionless constant α in Eq. (6.16)
- **Gamma** – Gruneisen gas parameter

6.2.9 exactpack.solvers.cog.cog8_timmes

An implementation of the Cog8 solution in Fortran written by Frank Timmes.

This is a Fortran based solver for the Cog8 solution, as implemented by Frank Timmes. The original Fortran source code is available at Frank Timmes' website, under release LA-CC-05-101. The exact solution is quite simple, and takes the form

$$\begin{aligned}\rho(r,t) &= \rho_0\, r^{(k-1)/(\beta-\alpha+4)} t^{-(k+1)-(k-1)/(\beta-\alpha+4)} \\ u(r,t) &= \frac{r}{t} \\ T(r,t) &= T_0\, r^{(1-k)/(\beta-\alpha+4)} t^{(1-\gamma)(k+1)+(k-1)/(\beta-\alpha+4)}\ .\end{aligned}$$

Free parameters: k, γ, c_v, α, β, ρ_0, and T_0. For the specific values $\alpha = -1$, $\beta = 2$, $\gamma = 5/3$, and $k = 2$ (spherical), the solution takes the simple form,

$$\begin{aligned}\rho(r,t) &= \rho_0\, r^{1/7} t^{-22/7} \\ u(r,t) &= r t^{-1} \\ T(r,t) &= \rho_0\, r^{-1/7} t^{-13/7}\ .\end{aligned}$$

class exactpack.solvers.cog.cog8_timmes.**Cog8**(***params*)

Computes the solution to the Cog8 problem.

Parameters

- **rho0** – initial density of the gas
- **beta** – dimensionless constant β in Eq. (6.16)
- **temp0** – temperature of the gas
- **alpha** – dimensionless constant α in Eq. (6.16)
- **cv** – specific heat at constant volume [erg/g/eV]
- **gamma** – ratio of specific heats $\gamma \equiv c_p/c_v$

6.2.10 exactpack.solvers.cog.cog9

A Cog9 solver in Python.

This is a pure Python implementation of the Cog9 solution using Numpy. The exact solution takes the form,

$$\begin{aligned}\rho(r,t) &= \rho_0\, r^{-(2\beta+k+7)/\alpha}\, t^{-2[\alpha(k+1)-2\beta-k-7]/\alpha[2+(\gamma-1)(k+1)]} \\ u(r,t) &= \frac{2}{2+(\gamma-1)(k+1)}\,\frac{r}{t} \\ T(r,t) &= \frac{2\alpha(\gamma-1)(k+1)}{\Gamma[2+(\gamma-1)(k+1)]^2(2\alpha-2\beta-k-7)}\cdot\left(\frac{r}{t}\right)^2\end{aligned}$$

Free parameters: α, β, k, ρ_0, γ, and Γ.

class exactpack.solvers.cog.cog9.**Cog9**(***kwargs*)

Computes the solution to the Cog9 problem.

Parameters

- **geometry** – 1=planar, 2=cylindrical, 3=spherical
- **rho0** – density coefficient

- **beta** – dimensionless constant β in Eq. (6.16)
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$
- **alpha** – dimensionless constant α in Eq. (6.16)
- **Gamma** – Gruneisen gas parameter

class exactpack.solvers.cog.cog9.**PlanarCog9**(***kwargs*)
The planar Cog9 problem.

Parameters

- **alpha** – dimensionless constant α in Eq. (6.16)
- **beta** – dimensionless constant β in Eq. (6.16)
- **rho0** – density coefficient
- **Gamma** – Gruneisen gas parameter
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog9.**CylindricalCog9**(***kwargs*)
The cylindrical Cog9 problem.

Parameters

- **alpha** – dimensionless constant α in Eq. (6.16)
- **beta** – dimensionless constant β in Eq. (6.16)
- **rho0** – density coefficient
- **Gamma** – Gruneisen gas parameter
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog9.**SphericalCog9**(***kwargs*)
The spherical Cog9 problem.

Parameters

- **alpha** – dimensionless constant α in Eq. (6.16)
- **beta** – dimensionless constant β in Eq. (6.16)
- **rho0** – density coefficient
- **Gamma** – Gruneisen gas parameter
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

6.2.11 exactpack.solvers.cog.cog10

A Cog10 solver in Python.

This is a pure Python implementation of the Cog10 solution using Numpy. The exact solution takes the form,

$$\rho(r,t) = \rho_0\, r^{-k}$$
$$u(r,t) = \frac{4ca\lambda_0}{3}\,\frac{\gamma-1}{\Gamma\gamma}\cdot k\rho_0^{\alpha-1}\,T_0^{\beta+3}$$
$$T(r,t) = T_0\, r^k$$
$$\alpha = \beta + 4 - 1/k \;\; (k \neq 0)$$

Free parameters: β, k, ρ_0, T_0, λ_0, γ, and Γ.

class exactpack.solvers.cog.cog10.**Cog10**(***kwargs*)
Computes the solution to the Cog10 problem.

Parameters

- **geometry** – 2=cylindrical, 3=spherical
- **rho0** – density coefficient
- **beta** – dimensionless constant β in Eq. (6.16)
- **temp0** – temperature coefficient
- **Gamma** – Gruneisen gas parameter
- **lambda0** – constant λ_0 in Eq. (6.16)
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog10.**CylindricalCog10**(***kwargs*)
The cylindrical Cog10 problem.

Parameters

- **rho0** – density coefficient
- **beta** – dimensionless constant β in Eq. (6.16)
- **temp0** – temperature coefficient
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$
- **lambda0** – constant λ_0 in Eq. (6.16)
- **Gamma** – Gruneisen gas parameter

class exactpack.solvers.cog.cog10.**SphericalCog10**(***kwargs*)
The spherical Cog10 problem.

Parameters

- **rho0** – density coefficient
- **beta** – dimensionless constant β in Eq. (6.16)
- **temp0** – temperature coefficient
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$
- **lambda0** – constant λ_0 in Eq. (6.16)
- **Gamma** – Gruneisen gas parameter

6.2.12 exactpack.solvers.cog.cog11

A Cog11 solver in Python.

This is a pure Python implementation of the Cog11 solution using Numpy. The exact solution takes the form,

$$\rho(r,t) = \rho_0\, r^{(\gamma-1)(k+1)-2}\, t^{1-k-(\gamma-1)(k+1)}$$
$$u(r,t) = \frac{r}{t}$$
$$T(r,t) = T_0\, r^{2-(\gamma-1)(k+1)}\, t^{-2}$$
$$\alpha = \beta + 4 + \frac{k-1}{2-(\gamma-1)(k+1)}$$

Free parameters: k, ρ_0, T_0, γ, and β (with α a function of k, β and γ).

class exactpack.solvers.cog.cog11.**Cog11**(***kwargs*)
Computes the solution to the Cog11 problem.

Parameters

- **geometry** – 1=planar, 2=cylindrical, 3=spherical
- **rho0** – density coefficient
- **beta** – dimensionless constant β in Eq. (6.16)
- **temp0** – temperature coefficient
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$
- **Gamma** – Gruneisen gas parameter

class exactpack.solvers.cog.cog11.**PlanarCog11**(***kwargs*)
The planar Cog11 problem.

Parameters

- **beta** – dimensionless constant β in Eq. (6.16)
- **rho0** – density coefficient
- **temp0** – temperature coefficient
- **Gamma** – Gruneisen gas parameter
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog11.**CylindricalCog11**(***kwargs*)
The cylindrical Cog11 problem.

Parameters

- **beta** – dimensionless constant β in Eq. (6.16)
- **rho0** – density coefficient
- **temp0** – temperature coefficient
- **Gamma** – Gruneisen gas parameter
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog11.**SphericalCog11**(***kwargs*)
The spherical Cog11 problem.

Parameters

- **beta** – dimensionless constant β in Eq. (6.16)
- **rho0** – density coefficient

- **temp0** – temperature coefficient
- **Gamma** – Gruneisen gas parameter
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

6.2.13 exactpack.solvers.cog.cog12

A Cog12 solver in Python.

This is a pure Python implementation of the Cog12 solution using Numpy. The exact solution takes the form,

$$\rho(r,t) = \rho_0\, r^{-2k/(\gamma+1)}$$
$$u(r,t) = u_0\, r^{k(1-\gamma)/(1+\gamma)}$$
$$T(r,t) = \frac{u_o^2\,(1-\gamma)}{2\Gamma\gamma} \cdot r^{2k(1-\gamma)/(1+\gamma)}$$
$$\alpha = (\beta+4)(1-\gamma) + \frac{(k-1)(\gamma+1)}{2k} \quad (k \neq 0)$$

Free parameters: k, ρ_0, u_0, γ, Γ, and β (with α a function of k, β, and γ). Note that $T > 0$ only when $\gamma < 1$.

class exactpack.solvers.cog.cog12.**Cog12**(***kwargs*)
Computes the solution to the Cog12 problem.

Parameters

- **geometry** – 2=cylindrical, 3=spherical
- **rho0** – density coefficient
- **u0** – velocity coefficient
- **beta** – dimensionless constant β in Eq. (6.16)
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$
- **Gamma** – Gruneisen gas parameter

class exactpack.solvers.cog.cog12.**PlanarCog12**(***kwargs*)
The planar Cog12 problem.

Parameters

- **Gamma** – Gruneisen gas parameter
- **beta** – dimensionless constant β in Eq. (6.16)
- **rho0** – density coefficient
- **u0** – velocity coefficient
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog12.**CylindricalCog12**(***kwargs*)
The cylindrical Cog12 problem.

Parameters

- **Gamma** – Gruneisen gas parameter
- **beta** – dimensionless constant β in Eq. (6.16)
- **rho0** – density coefficient
- **u0** – velocity coefficient

- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog12.**SphericalCog12**(***kwargs*)
The spherical Cog12 problem.

Parameters

- **Gamma** – Gruneisen gas parameter
- **beta** – dimensionless constant β in Eq. (6.16)
- **rho0** – density coefficient
- **u0** – velocity coefficient
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

6.2.14 exactpack.solvers.cog.cog13

A Cog13 solver in Python.

This is a pure Python implementation of the Cog13 solution using Numpy. The exact solution takes the form,

$$\rho(r,t) = \rho_0\, r^{2/(\alpha-\beta-4)}\, t^{-2/(\alpha-\beta-4)-k-1}$$
$$u(r,t) = \frac{r}{t}$$
$$T(r,t) = T_0\, r^{-2/(\alpha-\beta-4)} \cdot t^{[\alpha(k+1)-k-2]/(\beta+3)+2(\alpha-1)/[(\beta+3)(\alpha-\beta-4)]}$$
$$T_0 = \left[\frac{3\Gamma}{4ca\lambda_0(\gamma-1)} \cdot \frac{\alpha-1+(\beta+3)(\gamma-1)}{\beta+3} \cdot \rho_0^{1-\alpha}\, \frac{\beta+4-\alpha}{2}\right]^{1/(\beta+3)}$$

Free parameters: α, β k, ρ_0, λ_0, and Γ.

class exactpack.solvers.cog.cog13.**Cog13**(***kwargs*)
Computes the solution to the Cog13 problem.

Parameters

- **geometry** – 1=planar, 2=cylindrical, 3=spherical
- **rho0** – density coefficient
- **beta** – dimensionless constant β in Eq. (6.16)
- **Gamma** – Gruneisen gas parameter
- **alpha** – dimensionless constant α in Eq. (6.16)
- **lambda0** – constant λ_0 in Eq. (6.16)
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog13.**PlanarCog13**(***kwargs*)
The planar Cog13 problem.

Parameters

- **rho0** – density coefficient
- **beta** – dimensionless constant β in Eq. (6.16)
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$
- **alpha** – dimensionless constant α in Eq. (6.16)
- **lambda0** – constant λ_0 in Eq. (6.16)

- **Gamma** – Gruneisen gas parameter

class exactpack.solvers.cog.cog13.**CylindricalCog13**(***kwargs*)
The cylindrical Cog13 problem. $\gamma = 5/3$.

Parameters

- **rho0** – density coefficient
- **beta** – dimensionless constant β in Eq. (6.16)
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$
- **alpha** – dimensionless constant α in Eq. (6.16)
- **lambda0** – constant λ_0 in Eq. (6.16)
- **Gamma** – Gruneisen gas parameter

class exactpack.solvers.cog.cog13.**SphericalCog13**(***kwargs*)
The spherical Cog13 problem.

Parameters

- **rho0** – density coefficient
- **beta** – dimensionless constant β in Eq. (6.16)
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$
- **alpha** – dimensionless constant α in Eq. (6.16)
- **lambda0** – constant λ_0 in Eq. (6.16)
- **Gamma** – Gruneisen gas parameter

6.2.15 exactpack.solvers.cog.cog14

A Cog14 solver in Python.

This is a pure Python implementation of the Cog14 solution using Numpy. The exact solution takes the form,

$$\rho(r,t) = \rho_0\, r^{-k-b}$$
$$u(r,t) = \left(\frac{\Gamma T_0 (k-b)}{b}\right)^{1/2} \cdot r^b$$
$$T(r,t) = T_0\, r^{2b}$$
$$b = \frac{k-1-\alpha k}{2+\alpha-2(\beta+4)}$$
$$T_0 = \left[\frac{b}{\Gamma(k-b)}\left(\frac{4ca\lambda_0}{3}\,\frac{\gamma-1}{\Gamma}\right)^2 \cdot \frac{16\,\rho_0^{2\alpha-2}\,b^4}{[2b+(\gamma-1)(k+b)]^2}\right]^{-1/(5+2\beta)}$$

Free parameters: α, β k, ρ_0, λ_0, and Γ.

class exactpack.solvers.cog.cog14.**Cog14**(***kwargs*)
Computes the solution to the Cog14 problem.

Parameters

- **geometry** – 1=planar, 2=cylindrical, 3=spherical
- **rho0** – density coefficient

- **beta** – dimensionless constant β in Eq. (6.16)
- **Gamma** – Gruneisen gas parameter
- **alpha** – dimensionless constant α in Eq. (6.16)
- **lambda0** – constant λ_0 in Eq. (6.16)
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog14.**PlanarCog14**(***kwargs*)
The planar Cog14 problem.

Parameters

- **rho0** – density coefficient
- **beta** – dimensionless constant β in Eq. (6.16)
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$
- **alpha** – dimensionless constant α in Eq. (6.16)
- **lambda0** – constant λ_0 in Eq. (6.16)
- **Gamma** – Gruneisen gas parameter

class exactpack.solvers.cog.cog14.**CylindricalCog14**(***kwargs*)
The cylindrical Cog14 problem.

Parameters

- **rho0** – density coefficient
- **beta** – dimensionless constant β in Eq. (6.16)
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$
- **alpha** – dimensionless constant α in Eq. (6.16)
- **lambda0** – constant λ_0 in Eq. (6.16)
- **Gamma** – Gruneisen gas parameter

class exactpack.solvers.cog.cog14.**SphericalCog14**(***kwargs*)
The spherical Cog14 problem.

Parameters

- **rho0** – density coefficient
- **beta** – dimensionless constant β in Eq. (6.16)
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$
- **alpha** – dimensionless constant α in Eq. (6.16)
- **lambda0** – constant λ_0 in Eq. (6.16)
- **Gamma** – Gruneisen gas parameter

6.2.16 exactpack.solvers.cog.cog16

A Cog16 solver in Python.

This is a pure Python implementation of the Cog16 solution using Numpy. The exact solution takes the form,

$$\rho(r,t) = \rho_0\, r^{-k-b}$$
$$u(r,t) = u_0\, r^b$$
$$T(r,t) = \frac{u_0^2 b}{\Gamma(k-b)} \cdot t^{2b}$$
$$\rho_0 = \left(\frac{16ca\lambda_0(\gamma-1)}{3}\right)^k \cdot \frac{b^{(5k-1)/2}}{u_0(k-b)^{(k-1)/2}\Gamma^{(3k-1)/2}\Big[2b+(\gamma-1)(k+b)\Big]^k}$$
$$\alpha = 1 - \frac{1}{k} \quad \beta = \frac{1}{2}\,\alpha - 3 \;\; (k \neq 0)$$

Free parameters: b, k and u_0, γ, λ_0, and Γ.

class exactpack.solvers.cog.cog16.**Cog16**(***kwargs*)
Computes the solution to the Cog16 problem.

Parameters

- **b** – dimensionless constant
- **geometry** – 2=cylindrical, 3=spherical
- **u0** – velocity coefficient
- **Gamma** – Gruneisen gas parameter
- **lambda0** – constant λ_0 in Eq. (6.16)
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog16.**CylindricalCog16**(***kwargs*)
The cylindrical Cog16 problem.

Parameters

- **Gamma** – Gruneisen gas parameter
- **b** – dimensionless constant
- **lambda0** – constant λ_0 in Eq. (6.16)
- **u0** – velocity coefficient
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog16.**SphericalCog16**(***kwargs*)
The spherical Cog16 problem.

Parameters

- **Gamma** – Gruneisen gas parameter
- **b** – dimensionless constant
- **lambda0** – constant λ_0 in Eq. (6.16)
- **u0** – velocity coefficient
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

6.2.17 exactpack.solvers.cog.cog17

A Cog17 solver in Python.

This is a pure Python implementation of the Cog17 solution using Numpy. The exact solution takes the form,

$$\rho(r,t) = \rho_0\, r^{(2\beta-4)/(1-\alpha)}\, t^{(2\beta+5)/(\alpha-1)}$$
$$u(r,t) = u_0\, \frac{r}{t}$$
$$T(r,t) = T_0\, \left(\frac{r}{t}\right)^2$$
$$u_0 = \frac{2\beta+5}{2\beta-4+(1-\alpha)(k+1)}$$
$$T_0 = \frac{(\alpha-1)(2\beta+5)}{\Gamma\Big[2\beta-4+(1-\alpha)(k+1)\Big]^2} \cdot \frac{9-(1-\alpha)(k+1)}{2\beta-4+2(1-\alpha)}$$
$$\rho_0 = \left[\frac{3\Gamma}{4ca\lambda_0(\gamma-1)}\, T_0^{-\beta-3} \cdot \frac{1}{2}\, \frac{-2+u_0\,[2+(\gamma-1)(k+1)]}{\alpha(2\beta-4)/(1-\alpha)+2\beta+k+7}\right]^{1/(\alpha-1)}$$

Free parameters: α, β, k, λ_0, and Γ.

class exactpack.solvers.cog.cog17.**Cog17**(***kwargs*)
Computes the solution to the Cog17 problem.

Parameters

- **geometry** – 1=planar, 2=cylindrical, 3=spherical
- **beta** – dimensionless constant β in Eq. (6.16)
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$
- **alpha** – dimensionless constant α in Eq. (6.16)
- **lambda0** – constant λ_0 in Eq. (6.16)
- **Gamma** – Gruneisen gas parameter

class exactpack.solvers.cog.cog17.**PlanarCog17**(***kwargs*)
The planar Cog17 problem.

Parameters

- **alpha** – dimensionless constant α in Eq. (6.16)
- **beta** – dimensionless constant β in Eq. (6.16)
- **lambda0** – constant λ_0 in Eq. (6.16)
- **Gamma** – Gruneisen gas parameter
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog17.**CylindricalCog17**(***kwargs*)
The cylindrical Cog17 problem.

Parameters

- **alpha** – dimensionless constant α in Eq. (6.16)
- **beta** – dimensionless constant β in Eq. (6.16)
- **lambda0** – constant λ_0 in Eq. (6.16)
- **Gamma** – Gruneisen gas parameter

- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog17.**SphericalCog17**(***kwargs*)
The spherical Cog17 problem.

Parameters

- **alpha** – dimensionless constant α in Eq. (6.16)
- **beta** – dimensionless constant β in Eq. (6.16)
- **lambda0** – constant λ_0 in Eq. (6.16)
- **Gamma** – Gruneisen gas parameter
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

6.2.18 exactpack.solvers.cog.cog18

A Cog18 solver in Python.

This is a pure Python implementation of the Cog18 solution using Numpy. The exact solution takes the form,

$$\rho(r,t) = \rho_0\, r^{-(2\beta+k+7)/\alpha} \cdot \left(\tau^2 - t^2\right)^{-(k+1)/2+(2\beta+k+7)/2\alpha}$$
$$u(r,t) = -\frac{rt}{\tau^2 - t^2}$$
$$T(r,t) = \frac{\alpha\tau^2}{\Gamma(2\alpha - 2\beta - k - 7)} \cdot \frac{r^2}{(\tau^2 - t^2)^2}$$

with

$$\gamma = \frac{k+3}{k+1}\,.$$

Free parameters: α, β, k, τ, ρ_0, and Γ.

class exactpack.solvers.cog.cog18.**Cog18**(***kwargs*)
Computes the solution to the Cog18 problem. Note: choose alpha, beta, tau correctly.

Parameters

- **tau** – free parameter of dimension time
- **geometry** – 1=planar, 2=cylindrical, 3=spherical
- **rho0** – density coefficient
- **beta** – dimensionless constant β in Eq. (6.16)
- **alpha** – dimensionless constant α in Eq. (6.16)
- **Gamma** – Gruneisen gas parameter

class exactpack.solvers.cog.cog18.**PlanarCog18**(***kwargs*)
The planar Cog18 problem.

Parameters

- **alpha** – dimensionless constant α in Eq. (6.16)
- **beta** – dimensionless constant β in Eq. (6.16)
- **rho0** – density coefficient
- **Gamma** – Gruneisen gas parameter

- **tau** – free parameter of dimension time

class exactpack.solvers.cog.cog18.**CylindricalCog18**(***kwargs*)
The cylindrical Cog18 problem.

Parameters

- **alpha** – dimensionless constant α in Eq. (6.16)
- **beta** – dimensionless constant β in Eq. (6.16)
- **rho0** – density coefficient
- **Gamma** – Gruneisen gas parameter
- **tau** – free parameter of dimension time

class exactpack.solvers.cog.cog18.**SphericalCog18**(***kwargs*)
The spherical Cog18 problem.

Parameters

- **alpha** – dimensionless constant α in Eq. (6.16)
- **beta** – dimensionless constant β in Eq. (6.16)
- **rho0** – density coefficient
- **Gamma** – Gruneisen gas parameter
- **tau** – free parameter of dimension time

6.2.19 exactpack.solvers.cog.cog19

A Cog19 solver in Python.

This is a pure Python implementation of the Cog19 solution using Numpy. The solution contains a shock located at

$$R(t) = -\frac{1}{2}(\gamma - 1)\, u_0 t$$

Region 1:

$$\rho(r,t) = \rho_0 \left(\frac{\gamma+1}{\gamma-1}\right)^{k+1}$$
$$u(r,t) = 0$$
$$T(r,t) = \frac{u_0^2\,(\gamma-1)}{2\Gamma}$$

Region 2:

$$\rho(r,t) = \rho_0 \left(\frac{r - u_0 t}{r}\right)^{k}$$
$$u(r,t) = u_0$$
$$T(r,t) = 0$$

Free parameters: k, ρ_0, and u_0 (with $u_0 < 0$), γ, and Γ.

class exactpack.solvers.cog.cog19.**Cog19**(***kwargs*)
Computes the solution to the Cog19 problem. No conduction.

Parameters

- **geometry** – 1=planar, 2=cylindrical, 3=spherical
- **rho0** – density coefficient
- **Gamma** – Gruneisen gas parameter
- **u0** – velocity coefficient
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog19.**PlanarCog19**(***kwargs*)
The planar Cog19 problem.

Parameters

- **Gamma** – Gruneisen gas parameter
- **rho0** – density coefficient
- **u0** – velocity coefficient
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog19.**CylindricalCog19**(***kwargs*)
The cylindrical Cog19 problem.

Parameters

- **Gamma** – Gruneisen gas parameter
- **rho0** – density coefficient
- **u0** – velocity coefficient
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog19.**SphericalCog19**(***kwargs*)
The spherical Cog19 problem.

Parameters

- **Gamma** – Gruneisen gas parameter
- **rho0** – density coefficient
- **u0** – velocity coefficient
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

6.2.20 exactpack.solvers.cog.cog20

A Cog20 solver in Python.

This is a pure Python implementation of the Cog20 solution using Numpy. The solution contains a shock located at

$$R_{\text{shock}} = \frac{u_0(\gamma-1)}{4a}\,\frac{t(1-2at)}{1-at}$$

Region 1:

$$\rho(r,t) = \rho_0\left(\frac{\gamma+1}{\gamma-1}\right)^{k+1}\Big(1-at\Big)^{-k-1}$$
$$u(r,t) = -\frac{ar}{1-at}$$
$$T(r,t) = \frac{u_0^2(\gamma-1)}{2\Gamma}\Big(1-at\Big)^{-2}$$

Region 2:

$$\rho(r,t) = \rho_0\Big(1-at\Big)^{-k-1}\left(\frac{r-u_0 t}{r}\right)^k$$
$$u(r,t) = \frac{u_0 - ar}{1-at}$$
$$T(r,t) = 0$$

Free parameters: a, k, u_0, and ρ_0, γ, and Γ.

class exactpack.solvers.cog.cog20.**Cog20**(***kwargs*)
Computes the solution to the Cog20 problem. No conduction.

Parameters

- **a** – free parameter with dimensions of inverse time
- **geometry** – 1=planar, 2=cylindrical, 3=spherical
- **rho0** – density coefficient
- **u0** – velocity coefficient
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$
- **Gamma** – Gruneisen gas parameter

class exactpack.solvers.cog.cog20.**PlanarCog20**(***kwargs*)
The planar Cog20 problem.

Parameters

- **a** – free parameter with dimensions of inverse time
- **rho0** – density coefficient
- **Gamma** – Gruneisen gas parameter
- **u0** – velocity coefficient
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog20.**CylindricalCog20**(***kwargs*)
The cylindrical Cog20 problem.

Parameters

- **a** – free parameter with dimensions of inverse time
- **rho0** – density coefficient
- **Gamma** – Gruneisen gas parameter
- **u0** – velocity coefficient
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.cog.cog20.**SphericalCog20**(***kwargs*)
The spherical Cog20 problem.

Parameters

- **a** – free parameter with dimensions of inverse time
- **rho0** – density coefficient
- **Gamma** – Gruneisen gas parameter

- **u0** – velocity coefficient
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

6.2.21 exactpack.solvers.cog.cog21

A Cog21 solver in Python.

This is a pure Python implementation of the Cog21 solution using Numpy. The solution contains a shock located at

$$R_{\text{shock}} = \frac{2}{\Gamma T_0\, t^2} \text{ with}$$
$$k = 2 \text{ and } \gamma = 5\ .$$

Region 1:

$$\rho(r,t) = \frac{3}{2}\,\rho_0 r^{-3}$$
$$u(r,t) = 0$$
$$T(r,t) = T_0\, r^3$$

Region 2:

$$\rho(r,t) = \rho_0\, r^{-3}$$
$$u(r,t) = \frac{r}{t}$$
$$T(r,t) = 0$$

Free parameters: ρ_0, T_0, and Γ.

class exactpack.solvers.cog.cog21.**Cog21**(***kwargs*)

Computes the solution to the Cog21 problem. No conduction.

Parameters

- **rho0** – density coefficient
- **temp0** – temperature coefficient
- **Gamma** – Gruneisen gas parameter

6.3 The DSD Problems

Python based solvers for HE burn time for DSD verification problems.

This suite of solvers calculates HE burn times for DSD verification problems.

The DSD High Explosive Problem Set is a series of three problems designed to test the burn table solution (HE light times) generated by DSD level-set solvers for burn simulations. The Cylindrical Expansion test problem has an exact burn time solution in 2D *[Bdzil]*. The 2D Rate Stick and Explosive Arc test problems can each be reduced to a quasi-linear PDE, which can be solved using highly-accurate numerical methods. That solution must then be inverted to obtain the burn time solution on the specified grid.

DSD theory uses boundary angles to describe how the HE combustion wave (burn front) interacts with surrounding materials. The boundary angle, ω_c, is defined as the angle between the normal to the HE boundary and the normal to the burn front shock wave, as shown in the figure.

Three angles are considered:

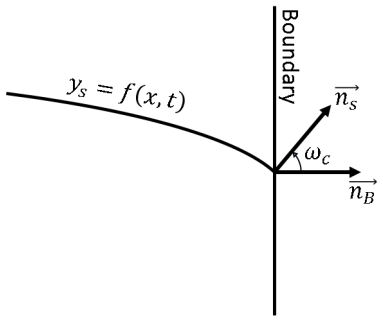

Fig. 6.1: The boundary angle ω_c as defined by the normals to the burn front, $\vec{n_s}$, and to the HE boundary, $\vec{n_B}$.

- The maximum possible angle is found in the case of a reflective boundary or an abutting rigid inert body. Here, the burn front is expected to be perpendicular to the boundary, resulting in a maximum edge angle of $\omega = \frac{\pi}{2}$. This holds for any HE.
- The minimum angle, ω_s, is found in the case of HE product expansion into a vacuum. The value of this angle will depend on which HE material is being used. For all HEs, $0 < \omega_s < \frac{\pi}{2}$.
- The edge angle, ω_c, between the HE and a deformable inert material depends on which HE material and which inert material are in use. For all combinations, $\omega_s < \omega_c < \frac{\pi}{2}$.

Which, if any, of these angles need to be defined depends on the test problem being evaluated.

6.3.1 exactpack.solvers.dsd.ratestick

A DSD Rate Stick solver in Python.

This is a pure Python implementation of the Rate Stick solution using Numpy.

The DSD Rate Stick problem is used to test how well a level set solver applies boundary conditions to an HE burn time table calculation, as the solution to this problem is dominated by the applied boundary conditions. In this problem, the boundaries of the explosive align with the DSD computational mesh.

In a cylindrical configuration, a rate stick consists of a right circular cylinder of high explosive (HE) confined by a tube of inert material. A solution can also be defined for a planar inert-HE-inert sandwich configuration.

The cylindrical (**geometry** $= 2$) configuration is defined by the radius of the HE cylinder, R, and the edge angle, ω_c, between the HE and the tube material. The system is assumed to be centered on the z-axis, so that $0 \leq r \leq R$. In the following equations, x is used to represent the r direction and y is used to represent the z direction.

The planar (**geometry** $= 1$) sandwich configuration is assumed to be centered on the y-axis. Thus, the problem is defined by the half thickness of the HE slab, R, and the edge angle, ω_c, between the HE and the inert material on either side. It is assumed that the same material is used on both sides of the HE. While the HE exists in the region $-R \leq x \leq R$, only the right half of this domain will be modeled ($0 \leq x \leq R$).

At time $t_d = 0.0$, the shape of the HE burn front shock wave is prescribed. The shape will be determined by the choice of initial condition, as described below. Any HE behind the initial burn front is assumed to ignite at time t_d, as if it were part of the detonation system.

The detonation trajectory for the above conditions is given by

$$y = f(x,t)$$

where the function f satisfies the PDE

$$f_t = D_n\sqrt{1 + (f_x)^2}$$

where D_n is the velocity of the HE detonation wave in the shock-normal direction, which is described by a deviation from the nominal constant Chapman-Jouguet detonation shock speed D_{CJ} of the HE.

In *[Bdzil]*, the deviation in D_n is linear with respect to the curvature, κ, of the detonation shock front:

$$D_n = D_{CJ} - \alpha\kappa$$

With this form of D_n and the geometrically appropriate curvatures, the function f satisfies the PDE

$$f_t - D_{CJ}\sqrt{1 + (f_x)^2} = \alpha\frac{f_{xx}}{1 + (f_x)^2} + \alpha n\frac{f_x}{x}$$

where $n =$ **geometry** -1. As currently implemented, κ will be calculated separately, then D_n will be calculated from this linear form. The function f will then be calculated from the more general form above. In this way, any other form for D_n could be added at a later date.

In the cylindrical case, the boundary conditions to be applied are

$$f_x(0, t) = 0$$
$$f_x(R, t) = \cot(\omega_c)$$

In the planar case, the nominal boundary conditions are

$$f_x(-R, t) = \cot(\omega_c)$$
$$f_x(R, t) = -\cot(\omega_c)$$

Because only the right half of the planar case is modeled, the actual boundary conditions to be applied are the same as the cylindrical case.

Three initial conditions have been implemented in the solver. In all three cases, the outer edge ($x = R$) of the burn front shock wave is assumed to be located at $y = 0$ at time t_d. The desired initial condition can be indicated through the IC parameter.

- Case $IC = 1$

  The HE is initiated on a circular detonation arc with radius r_d, which is centered on the y-axis. The center of the detonation is located at $(0, y_d)$, where

  $$y_d = -\sqrt{{r_d}^2 - R^2}$$

  The location of the burn front at $t_d = 0.0$, which provides the initial condition for the PDE, is thus found to be:

  $$f(x, 0) = y_d + \sqrt{{r_d}^2 - x^2}$$

  This case represents the physical situation that occurs when an expanding spherical detonation front reaches the bottom of the containment for the HE rate stick.

  The initial angle between the normal to the shock wave and the normal to the confinement boundary cannot exceed the applied edge angle. If this condition is violated, application of the edge angle results in a negative curvature, which is not allowed in level set theory. Thus, the radius of the detonation front, r_d, and the radius/thickness of the HE, R, must satisfy the condition

  $$r_d >= \frac{R}{\omega_c}.$$

  Note: This initial condition is slightly different than as described in *[Bdzil]*. The location of the front of the detonation (therefore, the center of the detonation) has been moved by 1 unit in the negative y-direction.

- Case $IC = 2$

  A circular arc for detonation initiation that is compatible with the boundary conditions above has a detonation radius of

  $$r_d = \frac{R}{\cos(\omega_c)}$$

  which clearly satisfies the initial angle condition stated in the previous case. The center of the detonation is then located at $(0, y_d)$, where

  $$y_d = -\sqrt{{r_d}^2 - R^2}$$

  Any value input for r_d will be overridden by the calculated value. Using this calculated value, the initial condition for the PDE is given by

  $$f(x, 0) = y_d + \sqrt{{r_d}^2 - x^2}$$

- Case $IC = 3$

  Under experimental conditions, the HE can also be initiated by a planar detonation wave, which has an initial condition

  $$f(x, 0) = 0.0$$

  This condition is equivalent to an infinite detonation radius and satisfies the initial angle condition stated in the first case. Any value input for r_d will be ignored in the calculation.

class exactpack.solvers.dsd.ratestick.**RateStick**(***kwargs*)

Computes the numerical solution to the Rate Stick Problem.

For the planar case (**geometry** $= 1$), the HE slab is assumed to be centered on the y-axis. The right half is modeled in the xy-plane. For the cylindrical case (**geometry** $= 2$), the HE cylinder is assumed to be centered on the z-axis. It is modeled in the rz-plane. Note that the *[Bdzil]* paper uses n to define the coordinate system, where $n =$ **geometry** -1. The edge angle, ω_c is assumed to satisfy $0 < \omega_c < \frac{\pi}{2}$.

The nominal detonation velocity of the HE, D_{CJ}, must be positive. The linear coefficient, α, of detonation velocity deviance must also be positive.

Detonation time is assumed to be $t_d = 0.0$. The burn front shock wave is assumed to have just reached the confinement material at detonation time, with the shock wave located at $y = 0$ at the outer edge of the HE ($x = R$) before application of the boundary condition.

The initial angle between the normal to the shock wave and the normal to the confinement boundary is assumed to be no more than the applied edge angle. Thus, if $IC = 1$, the radius of the detonation front, r_d, and the radius/thickness of the HE, R, must satisfy the condition $r_d >= \frac{R}{\cos(\omega_c)}$. This condition is automatically satisfied by the other two initial conditions.

Default values are selected to reflect the description in *[Bdzil]*, with the caveat listed in the first initial condition above. Default values are **geometry** $= 1$, $R = 1.0$, $\omega_c = \frac{\pi}{4}$, $D_{CJ} = 1.0$, $\alpha = 0.1$, $IC = 1$ and $r_d = \sqrt{626}$, which produces a detonator location of $(0.0, -25.0)$.

The solution for y of the burn front is calculated on a fine (x, t) mesh to obtain $y = y(x, t)$. The solution is then interpolated to the requested xy mesh and inverted to obtain the burn time as $t = t(x, y)$. The requested mesh must have a larger mesh spacing than the mesh used for the calculation. For efficiency in inverting the solution, the user must input the number of nodes in the x-direction, $xnodes$, and the number of nodes in the y-direction, $ynodes$ in the requested mesh.

Parameters

- **D_CJ** – nominal detonation velocity of the HE

- **omega_c** – DSD edge angle between HE and inert
- **R** – radius of HE cylinder or half-thickness of HE slab
- **xnodes** – number of nodes in x-direction
- **geometry** – 1=planar, 2=cylindrical
- **alpha** – coefficient of linear detonation velocity deviance
- **IC** – initial condition (see descriptions)
- **r_d** – initial detonation front radius
- **t_f** – final time
- **ynodes** – number of nodes in y-direction

Input evaluation points as an N x 2 2D array of positions: [$[x_0, y_0]$, $[x_1, y_1]$, ..., $[x_N, y_N]$].

A time value must be input in order to use the ExactPack machinery. The time value is ignored in the calculation.

6.3.2 exactpack.solvers.dsd.cylexpansion

A DSD Cylindrical Expansion solver in Python.

This is a pure Python implementation of the Cylindrical Expansion solution using Numpy.

The DSD Cylindrical Expansion problem is used to test a code's algorithm for updating the solution for the level-set equation. It is insensitive to any issues there may be with boundary conditions, as the only boundaries are parallel to level sets of the governing equation. It also tests the level-set solution algorithm for a multiple HE region.

A cylindrical HE tube with inner radius r_1 and outer radius r_2 is surrounded by an adjoining cylindrical tube of a second HE. Thus, the second HE tube has an inner radius of r_2 and extends indefinitely in the radial direction. At time t_d, the inner HE is initiated by a circle detonator of radius r_1. Without loss of generality, the entire system is assumed to be centered at the origin.

The only internal boundary in the complete system (between the HE materials) is parallel to the burn front. Thus, none of the boundary angles need to be defined for this solver.

For each HE material, the velocity of the detonation wave in the shock-normal direction, D_n, is described by a linear deviation from the nominal constant Chapman-Jouguet detonation shock speed D_{CJ}. The deviation depends on the curvature, κ, of the detonation shock front. For the inner HE material,

$$D_{n_1} = D_{CJ_1} - \alpha_1 \kappa$$

For the outer HE material,

$$D_{n_2} = D_{CJ_2} - \alpha_2 \kappa$$

Under the linear detonation velocity condition and the cylindrical HE configuration given above, the detonation trajectory is described by

$$\frac{dr}{dt} = D_{CJ} - \alpha\kappa = D_{CJ} - \frac{\alpha}{r}$$

The solution to the 1D ODE in each material is given by

$$D_{CJ_i}(t - t_{0_i}) = (r - r_{0_i}) + \frac{\alpha_i}{D_{CJ_i}} ln\left(\frac{r - \frac{\alpha_i}{D_{CJ_i}}}{r_{0_i} - \frac{\alpha_i}{D_{CJ_i}}}\right), r > \frac{\alpha_i}{D_{CJ_i}}, i = 1, 2$$

where (t_{0_i}, r_{0_i}) are the time and radius when detonation is initiated in each of the explosives.

class exactpack.solvers.dsd.cylexpansion.**CylindricalExpansion**(***kwargs*)

Computes the numerical solution to the Cylindrical Expansion Problem.

The HE regions are assumed to be two concentric cylindrical tubes centered at the origin, modeled in the $r\theta$-plane. All radii are assumed to be positive and large enough to avoid the singularity at the origin, i.e. $r_1 > \frac{\alpha_1}{D_{CJ_1}}$ and $r_2 > \frac{\alpha_2}{D_{CJ_2}}$. No boundary angles are necessary.

The nominal detonation velocities of both HEs, D_{CJ_i}, must be positive. The linear coefficients, α_i, of detonation velocity deviance must also be positive.

Default values are selected to be consistent with the problem definition in *[Bdzil]*. Default values are **geometry** $= 2$, $r_1 = 1.0$, $r_2 = 2.0$, $D_{CJ_1} = 0.5$, $D_{CJ_2} = 1.0$, $\alpha_1 = 0.1$, $\alpha_2 = 0.1$ and $t_d = 0.0$.

Parameters

- **alpha_2** – linear detonation velocity deviance coefficient for HE2
- **t_d** – initial detonation time
- **alpha_1** – linear detonation velocity deviance coefficient for HE1
- **geometry** – 2=cylindrical
- **r_1** – inner radius of HE1
- **r_2** – radius of interface between HE1 and HE2
- **D_CJ_2** – nominal detonation velocity of outer HE
- **D_CJ_1** – nominal detonation velocity of inner HE

Input evaluation points as an N x 2 2D array of positions: [$[x_0, y_0]$, $[x_1, y_1]$, ..., $[x_N, y_N]$].

A time value must be input in order to use the ExactPack machinery. The time value is ignored in the calculation.

6.3.3 exactpack.solvers.dsd.explosivearc

A DSD Explosive Arc solver in Python.

This is a pure Python implementation of the Explosive Arc solution using Numpy.

The DSD Explosive Arc problem is used to test how well a level set solver applies boundary conditions to an HE burn time table calculation, as the solution to this problem is dominated by the applied boundary conditions. In this problem, the boundaries of the explosive do not align with the DSD computational mesh.

An explosive arc consists of a semi-annulus of high explosive (HE) which occupies the region $r_1 <= r <= r_2, -\frac{\pi}{2} <= \theta <= \frac{\pi}{2}$. The problem is solved in an $r\theta$-coordinate system. An edge angle is defined at each of the radial boundaries. The free boundary found at the inner radius, r_1, is defined by the HE sonic angle, ω_s. The boundary at the outer radius, r_2, can be defined by either the HE confinement angle, ω_c, if a confinement material is present or by a fixed or reflective angle, $\omega_e = \frac{\pi}{2}$, if HE is present outside the modeled region.

At time $t_d = 0.0$, the shape of the HE burn front shock wave is prescribed. Any HE behind the initial burn front is assumed to ignite at time t_d, as if it were part of the detonation system. The detonator is located at (x_d, y_d) where y_d is at the center of the lower end of the annulus, i.e.,

$$y_d = -\frac{r_1 + r_2}{2},$$

The detonator is located outside of the annulus, so the x-coordinate of the detonator position is negative. The inital burn front is then located at

$$(x - x_d)^2 + (y - y_d)^2 = {r_d}^2$$

where r_d is the radius of the initial detonation front, found by

$$r_d = \sqrt{|x_d|^2 + \left(\frac{r_2 - r_1}{2}\right)^2}$$

The detonation trajectory for the above conditions is given by

$$\theta = f(r, t)$$

where the function f satisfies the PDE

$$f_t = D_n \sqrt{1 + (r f_r)^2}$$

where D_n is the velocity of the HE detonation wave in the shock-normal direction, which is described by a deviation from the nominal constant Chapman-Jouguet detonation shock speed D_{CJ} of the HE.

In *[Bdzil]*, the deviation in D_n is linear with respect to the curvature, κ, of the detonation shock front:

$$D_n = D_{CJ} - \alpha\kappa$$

With this form of D_n and the geometrically appropriate curvatures, the function f satisfies the PDE

$$f_t = -\frac{D_{CJ}}{r}\sqrt{1 + (r f_r)^2} + \frac{\alpha}{r}\frac{r f_{rr} + r^2 {f_r}^3 + 2 f_r}{1 + (r f_r)^2}$$

As currently implemented, κ will be calculated separately, then D_n will be calculated from this linear form. The function f will then be calculated from the more general form above. In this way, any other form for D_n could be added at a later date.

The boundary conditions to be applied are

$$r_1 f_r(r_1, t) = \cot(\omega_s)$$
$$r_2 f_r(r_2, t) = -\cot(\omega_e)$$

class exactpack.solvers.dsd.explosivearc.**ExplosiveArc**(***kwargs*)

Computes the numerical solution to the Explosive Arc Problem.

The HE semi-annulus is assumed to lie in the region $r_1 <= r <= r_2, -\frac{\pi}{2} <= \theta <= \frac{\pi}{2}$. The half-thickness of the annulus is then $R = \frac{r_2 - r_1}{2}$. The solution is obtained in the $r\theta$-plane.

The boundary at the inner radius, $r = r_1$ is assumed to be free, thus the DSD edge angle there is the HE sonic angle and $\omega_{in} = \omega_s$, where the sonic angle is assumed to satisfy $0 < \omega_s < \frac{\pi}{2}$. The boundary at the outer radius, $r = r_2$ is either confined by a material, in which case the DSD edge angle is the confinement angle and $\omega_{out} = \omega_c$, or a fixed boundary, in which case the DSD edge angle is $\omega_{out} = \frac{\pi}{2}$. For the confined case, ω_c is assumed to satisfy $\omega_s < \omega_c < \frac{\pi}{2}$.

The detonator is located at (x_d, y_d) where y_d is assumed to be in the center of the lower end of the annulus, i.e., $y_d = -\frac{r_1 + r_2}{2}$. The detonator is assumed to be located outside of the annulus, so the x-coordinate of the detonator position, x_d, is assumed to be negative. The nominal detonation velocity of the HE, D_{CJ}, must be positive. The linear coefficient, α, of detonation velocity deviance must also be positive.

Detonation time is assumed to be $t_d = 0.0$. The burn front shock wave is assumed to have just reached the confinement material at detonation time, with the shock wave located at $\theta = -\frac{pi}{2}$ at the edges of the annulus ($r = r_1$ and $r = r_2$) before application of the boundary conditions.

Default values are selected to reflect the description in *[Bdzil]*. Default values are **geometry** $= 1$, $r_1 = 2.0$, $r_2 = 4.0$, $\omega_{in} = \frac{\pi}{4}$, $\omega_{out} = \frac{\pi}{2}$, $x_d = -4.0$, $D_{CJ} = 1.0$ and $\alpha = 0.1$. This produces a detonator location of $(-4.0, -3.0)$ and an initial detonation radius of $r_d = \sqrt{17.0}$.

The incoming Cartesian (xy) mesh is converted to a polar ($r\theta$) mesh. The solution for θ of the burn front is calculated on a fine (r, t) mesh to obtain $\theta = \theta(r, t)$. The solution is then interpolated to the requested $r\theta$ mesh and inverted to obtain the burn time as $t = t(r, \theta)$. The requested mesh must have a larger mesh spacing than the mesh used for the calculation. For efficiency in inverting the solution, the user must input the number of nodes in the x-direction, $xnodes$, and the number of nodes in the y-direction, $ynodes$ in the requested mesh.

Parameters

- **x_d** – x-coordinate of detonator location
- **ynodes** – number of nodes in y-direction
- **alpha** – coefficient of linear detonation velocity deviance
- **omega_in** – inner radius DSD free-surface angle
- **omega_out** – outer radius DSD edge angle
- **D_CJ** – nominal detonation velocity of the HE
- **xnodes** – number of nodes in x-direction
- **geometry** – 1=planar
- **r_1** – inner radius of HE arc
- **r_2** – outer radius of HE arc
- **t_f** – final time

Input evaluation points as an N x 2 2D array of positions: [[x_0, y_0], [x_1, y_1], ..., [x_N, y_N]].

A time value must be input in order to use the ExactPack machinery. The time value is ignored in the calculation.

6.4 The Escape of HE Products Problem

The Escape of HE Products problem.

The escape of HE products (EHEP) problem was first published by Fickett and Rivard in 1974 *[Fickett1974]*. In 2002, *[Dykema2002]* published a derivation of the characteristics of the exact solution in x-t space. A complete description of the problem, the exact solution equations, and the solution algorithm is presented in *[Doebling2015]*. The problem concerns a planar one-dimensional rod of HE extending from the origin for a length of $\tilde{x}$, as shown in the Figure.

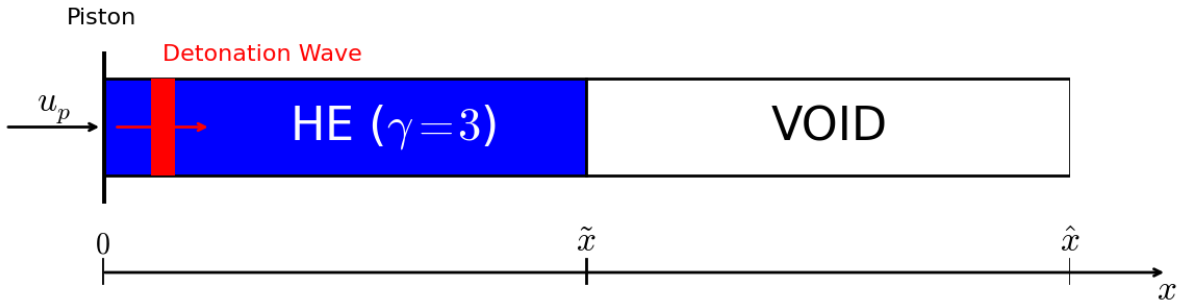

Fig. 6.2: Diagram of the EHEP problem

The HE is a polytropic ideal gas with adiabatic index $\gamma = 3$ and C-J detonation velocity $D_{\rm CJ}$. (The value $\gamma = 3$ is required to enable the derivation of the exact solution.) The unreacted HE and the reacted HE are assumed to have the same material properties. To the left of the HE is a piston moving in the +x direction with velocity u_p. To the right of the HE is a void. At $t = 0$, the detonation wave departs from the origin in the +x direction, and the piston begins to move and isentropically compresses the reaction products. When the detonation wave reaches $\tilde{x}$ at time $\tilde{t}$, the HE has all been consumed, and the material begins to expand isentropically into the void region. At the same time $\tilde{t}$, the arrival of the detonation wave at an interface with lesser impedence to the right causes a rarefaction wave to be propagated from $\tilde{x}$ in the -x direction. The independent fluid variables are (i) the gas density $\rho(x,t)$, (ii) the velocity of the gas $u(x,t)$, and (iii) the pressure $p(x,t)$, each at spatial location x and time t. The specific internal energy $e(r,t)$ is related to the other fluid variables by the equation of state (EOS) for an ideal gas at constant entropy,

$$p = (\gamma - 1)\rho e \ ,$$

where $\gamma \equiv c_p/c_v$ is the ratio of specific heats at constant pressure and volume. The choice of $\gamma = 3$ produces solution characteristics that are straight lines in the x-t plane, and the values for the physical variables are then derived for each region defined by the characteristics.

To find the solution at a given value of (x,t), we must first determine the region in which the point lies, and then use the corresponding Euler equations for $\rho(x,t), u(x,t), p(x,t), e(x,t)$ in that region. The easiest way to describe the regions is to define the coordinates of the vertices of the polygon that bounds each region. To close the polygons, we must select a maximum distance $x_{\max}$ and maximum time $t_{\max}$. For each region, the vertices that describe its polygon are listed here:

Region 0V (ahead of the shock in the void region):

$$\begin{aligned}
(\tilde{x}, 0) \\
(\tilde{x}, \tilde{t}) \\
(x_{\max}, \frac{x_{\max}}{D}) \\
(x_{\max}, 0) \\
c_s = 0 \\
u = 0 \\
p = 0 \\
\rho = 0
\end{aligned}$$

Region 0H (ahead of the shock within the HE):

$$\begin{aligned}
(0, 0) \\
(\tilde{x}, \tilde{t}) \\
(\tilde{x}, 0) \\
c_s = 0 \\
u = 0 \\
p = 0 \\
\rho = \rho_0
\end{aligned}$$

For Regions I through V, p and ρ are given as functions of c_s:

$$\begin{aligned}
p &= \frac{16}{27}\rho_0 D^2 \left(\frac{c_s}{D}\right)^3 \\
\rho &= \frac{16}{9}\rho_0 \left(\frac{cs}{D}\right)
\end{aligned}$$

Region I (behind the detonation wave):

$$(0,0)$$
$$\frac{3\tilde{x}}{2}\left(1-\frac{D}{4u_p+2D},\frac{1}{2u_p+D}\right)$$
$$(\tilde{x},\tilde{t})$$
$$c_s=\frac{1}{2}\left(\frac{x}{t}+\frac{D}{2}\right)$$
$$u=\frac{1}{2}\left(\frac{x}{t}-\frac{D}{2}\right)$$

Region II (isentropic expansion into the void)

$$(\tilde{x},\tilde{t})$$
$$\frac{3\tilde{x}}{2}\left(1-\frac{D}{4u_p+2D},\frac{1}{2u_p+D}\right)$$
$$((2u_p+D/2)t_{\max},t_{\max})$$
$$(x_{\max},\frac{x_{\max}}{D})$$
$$c_s=\frac{1}{2}\left(\frac{x}{t}-\frac{x-\tilde{x}}{t-\tilde{t}}\right)$$
$$u=\frac{1}{2}\left(\frac{x}{t}+\frac{x-\tilde{x}}{t-\tilde{t}}\right)$$

Region III (fluid pushed by piston)

$$(0,0)$$
$$\frac{3\tilde{x}}{2}\left(1-\frac{D}{4u_p+2D},\frac{1}{2u_p+D}\right)$$
$$\frac{3\tilde{x}}{2u_p+D}(1,u_p)$$
$$c_s=u_p+\frac{D}{2}$$
$$u=u_p$$

Region IV (fluid expansion and rarefaction)

$$\frac{3\tilde{x}}{2}\left(1-\frac{D}{4u_p+2D},\frac{1}{2u_p+D}\right)$$
$$\frac{3\tilde{x}}{2u_p+D}(1,u_p)$$
$$\left(-\frac{3}{2}\tilde{x}+(2u_p+\frac{D}{2})t_{\max},t_{\max}\right)$$
$$\left(x_{\max},\frac{x_{\max}}{2u_p+D/2}\right)$$
$$c_s=u_p+\frac{D}{2}\left(\frac{1}{2}-\frac{x-\tilde{x}}{Dt-\tilde{x}}\right)$$
$$u=u_p+\frac{D}{2}\left(\frac{1}{2}+\frac{x-\tilde{x}}{Dt-\tilde{x}}\right)$$

Region V (rarefaction interaction with piston)

$$\frac{3\tilde{x}}{2u_p + D}(1, u_p)$$
$$(u_p t_{\max}, t_{\max})$$
$$\left(-\frac{3}{2}\tilde{x} + (2u_p + \frac{D}{2})t_{\max}, t_{\max}\right)$$
$$c_s = \frac{(D - u_p)\tilde{t}}{t - \tilde{t}}$$
$$u = \frac{x - u_p\tilde{t}}{t - \tilde{t}}$$

Region 00 (behind the piston)

$$(0, 0)$$
$$(0, t_{\max})$$
$$(u_p t_{\max}, t_{\max})$$

6.4.1 exactpack.solvers.ehep.ehep

Exact solution for the Escape of HE Products problem

class exactpack.solvers.ehep.ehep.**EscapeOfHEProducts**(***kwargs*)

Computes the solution to the Escape of HE Products problem. For the complete problem definition and exact solution, see *[Doebling2015]*. The problem default values are selected to be consistent with the original problem definition in Fickett and Rivard *[Fickett1974]*.

Default values are: $D = 0.85\,\mathrm{cm}/\mu\mathrm{sec}, \rho = 1.6, u_p = 0.5, \tilde{x} = 1.0, x_{\max} = 10., t_{\max} = 10.$

Parameters

- **D** – Detonation velocity of the HE (cm/us)
- **xmax** – maximum value of x allowed for exact solution
- **geometry** – 1=axial
- **tmax** – maximum value of t allowed for exact solution
- **xtilde** – width of the HE material (cm)
- **up** – Velocity of the piston (cm/us)
- **gamma** – adiabatic index, must be 3.0
- **rho_0** – Initial density of the HE (g/cc)

Set default values if necessary, check for valid inputs, and compute polygon vertices for the parameter values.

p_rho(*rho_0*, *cs*, *D*)

Compute shocked pressure and Density given initial density, sound speed and shock speed

point_on_boundary(*corners*, *point*, *tol=1e-12*)

Determine whether a given point lies on the boundary of the polygon defined by "corners"

Algorithm:

```
on_boundary = False
Loop over each successive pair of corners
 (including last to first)
```

```
For each pair of corners:
       Call point_on_line
       If True, assign to on_boundary
```

point_on_line(*corners*, *point*, *tol=1e-05*)

Corners is a list of 2 points

Algorithm:

```
Compute distance between point and first corner
Compute distance between point and second corner
Compute distance between the two corners
If magnitude of the sum of the first two distances minus the
   third distance is less than a tolerance, then the point is
   on the line
```

6.5 The Guderley Problem

The Guderley Problem.

In 1942, G. Guderley *[Guderley1942]* found a semi-analytic solution for a self-similar convergent shock wave in an invisid, non-heat conducting polytropic gas. The Euler equations for this problem take the form *[Guderley2012]*

$$\frac{\partial \rho}{\partial t} + \frac{\partial (u\rho)}{\partial r} + (k-1)\,\frac{u\rho}{r} = 0$$
$$\frac{\partial u}{\partial t} + u\,\frac{\partial u}{\partial r} + \frac{1}{\rho}\frac{\partial p}{\partial r} = 0$$
$$\frac{\partial}{\partial t}\left(p\rho^{-\gamma}\right) + u\,\frac{\partial}{\partial r}\left(p\rho^{-\gamma}\right) = 0\,,$$

where $k = 1, 2, 3$ for planar, cylindrical, and spherical coordinates respectively. The independent fluid variables are (i) the gas density $\rho(r,t)$, (ii) the velocity of the gas $u(r,t)$, and (iii) the pressure $p(r,t)$, where the physical quantities are specified at at given radius r and time t. The specific internal energy $e(r,t)$ is related to other fluid variables by an equation of state (EOS). For an ideal gas at constant entropy, the EOS takes the form

$$p = (\gamma - 1)\rho e\,,$$

where $\gamma \equiv c_p/c_v$ is the ratio of specific heats at constant pressure and volume. For a solution to exist, we must specify the initial conditions in front of the shock wave, i.e. for $r \leq r_{\rm shock}(0)$. For the Guderley problem, the state of the pre-shocked region is : $\rho(r,0) = \rho_0 = \text{const}$, with vanishing initial velocity and pressure, $u(r,0) = 0$ and $p(r,0) = 0$. The default density is in cgs units is $\rho_0 = 1\,\mathrm{g/cm^3}$.

Note: *exactpack.solvers.guderley* loads *exactpack.solvers.guderley.ramsey*.

6.5.1 exactpack.solvers.guderley.ramsey

A Fortran based Guderley solver.

This is a Python interface to Scott Ramsey's Guderley fortran code.

class exactpack.solvers.guderley.ramsey.**Guderley**(***params*)

Computes the solution to the Guderley problem.

Parameters

- **geometry** – 1=planar, 2=cylindrical, 3=spherical

- **rho0** – initial uniform density
- **gamma** – specific heat ratio

6.6 Heat Conduction Problems

Heat Conduction Test Problems

The Heat Conduction test problems are meant to explore interesting and numerically relevant regimes of the classical diffusive heat flow problem in 1, 2, and 3 spatial dimensions, usually in simple geometries such as planar, cylindrical or spherical. The heat flow problem is "local", in that it may be specified on a (possibly compact) connected spatial region Ω with boundary $\partial\Omega$. The temperature at point $\mathbf{x} \in \Omega$ and time $t > 0$ is denoted by $T = T(\mathbf{x}, t)$, and satisfies the diffusion equation

$$\frac{\partial T}{\partial t} = \vec{\nabla} \cdot \left[\kappa(\mathbf{x}) \, \vec{\nabla} T \right] , \tag{6.17}$$

for all $\mathbf{x} \in \Omega$ and $t > 0$, with $\kappa = \kappa(\mathbf{x})$ being the heat diffusion parameter at position $\mathbf{x}$. For simplicity, we assume that κ is independent of position, in which case, the heat flow equation reduces to

$$\frac{1}{\kappa} \frac{\partial T}{\partial t} = \nabla^2 T \qquad \mathbf{x} \in \Omega \text{ and } t > 0 \, . \tag{6.18}$$

To solve this equation, we must provide an initial temperature profile $T_0(\mathbf{x})$ for all $\mathbf{x} \in \Omega$, and a boundary condition $\gamma(\mathbf{x})$ that specifies the temperature on $\mathbf{x} \in \partial\Omega$ for arbitrary time,

$$\begin{aligned} T(\mathbf{x}, 0) &= T_0(\mathbf{x}) \qquad && \mathbf{x} \in \Omega \, , \\ T(\mathbf{x}, t) &= \gamma(\mathbf{x}) \qquad && \mathbf{x} \in \partial\Omega \, , \, t > 0 \, . \end{aligned} \tag{6.19}$$

In fact, we will often consider the more general boundary condition (BC) in which a linear combination of the temperature and heat flux is specified,

$$\alpha \, T(\mathbf{x}, t) + \beta \, \vec{\mathbf{n}} \cdot \vec{\nabla} T(\mathbf{x}, t) = \gamma(\mathbf{x}) \qquad \mathbf{x} \in \partial\Omega \, , \, t > 0 \, , \tag{6.20}$$

where $\vec{\mathbf{n}} = \vec{\mathbf{n}}(\mathbf{x})$ is the outward normal to the boundary at $\mathbf{x} \in \partial\Omega$. The coefficients $\alpha = \alpha(\mathbf{x})$ and $\beta = \beta(\mathbf{x})$ are in general functions of the position on the boundary, although we shall usually take them to be constants independent of position.

In finding the solution for this problem, there are two cases to consider: homogeneous and nonhomogeneous BC's. In the homogeneous case, the right-hand side of the BC is taken to vanish, $\gamma = 0$, and the problem becomes

$$\begin{aligned} \frac{\partial \tilde{T}}{\partial t} &= \kappa^2 \, \nabla^2 \tilde{T} \qquad && \mathbf{x} \in \Omega \, , \, t > 0 \\ \alpha \, \tilde{T} + \beta \, \vec{\mathbf{n}} \cdot \vec{\nabla} \tilde{T} &= 0 \qquad && \mathbf{x} \in \partial\Omega \, , \, t > 0 \, , \end{aligned} \tag{6.21}$$

where we have used the notation $\tilde{T}(\mathbf{x}, t)$ to denote the homogeneous solution. In this case, the sum of any number of solutions remains a solution (this is not true for nonhomogeneous BC's, for which $\gamma \neq 0$), and consequently, the most general homogeneous solution can be found by separation of variables and a normal mode analysis on the spatial solutions. This gives a general solution of the form

$$\tilde{T}(\mathbf{x}, t) = \sum_n D_n \, X_n(\mathbf{x}) \, e^{-\kappa \, k_n^2 t} \, , \tag{6.22}$$

where the modes $X_n(\mathbf{x})$ satisfy the Helmholtz equation

$$\nabla^2 X_n + k_n^2 \, X_n = 0 \, , \tag{6.23}$$

and the mode numbers k_n are determined from the homogeneous BC's. The modes X_n are orthogonal, and since the Helmholtz equation is linear, we can scale the X_n to be orthonormal,

$$\int_\Omega dx\, X_n(\mathbf{x})\, X_m(\mathbf{x}) = \delta_{nm} \ . \tag{6.24}$$

The values of the expansion coefficients D_n will be determined from the initial conditions (IC's) and the nonhomogeneous solution, as given by (6.28) below.

The next step is to find a specific nonhomogeneous solution, which we accomplish by solving the static equation (Laplace's equation) with the appropriate nonhomogeneous BC,

$$\begin{aligned} \nabla^2 \bar{T}(\mathbf{x}) &= 0 \qquad && \mathbf{x} \in \Omega \\ \alpha\, \bar{T}(\mathbf{x}) + \beta\, \vec{\mathbf{n}} \cdot \vec{\nabla} \bar{T}(\mathbf{x}) &= \gamma(\mathbf{x}) \qquad && \mathbf{x} \in \partial\Omega \ . \end{aligned} \tag{6.25}$$

We denote the nonhomogeneous solution by $\bar{T}(\mathbf{x})$.

Once we have found $\tilde{T}$ and $\bar{T}$, the most general time dependent solution is the sum of the homogeneous and nonhomogeneous solutions,

$$T(\mathbf{x}, t) = \tilde{T}(\mathbf{x}, t) + \bar{T}(\mathbf{x}) \ . \tag{6.26}$$

The initial condition now becomes $T_0(\mathbf{x}) = \tilde{T}(\mathbf{x}, 0) + \bar{T}(\mathbf{x})$, or

$$\sum_n D_n\, X_n(\mathbf{x}) = T_0(\mathbf{x}) - \bar{T}(\mathbf{x}) \ , \tag{6.27}$$

which can be solved for D_n using the orthogonality condition (6.24),

$$D_n = \int_\Omega dx \left[T_0(\mathbf{x}) - \bar{T}(\mathbf{x}) \right] X_m(\mathbf{x}) \ . \tag{6.28}$$

6.6.1 exactpack.solvers.heat.rod1d

Heat flow in a 1D rod with boundary and initial conditions.

This problem involves heat conduction in a 1D rod of length L. The heat conduction equation for the temperature profile $T(x, t)$ is

$$\frac{\partial T}{\partial t} = \kappa\, \frac{\partial^2 T}{\partial x^2} \ , \tag{6.29}$$

where κ is the (constant) thermal conductivity of the rod. We place the ends of the rod at $x = 0$ and $x = L$, and use a linear combination of Dirichlet and Neumann and boundary conditions,

$$\begin{aligned} \alpha_1 T(0, t) + \beta_1 \partial_x T(0, t) &= 0 \\ \alpha_2 T(0, t) + \beta_2 \partial_x T(0, t) &= 0 \ , \end{aligned} \tag{6.30}$$

with $t > 0$. Finally, we must provide a continuous initial temperature profile $T(x, 0)$ to obtain a solution to the differential equation. For simplicity we shall consider only two cases: $T(x, 0)$ is a constant temperature T_0, or $T(x, 0)$ varies linearly between T_0 and T_1 over the length of the rod,

$$T(x, 0) = T_0 + \frac{T_1 - T_0}{L} \cdot x \ , \tag{6.31}$$

for $x \in (0, L)$. Note that $T(x = 0, 0) = T_0$ and $T(x = L, 0) = T_1$.

In general, the boundary conditions (BC's) give a transcendental equation, but there are four special cases that are particularly simple.

BC1: The first boundary condition choice is

$$\begin{aligned} T(0,t) &= 0 \\ T(L,t) &= 0 \,, \end{aligned} \tag{6.32}$$

which corresponds to $\alpha_1 = 1$, $\beta_1 = 0$, $\alpha_2 = 1$, and $\beta_2 = 0$. The solution takes the form

$$\begin{aligned} T(x,t) &= \sum_{n=1}^{\infty} \left[\frac{2T_0 - 2T_1\,(-1)^n}{n\pi} \right] \sin k_n x \, e^{-\kappa\, k_n^2 t} \\ k_n &= \frac{n\pi}{L} \,. \end{aligned} \tag{6.33}$$

BC2: The second boundary condition choice is

$$\begin{aligned} \partial_x T(0,t) &= 0 \\ \partial_x T(L,t) &= 0 \,, \end{aligned} \tag{6.34}$$

which corresponds to $\alpha_1 = 0$, $\beta_1 = 1$, $\alpha_2 = 0$, and $\beta_2 = 1$. The solution takes the form

$$\begin{aligned} T(x,t) &= \frac{1}{2}(T_0 + T_1) + \sum_{n=1}^{\infty} \left[2(T_1 - T_0)\, \frac{1-(-1)^n}{n^2\pi^2} \right] \cos k_n x \, e^{-\kappa\, k_n^2 t} \\ k_n &= \frac{n\pi}{L} \,. \end{aligned} \tag{6.35}$$

For the case of constant initial condition $T(x,0) = T_0$, or equivalently for $T_1 = T_0$, note that the solution reduces to the time independent form $T(x,t) = T_0$. This is because the BC's prevent heat from flowing across the boundaries, and the initial temperature profile remains fixed.

BC3: The third boundary condition choice is

$$\begin{aligned} T(0,t) &= 0 \\ \partial_x T(L,t) &= 0 \,, \end{aligned} \tag{6.36}$$

which corresponds to $\alpha_1 = 1$, $\beta_1 = 0$, $\alpha_2 = 0$, and $\beta_2 = 1$. The solution takes the form

$$\begin{aligned} T(x,t) &= \sum_{n=0}^{\infty} \left[\frac{4T_1}{(2n+1)\pi} - \frac{8(T_1 - T_0)}{(2n+1)^2\pi^2} \right] \sin k_n x \, e^{-\kappa\, k_n^2 t} \\ k_n &= \frac{(2n+1)\pi}{2L} \,. \end{aligned} \tag{6.37}$$

BC4: The fourth boundary condition choice is

$$\begin{aligned} \partial_x T(0,t) &= 0 \\ T(L,t) &= 0 \,, \end{aligned} \tag{6.38}$$

which corresponds to $\alpha_1 = 0$, $\beta_1 = 1$, $\alpha_2 = 1$, and $\beta_2 = 0$. On physical grounds, the solution must be identical to the one for BC3, except that the rod has been inverted. In terms of a series expansion, we have

$$\begin{aligned} T(x,t) &= \sum_{n=0}^{\infty} \left[\frac{4T_0\,(-1)^n}{(2n+1)\pi} - \frac{8(T_1 - T_0)}{(2n+1)^2\pi^2} \Big[1 - (-1)^n \Big] \right] \cos k_n x \, e^{-\kappa\, k_n^2 t} \\ k_n &= \frac{(2n+1)\pi}{2L} \,. \end{aligned} \tag{6.39}$$

The solutions for BC3 and BC4 appear to be quite different; however, they are indeed just a reflection across the midpoint of the rod. Since the infinite sums in ExactPack must be truncated at some order, comparing BC3 against BC4 provides a good metric to determine whether the truncation has been performed to sufficient accuracy. The first 100 terms seems gives reasonable results.

General BC: The solution is of the form

$$T(x,t) = \sum_n \Big[A_n \cos k_n x + B_n \sin k_n x \Big] \, e^{-\kappa \, k_n^2 t} \ ,$$

where the wave numbers are

$$k_n = \frac{\mu_n}{L} \ ,$$

with μ_n being the solutions to

$$\tan \mu = \frac{(\alpha_2 \bar{\beta}_1 - \alpha_1 \bar{\beta}_2)\mu}{\alpha_1 \alpha_2 + \bar{\beta}_1 \bar{\beta}_2 \mu^2} \quad \text{for} \ \ \bar{\beta}_i = \beta_i / L \ .$$

Case I: $\alpha_1 \neq 0$. The coefficients A_n and B_n are related by

$$A_n = -\frac{\beta_1 k_n}{\alpha_1} \, B_n \ ,$$

which leads us to write

$$X_n(x) = \sin k_n x - \frac{\beta_1 k_n}{\alpha_1} \, \cos k_n x$$

$$T(x,t) = \sum_{n=1}^{\infty} B_n \, X_n(x) \, e^{-\kappa \, k_n^2 t} \ .$$

Note that $n = 0$ does not contribute since $\mu_0 = 0$ means that $k_0 = 0$, and therefore $A_0 = 0$ and $\sin k_0 x = 0$ (for $\alpha_1 \neq 0$). The modes X_n are orthogonal for $n \neq m$, and calculating the normalization N_n is straightforward but tedious:

$$\int_0^L dx \, X_n(x) \, X_m(x) = N_n \, \delta_{nm}$$

$$\text{where} \ \ N_n = \frac{1}{4\alpha_1 k_n^2} \Big[-2\alpha_1 \beta_1 k_n + 2(\beta_1^2 k_n^2 + \alpha_1^2) k_n L + 2\alpha_1 \beta_1 k_n \cos 2k_n L + (\beta_1^2 k_n^2 - \alpha_1^2) \sin 2k_n L \Big] \ .$$

The coefficients B_n can be calculated from

$$B_n = \frac{1}{N_n} \int_0^L dx \, T(x,0) \, X_n(x) \ .$$

Since the expression for B_n is rather long for a constant plus a linear initial condition, $T(x,0) = T_0 + (T_1 - T_0) \, x/L$, we divide the result into two pieces:

$$T^{(0)}(x,0) = T_0 :$$

$$B_n^{(0)} = \frac{T_0}{N_n} \Big[\frac{1 - \cos k_n L}{k_n} - \frac{\beta_1}{\alpha_1} \, \sin k_n L \Big] \ ,$$

and

$$T^{(1)}(x,0) = \frac{T_1 - T_0}{L} \, x :$$

$$B_n^{(1)} = \frac{T_1 - T_0}{N_n \, L} \, \frac{1}{\alpha_1 k_n^2} \Big[\beta_1 k_n - (\alpha_1 k_n L + \beta_1 k_n) \cos k_n L + (\alpha_1 - \beta_1 k) n^2 L) \sin k_n L \Big] \ .$$

Since the boundary conditions are homogenous, we may add the two solutions the constant plus linear initial condition, $T(x,0) = T^{(0)}(x,0) + X^{(1)}(x,0)$, in which case $B_n = B_n^{(0)} + B_n^{(1)}$.

Case II: $\alpha_1 = 0$. We may assume that $\beta_2 \neq 0$, otherwise this is BC4, which has already been treated. The wave numbers are given by $k_n = \mu_n / L$, where μ_n are the solutions to

$$\tan \mu = \frac{\alpha_2}{\beta_2} \frac{1}{\mu} .$$

The coefficients of the sin-function vanish,

$$B_n = 0 ,$$

which leads us to write

$$X_n(x) = \cos k_n x$$

$$T(x,t) = \sum_{n=1}^{\infty} A_n \, X_n(x) \, e^{-\kappa \, k_n^2 t} .$$

The orthogonality relation is

$$\int_0^L dx \, X_n(x) \, X_m(x) = N_n \, \delta_{nm}$$

$$\text{where } \; N_n = \frac{1}{4k_n} \Big[2k_n L + sin 2k_n L \Big] .$$

The coefficients B_n can be calculated from

$$A_n = \frac{1}{N_n} \int_0^L dx \, T(x,0) \, X_n(x)$$

$$= T_0 \, \frac{sin k_n L}{k_n} + \frac{(T_1 - T)}{L k_n^2} \Big[-1 + \cos k_n L + k_n L \, \sin k_n L \Big] .$$

class `exactpack.solvers.heat.rod1d.`**`Rod1D`**(***kwargs*)

Computes the solution to the 1D heat conduction problem om rod for boundary conditions given by linear combinations of Neumann and Dirichlet.

The four boundary condition types BC1, BC2, BC3, BC4 have been implements for homogeneous and nonhomogeneous cases. BC1 is equivalent to the planar sandwich, and BC2 to the hot planar sandwich. Note that BC3 and BC4 are physically equivalent correspond to the same physical system, which one might call the planar half sandwich, with constant temperature at one end, and constant heat flux at the other. This is a new test problem, not mentioned by Dawes et al. The general case for rod1D might be called PlanarSandwichTheWorks.

Parameters

- **`gamma2`** – nonhomogeneous BC parameter for x=L
- **`gamma1`** – nonhomogeneous BC parameter at x=0
- **`L`** – Length of the rod
- **`Nsum`** – Number of terms to include in the sum.
- **`alpha2`** – BC parameter for temperature at x=L
- **`alpha1`** – BC parameter for temperature at x=0
- **`kappa`** – Thermal diffusivity

- **TR** – Value of IC profile for the right end of the rod at x=L
- **TL** – Value of IC profile for the left end of the rod at x=0
- **beta2** – BC parameter for flux at x=L
- **beta1** – BC parameter for flux at x=0

modes_BC1()
Computes coefficients A_n and B_n and the modes k_n for the first boundary condition special case:

$\alpha_1 \neq 0$, $\beta_1 = 0$, $\alpha_2 \neq 0$, and $\beta_2 = 0$

modes_BC2()
Computes coefficients A_n and B_n and the modes k_n for the second boundary condition special case:

$\alpha_1 = 0$, $\beta_1 \neq 0$, $\alpha_2 = 0$, and $\beta_2 \neq 0$

modes_BC3()
Computes coefficients A_n and B_n and the modes k_n for the third boundary condition special case:

$\alpha_1 \neq 0$, $\beta_1 = 0$, $\alpha_2 = 0$, and $\beta_2 \neq 0$

modes_BC4()
Computes coefficients A_n and B_n and the modes k_n for the fourth boundary condition special case:

$\alpha_1 = 0$, $\beta_1 \neq 0$, $\alpha_2 \neq 0$, and $\beta_2 = 0$

modes_BCgen()
Computes coefficients A_n and B_n and the modes k_n for the general boundary condition:

$\alpha_1 \neq 0$, $\beta_1 \neq 0$, $\alpha_2 \neq 0$, and $\beta_2 \neq 0$

6.6.2 exactpack.solvers.heat.planar_sandwich

The Planar Sandwich *[Dawes2016]* implemented in python, which is the default in ExactPack.

This is the planar sandwich heat conduction problem, a heat flow problem in 2D rectangular coordinates x and y. See Ref. *[Dawes2016]* for details. The problem consists of three material layers aligned along the y-direction. The outer two layers do not conduct heat ($\kappa = 0$), while the inner layer is heat conducting with $\kappa > 0$, forming a sandwich of conducting and non-conducting materials. When the initial temperatures and boundary conditions do not vary along the x-direction, on both the upper and lower boundaries, symmetry arguments imply that heat only flows in the y-direction,

$$\frac{\partial T}{\partial t} = \kappa \frac{\partial^2 T}{\partial y^2} ,$$

and the problem reduces to a 1D heat equation. The lower side of the sandwich along $y = 0$ is held at a constant temperature T_0, while the temperature of the upper portion of the sandwich at $y = L$ is T_L, providing the boundary condtions

$$T(0, t) = T_0$$

$$T(L, t) = T_L .$$

As for initial conditions, the sandwich starts at zero temperature,

$$T(x, 0) = 0 .$$

Variable separation and Fourier analysis yield the solution

$$T(y,t) = T_0 + \frac{T_L - T_0}{L}\, y + \sum_{n=1}^{\infty} \left[\frac{2T_L\,(-1)^n - 2T_0}{n\pi} \right] \sin k_n y\, e^{-\kappa\, k_n^2 t}$$

$$k_n = \frac{n\pi}{L}\ .$$

Not surprisingly, this is the solution BC1 of the 1D rod (modulo a relative sign).

class `exactpack.solvers.heat.planar_sandwich.`**`PlanarSandwich`**(***kwargs*)

Computes the solution to the Planar Sandwich heat conduction problem. This is direct python interface to fortran source code provided Alan Dawes.

Parameters

- **`kappa`** – Thermal diffusivity
- **`TT`** – The boundary condition at the top.
- **`TR`** – Value of IC profile for the right end of the rod at x=L
- **`L`** – Length of the rod
- **`TL`** – Value of IC profile for the left end of the rod at x=0
- **`Nsum`** – Number of terms to include in the sum.
- **`TB`** – The boundary condition at the bottom

6.6.3 `exactpack.solvers.heat.cylindrical_sandwich`

The Cylindrical Sandwich *[Dawes2016]* .

This test problem was proposed by Allan Dawes in Ref. *[Dawes2015]*, and the exact analytic solution was presented in Appendix B of Ref. *[Dawes2016]*. This problems considers heat conduction in a 2D annulus with inner radius $r = a$ and outer radius $r = b$, with $0 \le \theta \le \pi/2$ (the first quadrant of the plane). The heat conduction equation on the annulus, expressed in cylindrical coordinates, takes the form

$$\begin{aligned} \frac{1}{\kappa}\frac{\partial T}{\partial t} &= \nabla^2 T \\ &= \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial T}{\partial r} \right) + \frac{1}{r^2}\frac{\partial^2 T}{\partial \theta^2}\ . \end{aligned}$$

The differential equation holds in the open annular region $a < r < b$ and $0 < \theta < \pi/2$ at time $t > 0$, and we take the initial condition in this region are taken as

$$T(r,\theta,t=0) = 0\ .$$

The boundary conditions are

$$\begin{aligned} T(r,\theta=0,t) &= T_0 \\ T(r,\theta=\pi/2,t) &= T_1 \\ \partial_r T(r=a,\theta,t) &= 0 \\ \partial_r T(r=b,\theta,t) &= 0\ . \end{aligned}$$

The boundary condition at $\theta = 0, \pi/2$ render the system non-homogeneous. Therefore, the solution is a combination of a general homogeneous solution and a specific time-independent solution to the non-homogeneous Laplace equation,

$$T(r,\theta,t) = \tilde{T}(r,\theta,t) + \bar{T}(r,\theta)\ .$$

The non-homogeneous Laplace equation is

$$\nabla^2 \bar{T} = \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial \bar{T}}{\partial r}\right) + \frac{1}{r^2}\frac{\partial^2 \bar{T}}{\partial \theta^2} = 0$$

$$\begin{aligned} \bar{T}(r, \theta = 0) = T_1 \quad & \bar{T}(r, \theta = \pi/2) = T_1 \\ \partial_r \bar{T}(r = a, \theta) = 0 \quad & \partial_r \bar{T}(r = b, \theta = 0) = 0 \,, \end{aligned}$$

which has the solution

$$\bar{T}(r, \theta) = T_0 + \frac{2\theta}{\pi} T_1 \,.$$

The homogeneous heat equation becomes

$$\frac{1}{\kappa}\frac{\partial \tilde{T}}{\partial t} = \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial \tilde{T}}{\partial r}\right) + \frac{1}{r^2}\frac{\partial^2 \tilde{T}}{\partial \theta^2}$$

$$\tilde{T}(r, \theta, t = 0) = -\bar{T}(r, \theta)$$

$$\begin{aligned} \tilde{T}(r, \theta = 0) = 0 \quad & \tilde{T}(r, \theta = \pi/2) = 0 \\ \partial_r \tilde{T}(r = a, \theta) = 0 \quad & \partial_r \tilde{T}(r = b, \theta = 0) = 0 \,. \end{aligned}$$

The initial condition is of the form of a constant plus linear term.

NOTE This solver is currently only accurate to 10^{-3}. This is because the Bessel functions become highly oscillatory for large mode numbers, and the quadrature method used to calculated the coefficients becomes unreliable. For higher accuracy, one must use an asymptotic form of the high mode Bessel functions should be used. This will be implemented in a future release.

class exactpack.solvers.heat.cylindrical_sandwich.**CylindricalSandwich**(***kwargs*)

Computes the solution to Cylindrical Sandwich *[Dawes2016]* .

Parameters

- **a** – Inner radius of annulus
- **Msum** – Number of terms to include in the m-sum.
- **b** – Outer radius of annulus
- **kappa** – Thermal diffusivity
- **Nsum** – Number of terms to include in the n-sum.
- **NonHomogeneousOnly** – If True, then compute only the static nonhomogeneous solution.
- **T0** – Temperature boundary condition along $\theta = 0$ for $a \leq r \leq b$
- **T1** – Temperature boundary condition along $\theta = \pi/2$ for $a \leq r \leq b$

6.6.4 exactpack.solvers.heat.planar_sandwich_dawes

The Planar Sandwich *[Dawes2016]* implemented in Fortran. This is a python interface to Alan Dawes' fortran code.

class exactpack.solvers.heat.planar_sandwich_dawes.**PlanarSandwichDawes**(***kwargs*)

Computes the solution to Planar Sandwich. This is direct python interface to fortran source code provided Alan Dawes.

Parameters

- **TB** – Temperature BC on the bottom of the square at y=0
- **TT** – Temperature BC on the top of the square at y=L
- **Nsum** – Number of terms to include in the sum.
- **kappa** – Thermal diffusivity
- **L** – Length of the square in the y-direction

6.6.5 exactpack.solvers.heat.rectangle

The Single Component Rectangle. This test problem consists of a rectangular region of length a along the x-direction and length b along the y-direction. The boundary conditions are given by keeping the bottom of the rectangle at zero temperature and the top at a fixed temperature. The sides of the rectangle use zero heat flux conditions. The initial temperature of the rectangle is taken to vanish.

class exactpack.solvers.heat.rectangle.**Rectangle**(***kwargs*)

Computes the solution to a rectangular heat flow problem.

Parameters

- **a** – Length of the rectangle in the x-direction
- **b** – Length of the rectangle in the y-direction
- **kappa** – Thermal diffusivity
- **Ttop** – Temperature BC on the top of the rectangle at y=b
- **Nsum** – Number of terms to include in the sums.
- **NonHomogeneousOnly** – If True, then compute only the static nonhomogeneous solution.

6.6.6 exactpack.solvers.heat.hutchens1

This is the first heat conduction problem of Hutchens in Ref. *[Hutchens2009]* .

This problem involves the heat conduction of a spherical mass of radius $r = b$, with the material priorities of mass density ρ, thermal conductivity k, and specific heat at constant pressure $c_{\rm p}$. In terms of these properties, the thermal diffusivity is given by

$$\alpha = \frac{k}{\rho c_{\rm p}} \ .$$

The heat conduction equation for the temperature profile $T(r,t)$ takes the following form in spherical coordinates,

$$\frac{\partial T}{\partial t} = \alpha \, \frac{1}{r^2} \frac{\partial}{\partial r} \left(r^2 \frac{\partial T}{\partial r} \right) \ .$$

The initial temperature of the sphere is taken to be $T = T_0$, while the radial surface temperature of the sphere is held at a fixed at temperature $T = T_b$, in which case the initial and boundary conditions are,

$$T(r,0) = T_0$$

$$\frac{\partial T}{\partial r}(0,t) = 0$$

$$T(b,t) = T_b \ .$$

A Fourier analysis and the separation of variables technique yields the exact solution,

$$T(r,t) = T_b + \frac{2b}{\pi r}\Big(T_b - T_0\Big)\sum_{n=1}^{\infty}\frac{(-1)^n}{n}\,\sin\Big(\frac{n\pi}{b}\,r\Big)\exp\left(-\alpha\frac{n^2\pi^2}{b^2}\,t\right)\,.$$

In practice, we must truncate the series at some maximum value $n_{\max} = N$. Care must be taken when choosing the value of N, especially since the series solution is alternating in sign. For the iron sphere of problem 1 in Ref. *[Hutchens2009]* , the choice $N = 100$ is made.

class exactpack.solvers.heat.hutchens1.**Hutchens1**(***kwargs*)

Computes the solution to the first heat conduction problem of Hutchens.

This solver is given in unitis in which the temperature is measured in eV. Arbitrary units can be chosen, but then then the user must supply a value for the Boltzmann constant $k_{\rm B}$, or alternatively, the radiation constant a.

Parameters

- **b** – Radius of the sphere [cm]
- **k** – thermal conductivity [erg/s-cm-eV]
- **T0** – Initial uniform temperature of the sphere‘ [eV]
- **Nsum** – Number of terms to include in the sum.
- **rho** – density [g/cm^3]
- **cp** – specific heat at constant pressure [erg/g-eV]
- **Tb** – Temperature of the radial boundary at $r = b$ [eV]

6.6.7 exactpack.solvers.heat.hutchens2

This is the second heat conduction problem of Hutchens in Ref. *[Hutchens2009]* .

This problem involves the heat conduction of a cylindrical mass of radius $r = b$ and height $z = L$.

The steady state heat conduction equation for the temperature profile $T(r,z,t)$ takes the following form in cylindrical coordinates,

$$\frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial T}{\partial r}\right) + \frac{\partial^2 T}{\partial z^2} + \frac{g_0}{k} = 0\,.$$

The boundary conditions are

$$\begin{aligned}\frac{\partial T}{\partial r}(r=0,z) &= 0\\ T(r=b,z) &= T_b\\ T(r,z=0) &= T_0\\ T(r,z=L) &= T_L\,.\end{aligned}$$

A Fourier analysis and the separation of variables technique yields the exact solution,

$$\begin{aligned}T(r,t) = T_b + \Big(T_L - T_0\Big)\frac{z}{L} + \frac{1}{2}\frac{g_0}{k}z\Big(L-z\Big) + \\ \frac{2}{\pi}\,T_L\sum_{n=0}^{\infty}\frac{(-1)^{2n+1}}{2n+1}\,\frac{I_0(\lambda_n r)}{I_0(\lambda_n b)}\,\sin(\lambda_n z) - \frac{4g_0L^2}{\pi^3 k}\sum_{n=0}^{\infty}\frac{1}{(2n+1)^3}\frac{I_0(\lambda_n r)}{I_0(\lambda_n b)}\,\sin(\lambda_n z)\,,\end{aligned}$$

where $I_0(z)$ is a modified Bessel function of the first kind, and

$$\lambda_n = \frac{(2n+1)\pi}{L} \ .$$

In practice, we must truncate the series at some maximum value $n_{\max} = N$. Care must be taken when choosing the value of N, especially since the series solution is alternating in sign. For the iron sphere of problem 1 in Ref. *[Hutchens2009]* , the choice $N = 100$ is made.

class `exactpack.solvers.heat.hutchens2.`**`Hutchens2`**(***kwargs*)

Computes the solution to the second heat conduction problem of Hutchens.

This solver is given in unitis in which the temperature is measured in eV. Arbitrary units can be chosen, but then then the user must supply a value for the Boltzmann constant $k_{\rm B}$, or alternatively, the radiation constant a.

Parameters

- **`TL`** – Temperature of the cylinder at $z = L$ [eV]
- **`Nsum`** – Number of terms to include in the sum.
- **`g0`** – rate of heat generation [erg/s-cm^3]
- **`k`** – thermal conductivity [erg/s-cm-eV]
- **`b`** – Radius of the cylinder [cm]
- **`Tb`** – Temperature of the radial boundary at $r = b$ [eV]
- **`L`** – Height of cylinder [cm]
- **`T0`** – Temperature of the cylinder at $z = 0$ [eV]

6.7 The Kenamond Problems

Python based solvers for high explosive burn times when using programmed burn models.

This suite of solvers calculates high explosive (HE) burn times for programmed burn models *[Kenamond2011]*. The Kenamond HE problems are a series of problems designed to test the burn table solution (HE light times) generated for programmed burn simulations. The suite of test problems has exact solutions in 2D and 3D.

It should be understood that these burn time calculations are purely geometry-based solutions. They do not account for HE behaviors such as shock formation time, inert boundary behaviors, or behavior at boundaries between two high explosives.

6.7.1 `exactpack.solvers.kenamond.kenamond1`

A Kenamond1 solver in Python.

This is a pure Python implementation of the Kenamond1 solution using Numpy.

The Kenamond HE Problem 1 is used to test a code's ability to calculate burn time tables for an unobstructed line-of-sight, single-point initiation of a single HE region. An infinite medium of a single HE with constant detonation velocity D is ignited at time $t = t_d$ by a single point detonator located at $\vec{x} = \vec{x}_d$.

The HE light time solution for spherical propagation at a specified detonation velocity becomes:

$$t(\vec{x}) = t_d + \frac{||\vec{x} - \vec{x}_d||}{D}$$

class exactpack.solvers.kenamond.kenamond1.**Kenamond1**(***kwargs*)

Computes the general solution to the Kenamond HE Problem 1.

Can be used to solve the Kenamond1 problem for a high explosive with any detonation velocity, D, detonator location, $\vec{x}_d$, and detonation time, t_d. Supports 2D and 3D Cartesian solutions.

Default values are selected to be consistent with the problem definition in *[Kenamond2011]*. Default values are: **geometry** $= 2$, $D = 1.0$, $x_d = (0.0, 0.0)$, and $t_d = 0.0$.

Parameters

- **geometry** – 2=two-dimensional, 3=three-dimensional
- **x_d** – detonator location, enter as a tuple: (x, y [, z])
- **D** – detonation velocity of the HE
- **t_d** – detonation time

Input evaluation points as an N x **geometry** 2D array of positions: [[x_0, y_0 (, z_0)], [x_1, y_1 (, z_1)], ..., [x_N, y_N (, z_N)]].

A time value must be input in order to use the ExactPack machinery. The time value is ignored in the calculation.

6.7.2 exactpack.solvers.kenamond.kenamond2

A Kenamond2 solver in Python.

This is a pure Python implementation of the Kenamond2 solution using Numpy.

The Kenamond HE Problem 2 *[KaulK22016]* is used to test a code's ability to calculate burn time tables for an unobstructed line-of-sight, multi-point initiation of a multiple HE region. An HE sphere of radius R centered at the origin with constant detonation velocity D_1 is surrounded by an infinite medium of a second HE with constant detonation velocity D_2. Five point detonators located at $\vec{x} = \vec{x}_{d_i}$ are ignited at times $t = t_{d_i}$. The detonators are located on the y-axis in the 2D test and on the z-axis in the 3D test.

This solver implementation is based on some specific conditions.

- The detonation velocity of the inner HE region is higher than the detonation velocity of the outer HE region: $D_1 > D_2$.
- Detonator 3 is located at the origin.
- Only detonator 3 is located inside the inner HE region.
- The detonation times for the other detonators must allow the burn wave from detonator 3 to exit the inner HE region before interaction.

In order to meet the last criteria, each detonation time must satisfy the following condition:

$$t_{d_i} \geq t_{d_3} + R\left(\frac{1}{D_1} + \frac{1}{D_2}\right) - \frac{|a_{d_i}|}{D_2}$$

where $i = 1, 2, 4, 5$ and a_{d_i} is the axial location of detonator d_i ($a_{d_i} = y_{d_i}$ in 2D and $a_{d_i} = z_{d_i}$ in 3D).

With these conditions, the burntime solution at the point $\vec{p} = (x, y\ [, z])$ is as follows;

$$t(\vec{p}) = \min(t_1, t_2, \max(t_3, t_4), t_5, t_6)$$

where,

$$t_1(\vec{p}) = t_{d_1} + \frac{||\vec{p} - \vec{x}_{d_1}||}{D_2}$$
$$t_2(\vec{p}) = t_{d_2} + \frac{||\vec{p} - \vec{x}_{d_2}||}{D_2}$$
$$t_3(\vec{p}) = t_{d_3} + \frac{||\vec{p} - \vec{x}_{d_3}||}{D_1}$$
$$t_4(\vec{p}) = t_{d_3} + \frac{||\vec{p} - \vec{x}_{d_3}||}{D_2} + R\left(\frac{1}{D_1} - \frac{1}{D_2}\right)$$
$$t_5(\vec{p}) = t_{d_4} + \frac{||\vec{p} - \vec{x}_{d_4}||}{D_2}$$
$$t_6(\vec{p}) = t_{d_5} + \frac{||\vec{p} - \vec{x}_{d_5}||}{D_2}$$

class `exactpack.solvers.kenamond.kenamond2.`**`Kenamond2`**(***kwargs*)

Computes the general solution to the Kenamond HE Problem 2.

The detonator locations are on the y-axis in 2D and on the z-axis in 3D. Detonator 3 will be located at the origin. User will specify the other four y or z axial detonator locations and all five detonation times.

Default values are selected to be consistent with the problem definition in *[Kenamond2011]*. Default values are **geometry** $= 2$, $R = 3.0$, $D_1 = 2.0$, $D_2 = 1.0$, $t_d = [2.0, 1.0, 0.0, 1, 0, 2.0]$, and **dets** $= [10.0, 5.0, -5.0, -10.0]$.

Parameters

- **`geometry`** – 2=two-dimensional, 3=three-dimensional
- **`t_d`** – detonation times, enter as a list: $[t_{d_1}, t_{d_2}, t_{d_3}, t_{d_4}, t_{d_5}]$
- **`R`** – radius of inner HE
- **`dets`** – axial detonator locations, enter as a list: $[a_{d_1}, a_{d_2}, a_{d_4}, a_{d_5}]$. Detonator 3 will be automatically inserted at the origin.
- **`D2`** – detonation velocity of the outer HE, D2 < D1
- **`D1`** – detonation velocity of the inner HE

Input evaluation points as an N x **geometry** 2D array of positions: [[x_0, y_0 (, z_0)], [x_1, y_1 (, z_1)], ..., [x_N, y_N (, z_N)]].

A time value must be input in order to use the ExactPack machinery. The time value is ignored in the calculation.

6.7.3 `exactpack.solvers.kenamond.kenamond3`

A Kenamond3 solver in Python.

This is a pure Python implementation of the Kenamond3 solution using Numpy.

The Kenamond HE Problem 3 *[KaulK32016]* is used to test a code's ability to calculate burn time tables for a single-point initiation of a single HE region surrounding an inert region. An infinite medium of a single HE with constant detonation velocity D surrounds an inert spherical obstacle of radius R centered at the origin. A single point detonator located at $\vec{x} = \vec{x}_d$ is ignited at time $t = t_d$. The detonator must be located outside of the inert region.

The HE material can be divided into two solution regions: the material in the line-of-sight of the detonator and the material in the shadow of the inert object. Let t_1 designate the line-of-sight region and t_2 designate the shadow region.

The burntime solution at the point $\vec{p} = (x, y\ [, z])$ is as follows:

$$t(\vec{p}) = \begin{cases} t_1 & \text{if } \theta \leq 0 \\ t_2 & \text{if } \theta > 0 \end{cases}$$

where,

$$\begin{aligned} t_1(\vec{p}) &= t_d + \frac{||\vec{p} - \vec{x}_d||}{D} \\ t_2(\vec{p}) &= t_d + \frac{l_{da} + l_{ab} + l_{bp}}{D} \\ \theta &= \pi - \alpha - \beta - \psi \\ \alpha &= \arccos\left(-\frac{\vec{p} \cdot \vec{x}_d}{||\vec{p}||||\vec{x}_d||}\right) \\ \beta &= \arccos\left(\frac{R}{l_{op}}\right) \\ \psi &= \arccos\left(\frac{R}{l_{od}}\right) \\ l_{da} &= \sqrt{{l_{od}}^2 - R^2} \\ l_{ab} &= R\theta \\ l_{bp} &= \sqrt{{l_{op}}^2 - R^2} \\ l_{op} &= ||\vec{p}|| = \sqrt{x^2 + y^2(+z^2)} \\ l_{od} &= ||\vec{x}_d|| = \sqrt{{x_d}^2 + {y_d}^2(+{z_d}^2)} \end{aligned}$$

class `exactpack.solvers.kenamond.kenamond3.`**`Kenamond3`**`(**kwargs)`

Computes the general solution to the Kenamond HE Problem 3.

The inert obstacle is assumed to be a spherical region centered at the origin. The solver will only accept points outside of or on the surface of the inert region for burn time calculation.

Default values are selected to be consistent with the problem definition in *[Kenamond2011]*. Default values are **geometry** $= 2$, $R = 3.0$, $D = 2.0$, $x_d = (0.0, 5.0)$, and $t_d = 0.0$.

Parameters

- **`geometry`** – 2=two-dimensional, 3=three-dimensional
- **`x_d`** – detonator location, enter as a tuple: $(x, y\ [, z])$
- **`R`** – radius of inert obstacle
- **`D`** – detonation velocity of the HE
- **`t_d`** – detonation time

Input evaluation points as an N x **geometry** 2D array of positions: [[x_0, y_0 (, z_0)], [x_1, y_1 (, z_1)], ..., [x_N, y_N (, z_N)]]. These points must be outside of or on the surface of the inert region.

A time value must be input in order to use the ExactPack machinery. The time value is ignored in the calculation.

6.8 The Mader Problem

The Mader rarefaction burn wave.

The Mader Problem is a one-dimensional piston-driven detonation wave with a trailing polytropic rarefaction, in which a slab of high explosive (HE) is initiated on one side and a detonation wave propagates to the other side. The Mader Problem tests a code's ability to compute the evolution of the rarefaction behind the burn front, and the CJ state at the detonation front, assuming the detonation wave propagates through a one-dimensional compressible gas.

The Mader Problem is a special case of the detonation wave solution given on page 24 of *[Fickett1979]*. A detonation wave is driven by a piston moving in the +x direction in a one-dimensional 5 cm slab of gamma-law gas. A rarefaction (i.e., a Taylor wave) follows the detonation front. The Mader Problem solution is given here in the rest frame of the piston. The head of the rarefaction is at the detonation front and the tail is half-way between the front and the piston. For detailed analysis see Timmes, *et al.*, *[Timmes2005]*, and Kirkpatrick, *et al.* *[Kirkpatrick2004]*, See also *[Kamm2008]*.

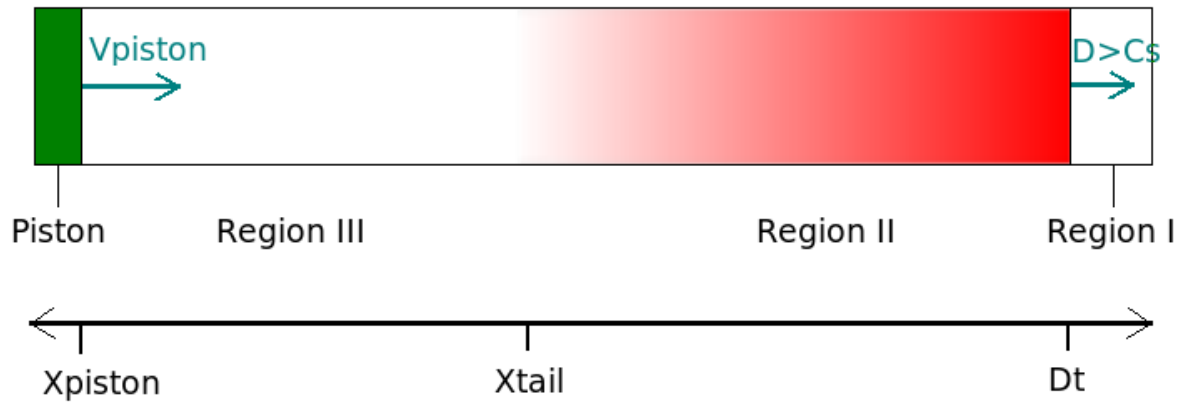

Fig. 6.3: Geometry of the Mader Problem

In the Mader Problem, there is no explicit reaction chemistry, and the reaction zone has zero length and the reaction energy has a fixed value. The CJ detonation speed is given as a parameter, along with the reaction enthalpy (chemical energy released), and the initial state ahead of the burn front. Given these parameters and the assumptions of the model, all of the remaining quantities can be computed analytically. The Hugoniot and CJ state are easily found from the jump conditions and Rayleigh line; the Euler equations yield a self-similar analytic solution for the Taylor wave behind the burn front. This is the essence of the Mader Problem.

The structure of the detonation wave in the slab is divided into three regions:

1. Ahead of the shock front, the unburned gas is in a uniform initial state (Region I).
2. The rarefaction region, proceeding from the the moving shock front to a terminal point exactly half-way between the shock front and the piston. This is a consequence of the polytropic assumption with $\gamma = 3$ (Region II).
3. Between the piston and the rarefaction, the material is in a uniform final state (Region III).

At the final time of $6.25\mu s$, the detonation front has reached the end of the slab, at 5 cm, and the terminal point of the rarefaction will be at 2.5 cm. The Hugoniot relations and final-state isentrope determine the CJ state, which is the state of the reaction products as they exit the detonation front in Region II.

Assuming a γ-law gas, with equation of state (EOS) $p = (\gamma - 1)\rho e$, the CJ state is given by,

$$\begin{aligned} p_{\rm cj} &= \frac{\rho_0 D_{\rm cj}}{(\gamma+1)} \\ \rho_{\rm cj} &= \rho_0 \left(\frac{\gamma+1}{\gamma}\right) \\ c_{\rm cj} &= D_{\rm cj} \left(\frac{\gamma}{\gamma+1}\right) \\ v_{\rm cj} &= \frac{D_{\rm cj}}{\gamma+1} \ . \end{aligned}$$

The specific reaction enthalpy q and the position of the detonation front at time t are given by,

$$\begin{aligned} q &= \frac{D_{\rm CJ}^2}{2(\gamma^2-1)} \\ x_{\rm det} &= D_{\rm cj} t \ . \end{aligned}$$

In Region I, the material is in the constant initial state given above, but, is assumed to be moving in the frame of the piston in the -x direction with speed, $v_0 = -v_{\rm piston}$, where $v_{\rm piston}$ is the speed of the piston in the lab frame. Thus, the initial state in Region I can be chosen to have the following simple form, consistent with the imposed Hugoniot, detonation speed, and chemical energy,

$$\begin{aligned} v &= -v_{\rm piston} \\ p &= 0.0 \\ \rho &= 1.857 \\ c &= 0 \ . \end{aligned}$$

The rarefaction fan in Region II consists of the set of characteristics, $v + c = x/t$. The flow is self-similar here, determined only by the ratio, x/t. The characteristics are bounded between $x/t = D_{\rm cj}$, at the detonation front and, $x/t = D_{\rm cj}/2$, at the tail (for the polytropic case). So, the transition point between tail of the the rarefaction fan and the final state at any time is given by $x_{\rm tail} = (1/2)D_{\rm cj}t$. Then, the self-similar solution for the Taylor wave in Region II is found to be,

$$\begin{aligned} v &= \frac{(2(x/t) - D_{\rm cj})}{(\gamma+1)} \\ p/p_{\rm cj} &= \left[1 + \frac{(\gamma-1)(v - v_{\rm cj})}{2c_{\rm cj}}\right]^{2\gamma/(\gamma-1)} \\ \rho &= \rho_{\rm cj}(p/p_{\rm cj})^{1/\gamma} \\ c &= c_{\rm cj}(p/p_{\rm cj})^{(\gamma-1)/2\gamma} \ . \end{aligned}$$

The constant final state in Region III, equal to the state at the tail of the rarefaction. Thus we have,

$$\begin{aligned} v &= 0 \\ p/p_{\rm cj} &= \left[1 - \frac{v_{\rm cj}(\gamma-1)}{2c_{\rm cj}}\right]^{2\gamma/(\gamma-1)} \\ \rho &= \rho_{\rm cj}(p/p_{\rm cj})^{1/\gamma} \\ c &= c_{\rm cj}(p/p_{\rm cj})^{(\gamma-1)/2\gamma} \ . \end{aligned}$$

To obtain the solutions above in the lab frame, simply subtract $v_{\rm piston}$ from all of the velocities, e.g.,

$$v \rightarrow (v - v_{\rm piston}) \ .$$

One will also need to subtract the initial position of the detonation front from all of the x quantities above if you started the detonation from some position other than the initial position of the piston.

The following table gives the default parameters for Mader Problem. The specific reaction enthalpy, $q = D_{cj}^2/2(\gamma^2 - 1)$, is computed from the values of D_{cj} and γ.

Table 6.1: Parameters for the Mader Problem.

$t_{\rm fin}$	γ	D_{cj}	q
[μs]	[-]	[cm/μs]	[erg/gm]
6.25	3	0.8	4.0 x 10^{10}

Initial conditions in Region I are given in the table below. This table represents a nominal initial state in Region I corresponding to the analytic solution given here. The rarefaction solution should be somewhat (or entirely) insensitive to the initial pressure and sound speed as long as a simulation code is able to sufficiently approximate the given CJ state directly behind the detonation front, so there may be some freedom to deviate somewhat from this prescription, if necessary, for running the Mader Problem on individual codes.

Table 6.2: Initial Conditions for the Mader Problem.

v_0	ρ_0	p_0	c_0
[cm/μs]	[g/cm ^{3}]	[dyn/cm ^{2}]	[cm/μs]
$-v_{piston}$	1.875	0.0	0.0

6.8.1 Theory Discussion

The Mader Problem is based on the simplest HE detonation theory, as outlined in section 2A of Fickett and Davis *[Fickett1979]*. This theory is based on the following assumptions:

1. One-dimensional flow in a simple polytropic gas expansion.
2. The planar detonation front is a jump discontinuity or shock in which the thermodynamic path is the Rayleigh line connecting the initial and final states, as determined by the shock Hugoniot of the unreacted HE, modified by the addition of the full chemical reaction enthalpy.
3. The chemical reaction is assumed to burn to completion instantaneously, so the reaction products are emerge, in equilibrium, from the detonation front.
4. The detonation front motion is assumed to to be steady, so the the state of the material emerging from the shock front is time-independent.

In the simple piston detonation problem, the fuel is assumed to burn instantaneously and completely at the detonation front, which may be treated as idealized shock front, thus the material in the trailing rarefaction is assumed to consist entirely of reaction products. Therefore, the reaction enthalpy - the total heat produced by burning the fuel - is simply added to the energy Hugoniot. Using this to derive the total Hugoniot from the conservation laws, one finds that the slope of the P-V Hugoniot, which is proportional to D_{cj}^2, is only changed by the linear addition of this chemical energy. Equating the expression for the slope of the P-V Hugoniot with the slope of the Rayleigh line one finds that, $(\partial E/\partial v)_{\mathcal{H}} = -p$, on the Hugoniot. But, this relation also obtains on the isentrope of the reaction products, i.e., $(\partial E/\partial v)_{\mathcal{S}} = -p$. Therefore, the CJ state is a special point where the slope is the same for the Rayleigh line, the Hugoniot *and* the reaction product isentrope. This slope is proportional to D_{cj}^2. This implies that D_{cj} is the stable propagation speed for the detonation wave: If the detonation wave propagates a bit *faster* than D_{cj}, the reaction

products will be moving away from the detonation front faster than the local sound speed, so pressure disturbances due to the energy released from the reaction will not be able to support the shock wave. Therefore, the detonation wave should slow down until disturbances from behind "catch up". On the other hand, if the detonation wave travels a bit *slower* than D_{cj}, the sound speed for the reaction products will be greater the detonation speed, so pressure disturbances from the reaction products will overtake it, increasing the shock strength - the pressure and shock speed - until it reaches D_{CJ}, at which point $u_{cj} + c_{cj} = D_{cj}$, i.e, the detonation is exactly sonic in the frame moving with the detonation front.

6.8.2 exactpack.solvers.mader.timmes

A Fortran based Mader solver.

This is a Python interface to the Mader rarefaction solution code from Frank Timmes' website. Timmes' solution code is released under LA-CC-05-101.

For a CJ detonation speed of 0.8 cm/s, it takes 6.25 μ s to travel 5 cm. This time has been hardwired into Timmes' code.

class exactpack.solvers.mader.timmes.**Mader**(***params*)

Computes the solution to the Mader problem.

Parameters

- **d_cj** – Chapman-Jouget density
- **p_cj** – Chapman-Jouget pressure
- **gamma** – ratio of specific heats $\gamma \equiv c_p/c_v$
- **u_piston** – speed of piston

6.9 The Noh Problem

The Noh Problem.

The Noh problem *[Noh1987]* is a self-similar adiabatic compression wave in an ideal gas, and it can be formulated in spherical, cylindrical, or planar geometry. The independent fluid variables are (i) the mass density $\rho(r,t)$, (ii) the velocity of the gas $u(r,t)$, and (iii) the pressure $P(r,t)$, each at spatial location r and time t. Note that in spherical coordinates, $u(r,t)$ is the radial velocity of the gas, and a negative value indicates that gas is flowing in toward the origin. The specific internal energy $e(r,t)$ is related to the other fluid variables by the γ-law equation of state (EOS) for an ideal gas,

$$P = (\gamma - 1)\rho e \ ,$$

where $\gamma \equiv c_p/c_v$ is the ratio of specific heats at constant pressure and volume. The Euler equations for the conservation of mass, momentum, and entropy can be written as:

$$\begin{aligned} \frac{\partial \rho}{\partial t} + u\,\frac{\partial \rho}{\partial r} + \frac{\rho}{r^{k-1}}\,\frac{\partial}{\partial r}\Big(u\,r^{k-1}\Big) &= 0 \\ \frac{\partial u}{\partial t} + u\,\frac{\partial u}{\partial r} + \frac{1}{\rho}\,\frac{\partial P}{\partial r} &= 0 \\ \frac{\partial}{\partial t}\Big(P\rho^{-\gamma}\Big) + u\,\frac{\partial}{\partial r}\Big(P\rho^{-\gamma}\Big) &= 0 \ , \end{aligned}$$

where $k = 1, 2, 3$ is the geometry factor specifying planar, cylindrical, or spherical symmetry, respectively.

This problem is often cast in dimensionless variables in which the gas is moving radially inward with uniform constant velocity $u(r,0) = -1\,\mathrm{cm/s}$ and constant density $\rho(r,0) = 1\,\mathrm{g/cm^3}$. An infinitesimal time after the the gas accumulates at the origin $r = 0$, an infinitely strong stagnation shock forms and starts propagating outwards, leaving behind a region of non-zero pressure and internal energy in its wake. This problem exercises the code's ability to transform kinetic energy into internal energy, and the fidelity with which supersonic flows are tracked *[Timmes2005]* .

The analytic solution is particularly simple, taking the closed form given by *[Gehmeyr1997]* (typos in the paper corrected). Given a uniform inward radial velocity $u(r,0) = -u_0$ (with u_0 positive), and a constant density $\rho(r,0) = \rho_0$, and keeping the gas constant γ general, the solution takes the form:

$$r_{\mathrm{shock}} = \frac{1}{2}\,(\gamma - 1)\,u_0\,t \quad \text{for} \;\; u_0 > 0\,,$$

with the post-shock state given by:

$$\begin{aligned} r < r_{\mathrm{shock}}(t)&: \\ \rho(r,t) &= \left(\frac{\gamma+1}{\gamma-1}\right)^k \rho_0 \\ e(r,t) &= \frac{1}{2}\,u_0^2 \\ u(r,t) &= 0 \\ P(r,t) &= \frac{4^k}{3}\,\rho_0 u_0^2 \end{aligned}$$

and the pre-shock state as:

$$\begin{aligned} r > r_{\mathrm{shock}}(t)&: \\ \rho(r,t) &= \left(1 + \frac{u_0\,t}{x}\right)^{k-1} \rho_0 \\ e(r,t) &= 0 \\ u(r,t) &= -u_0 \quad (u_0 > 0) \\ P(r,t) &= 0 \end{aligned}$$

The geometry factor $k = 1, 2, 3$ corresponds to planar, cylindrically, and spherical geometries, respectively. To cast the equations in terms of dimensionless coordinates, set $u_0 = 1$ and $\rho_0 = 1$.

By default, `exactpack.solvers.noh` loads `exactpack.solvers.noh.noh1`.

6.9.1 exactpack.solvers.noh.noh1

A Noh solver in Python.

This is a pure Python implementation of the Noh solution using Numpy. The package includes specialized classes for specific geometries.

class `exactpack.solvers.noh.noh1.`**Noh**(***kwargs*)

Computes the solution to the general Noh problem.

This is a base class with default values, which can be used to solve the Noh problem for a perfect-gas with any specific heat ratio, γ. It supports one-dimensional planar, two-dimensional cylindrical, and three-dimensional spherical shocks. The default values for the parameters are $\rho_0 = 1$ and $u_0 = 1$, although the user is free to change these parameters. The default geometry is spherical, with $\gamma = 5/3$.

The solver reports jumps in the `exactpack.base.ExactSolution.jumps` attribute of the return value.

Parameters

- **geometry** – 1=planar, 2=cylindrical, 3=spherical
- **rho0** – density
- **u0** – incident velocity (negative)
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.noh.noh1.**PlanarNoh**(***kwargs*)
The standard planar Noh problem.

The planar Noh problem as defined in *[Noh1987]*, with a default value of $\gamma = 5/3$.

Parameters **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.noh.noh1.**CylindricalNoh**(***kwargs*)
The standard cylindrical Noh problem.

The cylindrical Noh problem as defined in *[Noh1987]*, with a default value of $\gamma = 5/3$.

Parameters **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.noh.noh1.**SphericalNoh**(***kwargs*)
The standard spherical Noh problem.

The spherical Noh problem as defined in *[Noh1987]*, with a default value of $\gamma = 5/3$.

Parameters **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

6.9.2 exactpack.solvers.noh.timmes

A Fortran based Noh solver.

This is a Python interface to the Noh solution code from Frank Timmes' website. The code was developed at Los Alamos National Laboratory, and released under LA-CC-05-101. A full description of the solution and code listing is in a Los Alamos report *[Timmes2005]*.

class exactpack.solvers.noh.timmes.**Noh**(***kwargs*)
Computes the solution to the general Noh problem.

The functionality of this class is completely superseded by exactpack.solvers.noh1.Noh, the only difference being that this class is a wrapper to a Fortran library. This version is provided primarily as a template example to developers to demonstrate how to add an exact solution that is computed by an external Fortran code. The default geometry is spherical, with $\gamma = 5/3$. The parameters values for the density and gas velocity are are $\rho_0 = 1$ and $u_0 = 1$.

Parameters

- **geometry** – 1=planar, 2=cylindrical, 3=spherical
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

6.10 The Noh2 Problem

The Noh2 Problem.

6.10.1 The Uniform Collapse problem, a.k.a Noh2

Problem statement

The "uniform collapse" problem, from Noh's JCP (1987), page 92. The fluid is initially distributed uniformly with initial density, internal energy, and velocity prescribed as:

$$\begin{aligned} \rho(r,0) &= \rho_0 \\ e(r,0) &= e_0 \\ u(r,0) &= -r \end{aligned} \tag{6.40}$$

The EOS gives the temperature profile in terms of the kinematic variables.

Problem solution

We note that this problem is identical to a special case of Coggeshall #1 (1991), where the general solution is:

$$\begin{aligned} \rho(r,t) &= \rho_0\, r^b t^{-b-k-1} \\ u(r,t) &= r/t \\ T(r,t) &= T_0\, r^{-b} t^{b-(\gamma-1)(k+1)} \end{aligned} \tag{6.41}$$

where b, k, ρ_0, T_0 are free parameters, and γ is the adiabatic constant for the gas.

Let us multiply the temperature equation by the ideal gas constant R to get specific internal energy $e = RT$, choose the values $b = 0$ and $k = \delta - 1$ (where δ is a geometry factor, equal to 1 for slab, 2 for cylindrical, 3 for spherical), make a variable substitution in time of $t = 1 - \tau$, and make a variable substitution in space of $R = -r$.

Now Eq. (6.41) reduces to:

$$\begin{aligned} \rho(R,\tau) &= \rho_0 (1-\tau)^{-\delta} \\ u(R,\tau) &= -R/(1-\tau) \\ e(R,\tau) &= e_0\, (1-\tau)^{-(\gamma-1)\delta} \end{aligned} \tag{6.42}$$

For the sake of simplicity we substitute t for τ and r for R. Compute pressure using the polytropic EOS $p = \rho e(\gamma - 1)$. We also drop the function call to r on those variables that are independent of it. We now have the complete solution to the Uniform Collapse problem:

$$\begin{aligned} \rho(t) &= \rho_0\, (1-t)^{-\delta} \\ u(r,t) &= -r(1-t)^{-1} \\ e(t) &= e_0\, (1-t)^{-\delta(\gamma-1)} \\ p(t) &= (\gamma-1)\rho_0 e_0\, (1-t)^{-\delta\gamma} \end{aligned} \tag{6.43}$$

It should be noted that the thermodynamic quantities ρ, p, e are uniform over the problem domain. Also, the time domain of the problem is $0 \le t < 1$. Note that at $t = 1$ the solution becomes singular.

Conservation Equations

We will now endeavor to demonstrate that the above equations satisfy the conservation equations.

Euler equations, from Fickett & Davis, page 124

Conservation of mass:

$$\dot{\rho} + \rho \frac{\partial u}{\partial r} = -\frac{\alpha \rho u}{r} \tag{6.44}$$

where $\alpha = 0, 1, 2$ for slab, cylinder, sphere

First we note that $\alpha = \delta - 1$. From Eq. (6.43) we get:

$$\dot{\rho} = \delta \rho_0 (1-t)^{-\delta-1}$$
$$\frac{\partial u}{\partial R} = -(1-t)^{-1}$$

thus, the left hand side of Eq. (6.44) becomes:

$$\delta \rho_0 (1-t)^{-\delta-1} + \rho_0 (1-t)^{-\delta}(-1)/(1-t)$$

$$= \delta \rho_0 (1-t)^{-\delta-1} - \rho_0 (1-t)^{-\delta-1}$$

$$= \rho_0 (1-t)^{-\delta-1}(\delta - 1)$$

$$= \rho_0 (\delta - 1)(1-t)^{-\delta-1}$$

and the right hand side of Eq. (6.44) becomes

$$-\frac{(\delta-1)\rho_0(1-t)^{-\delta}(-r)/(1-t)}{r}$$
$$= \frac{(\delta-1)\rho_0(1-t)^{-\delta}}{(1-t)}$$

$$= \rho_0 (\delta - 1)(1-t)^{-\delta-1}$$

QED: Mass is conserved.

Conservation of momentum:

$$\dot{u} + \frac{1}{\rho}\frac{\partial p}{\partial r} = 0 \tag{6.45}$$

First we compute $\dot{u}$:

$$\dot{u} = \frac{\partial u}{\partial t} + \frac{\partial u}{\partial r}\dot{r}$$

$$\frac{\partial u}{\partial t} = -r(1-t)^{-2}$$
$$\frac{\partial u}{\partial r} = -(1-t)^{-1}$$
$$\dot{r} = u$$

and thus

$$\begin{aligned}\dot{u} &= -r(1-t)^{-2} - (1-t)^{-1}(-r)(1-t)^{-1} \\ &= -r(1-t)^{-2} + r(1-t)^{-2} \\ &= 0\end{aligned}$$

We note that p is not a function of r, and therefore $\frac{\partial p}{\partial r} = 0$. Thus, momentum is conserved.

Conservation of energy:

$$\dot{e} + p\dot{v} = 0 \tag{6.46}$$

where $v = 1/\rho$ (specific volume). First we compute $\dot{e}$ and $\dot{v}$:

$$\begin{aligned}\dot{e} &= \delta(\gamma-1)e_0(1-t)^{-\delta(\gamma-1)-1} \\ \dot{v} = \frac{d(1/\rho)}{dt} &= 1/\rho_0\delta(1-t)^{\delta-1} * (-1) = -\delta/\rho_0(1-t)^{\delta-1}\end{aligned}$$

Substituting into the left hand side of Eq. (6.46) yields:

$$\delta(\gamma-1)e_0(1-t)^{-\delta(\gamma-1)-1} + [(\gamma-1)\rho_0 e_0(1-t)^{-\delta\gamma}] * [-\delta/\rho_0(1-t)^{\delta-1}]$$

which reduces to

$$\begin{aligned}\delta(\gamma-1)e_0(1-t)^{-\delta(\gamma-1)-1} - \delta(\gamma-1)e_0(1-t)^{-\delta\gamma+\delta-1} \\ = 0\end{aligned}$$

QED: Energy is conserved.

Derivation of solution from equations of motion

Assume that the entire fluid collapses to the origin in one time step, with velocity proportional to radius. The velocity solution thus has the form $u = -r(1-t)^{-1}$.

Start with conservation of mass:

$$\dot{\rho} + \rho\frac{\partial u}{\partial r} = -\frac{\alpha\rho u}{r} \tag{6.47}$$

Using the assumed velocity solution, and also $\alpha = \delta - 1$, the components of Eq. (6.47) can be computed as:

$$\begin{aligned}\frac{\partial u}{\partial r} &= -(1-t)^{-1} \\ -\frac{\alpha\rho u}{r} &= (\delta-1)(1-t)^{-1}\rho\end{aligned} \tag{6.48}$$

Substituting Eq. (6.48) into Eq. (6.47) yields:

$$\begin{aligned}\dot{\rho} - (1-t)^{-1}\rho &= (\delta-1)(1-t)^{-1}\rho \\ \dot{\rho} &= \delta(1-t)^{-1}\rho \\ \frac{d\rho}{\rho} &= \frac{\delta dt}{(1-t)} \\ \ln\rho &= -\delta\ln(1-t) + C_1 \\ \rho(t) &= C_2(1-t)^{-\delta}\end{aligned} \tag{6.49}$$

Applying the inital condition $\rho(0) = \rho_0$ yields

$$\rho_0 = C_2 \tag{6.50}$$

Thus the solution for density is:

$$\rho(t) = \rho_0(1-t)^{-\delta} \tag{6.51}$$

Now, solve conservation of energy for the specific internal energy, e:

$$\dot{e} + p\dot{v} = 0 \tag{6.52}$$

Compute the components of Eq. (6.52):

$$\begin{aligned} p = (\gamma-1)\rho e = (\gamma-1)\rho_0(1-t)^{-\delta}e \\ v = \rho^{-1} = \rho_0^{-1}(1-t)^{\delta} \\ \dot{v} = -\rho_0^{-1}\delta(1-t)^{\delta-1} \end{aligned} \tag{6.53}$$

Substituting (6.53) into (6.52) yields:

$$\begin{aligned} \dot{e} + [(\gamma-1)\rho_0(1-t)^{-\delta}e] * [-\rho_0^{-1}\delta(1-t)^{\delta-1}] = 0 \\ \dot{e} = (\gamma-1)\delta(1-t)^{-1}e \\ \frac{de}{e} = \frac{\delta(\gamma-1)dt}{(1-t)} \\ \ln e = -\delta(\gamma-1)\ln(1-t) + C_3 \\ e(t) = C_4(1-t)^{-\delta(\gamma-1)} \end{aligned} \tag{6.54}$$

Applying the inital condition $e(0) = e_0$ yields

$$e_0 = C_4$$

Thus the solution for specific internal energy is:

$$e(t) = e_0(1-t)^{-\delta(\gamma-1)}$$

And thus by the EOS, pressure is:

$$\begin{aligned} p &= (\gamma-1)\rho e \\ &= (\gamma-1)\rho_0(1-t)^{-\delta}e_0(1-t)^{-\delta(\gamma-1)} \\ &= (\gamma-1)\rho_0 e_0(1-t)^{-\delta\gamma} \end{aligned}$$

We now have the complete solution to the Uniform Collapse problem:

$$\begin{aligned} \rho(t) &= \rho_0(1-t)^{-\delta} \\ p(t) &= (\gamma-1)\rho_0 e_0(1-t)^{-\delta\gamma} \\ e(t) &= e_0(1-t)^{-\delta(\gamma-1)} \\ u(r,t) &= -r(1-t)^{-1} \end{aligned}$$

We have already proven that this solution satisfies conservation of momentum.

By default, `exactpack.solvers.noh2` loads `exactpack.solvers.noh2.noh2`.

6.10.2 exactpack.solvers.noh2.noh2

Noh's 2nd solution in Python.

This is a pure Python implementation of the Noh2 solution using Numpy. The package includes specialized classes for specific geometries.

class exactpack.solvers.noh2.noh2.**Noh2**(***kwargs*)
Computes the solution to the general Noh2 problem.

Parameters

- **geometry** – 1=planar, 2=cylindrical, 3=spherical
- **rho0** – initial density
- **e0** – initial internal energy
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.noh2.noh2.**PlanarNoh2**(***kwargs*)
The standard planar Noh2 problem.

The planar Noh2 problem as defined in *[Noh1987]*, with a default value of $\gamma = 5/3$.

Parameters

- **rho0** – initial density
- **e0** – initial internal energy
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.noh2.noh2.**CylindricalNoh2**(***kwargs*)
The standard cylindrical Noh2 problem.

The cylindrical Noh2 problem as defined in *[Noh1987]*, with a default value of $\gamma = 5/3$.

Parameters

- **rho0** – initial density
- **e0** – initial internal energy
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

class exactpack.solvers.noh2.noh2.**SphericalNoh2**(***kwargs*)
The standard spherical Noh2 problem.

The spherical Noh2 problem as defined in *[Noh1987]*, with a default value of $\gamma = 5/3$.

Parameters

- **rho0** – initial density
- **e0** – initial internal energy
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

6.11 The Riemann Solver

The Riemann problem.

This problem exercises the one-dimensional Cartesian Riemann solver for an inviscid, non-heat conducting polytropic gas. The independent fluid variables are (i) the mass density $\rho(r,t)$, (ii) the velocity of the gas $u(r,t)$, and (iii) the

pressure $P(r,t)$, each at spatial location r and time t. The specific internal energy $e(r,t)$ is related to the other fluid variables by the equation of state (EOS) for an ideal gas at constant entropy,

$$P = (\gamma - 1)\rho e \ ,$$

where $\gamma \equiv c_p/c_v$ is the ratio of specific heats at constant pressure and volume. The Euler equations take the form

$$\begin{aligned} \frac{\partial \rho}{\partial t} + u\frac{\partial \rho}{\partial r} + \frac{\rho}{r^{k-1}}\frac{\partial}{\partial r}\Big(ur^{k-1}\Big) &= 0 \\ \frac{\partial u}{\partial t} + u\,\frac{\partial u}{\partial r} + \frac{1}{\rho}\frac{\partial P}{\partial r} &= 0 \\ \frac{\partial}{\partial t}\Big(P\rho^{-\gamma}\Big) + u\,\frac{\partial}{\partial r}\Big(P\rho^{-\gamma}\Big) &= 0 \ , \end{aligned}$$

where $k = 1, 2, 3$ for planar, cylindrical, and spherical coordinates respectively.

The gas starts with constant values of density, velocity, and pressure in two contiguous regions, the boundary between which is located at $x = x_0$:

$$\begin{aligned} &x < x_0 : \\ &\quad \rho(x,0) = \rho_{_L} \\ &\quad u(x,0) = u_{_L} \\ &\quad P(x,0) = P_{_L} \ , \end{aligned} \tag{6.55}$$

and

$$\begin{aligned} &x > x_0 : \\ &\quad \rho(x,0) = \rho_{_R} \\ &\quad u(x,0) = u_{_R} \\ &\quad P(x,0) = P_{_R} \ . \end{aligned} \tag{6.56}$$

We can imagine a membrane at location r_0 separating the left and right regions. At $t = 0$ we remove the membrane, allowing the left and right regions to interact. The resulting wave is a combination of shock waves, rarefactions, and contact discontinuities. Shock waves differ from contact discontinuities in that shocks are discontinuous in density, velocity, and pressure, while contacts are continuous in velocity, but discontinuous in density and pressure.

We shall choose six variants of the Riemann problem to test code algorithms in various physics regimes, as defined in the Tables below. The solution can be computed at any time $t_{\rm fin}$ with adiabatic coefficient γ.

Test Number	Problem Name	r_0	$t_{\rm fin}$	γ
1	Sod	0.5	0.25	7/5
2	Einfeldt	0.5	0.15	7/5
3	Stationary-Contact	0.8	0.012	7/5
4	Slow-Shock	0.5	1.0	7/5
5	Shock-Contact-Shock	0.5	0.3	7/5
6	LeBlanc	0.3	0.5	5/3

For each test case, we choose the initial conditions on either side of the membrane at r_0 to be

Test	ρ_L	u_L	P_L	ρ_R	u_R	P_R
1	1.0	0.0	1.0	0.125	0.0	0.1
2	1.0	-2.0	0.4	1.0	2.0	0.4
3	1.0	-19.59745	10^3	1.0	-19.59745	10^{-2}
4	3.857143	-0.810631	10.33333	1.0	-3.44	1.0
5	1.0	0.5	1.0	1.25	-0.5	1.0
6	1.0	0.0	$(2/3) \times 10^{-1}$	10^{-2}	0.0	$(2/3) \times 10^{-10}$

These problems are implemented in the following helper classes.

class exactpack.solvers.riemann.**Sod**(***params*)

Test1: This is the canonical Sod shock tube with a rarefaction-contact-shock structure. While not a challenging problem, it quickly identifies algorithmic problems resolving basic wave structure.

Parameters

- **pr** – pressure on right in Eq. (6.56)
- **ul** – velocity on left in Eq. (6.55)
- **interface_loc** – initial interface location r_0
- **gammar** – right specific heat ratio $\gamma \equiv c_p/c_v$
- **rhol** – density on left in Eq. (6.55)
- **rhor** – density on right in Eq. (6.56)
- **gammal** – left specific heat ratio $\gamma \equiv c_p/c_v$
- **pl** – pressure on left in Eq. (6.55)
- **ur** – velocity on right in Eq. (6.56)

class exactpack.solvers.riemann.**Einfeldt**(***params*)

Test2: This is the Einfeldt (or 1-2-3) problem, and consists of two strong rarefaction waves, with a near-vacuum between them. Computational methods that conserve total energy might show internal energy errors for this problem.

Parameters

- **pr** – pressure on right in Eq. (6.56)
- **ul** – velocity on left in Eq. (6.55)
- **interface_loc** – initial interface location r_0
- **gammar** – right specific heat ratio $\gamma \equiv c_p/c_v$
- **rhol** – density on left in Eq. (6.55)
- **rhor** – density on right in Eq. (6.56)
- **gammal** – left specific heat ratio $\gamma \equiv c_p/c_v$
- **pl** – pressure on left in Eq. (6.55)
- **ur** – velocity on right in Eq. (6.56)

class exactpack.solvers.riemann.**StationaryContact**(***params*)

Test3: This is the stationary contact problem, and consists of a strong shock wave moving to the right, a stationary contact, and a strong rarefaction moving to the left. This problem is based on the left part of the well-known Woodward-Colella problem, but with the velocity shifted to make the contact stationary. This problem tests an algorithm's dissipation by how much the contact is smeared.

Parameters

- **pr** – pressure on right in Eq. (6.56)
- **ul** – velocity on left in Eq. (6.55)
- **interface_loc** – initial interface location r_0
- **gammar** – right specific heat ratio $\gamma \equiv c_p/c_v$
- **rhol** – density on left in Eq. (6.55)
- **rhor** – density on right in Eq. (6.56)

- **gammal** – left specific heat ratio $\gamma \equiv c_p/c_v$
- **pl** – pressure on left in Eq. (6.55)
- **ur** – velocity on right in Eq. (6.56)

class exactpack.solvers.riemann.**SlowShock**(***params*)

Test4: This is the slow shock problem, and consists of a Mach 3 shock wave moving slowly to the right. Some numerical methods exhibit unphysical oscillations behind the shock.

Parameters

- **pr** – pressure on right in Eq. (6.56)
- **ul** – velocity on left in Eq. (6.55)
- **interface_loc** – initial interface location r_0
- **gammar** – right specific heat ratio $\gamma \equiv c_p/c_v$
- **rhol** – density on left in Eq. (6.55)
- **rhor** – density on right in Eq. (6.56)
- **gammal** – left specific heat ratio $\gamma \equiv c_p/c_v$
- **pl** – pressure on left in Eq. (6.55)
- **ur** – velocity on right in Eq. (6.56)

class exactpack.solvers.riemann.**ShockContactShock**(***params*)

Test5: This is the shock-contact-shock problem. When two shocks separate from the initial state, with a contact between them, errors are produced in all fields, and this problem tests how well an algorithm handles those errors. This problem is similar to the planar Noh problem but with weaker shocks.

Parameters

- **pr** – pressure on right in Eq. (6.56)
- **ul** – velocity on left in Eq. (6.55)
- **interface_loc** – initial interface location r_0
- **gammar** – right specific heat ratio $\gamma \equiv c_p/c_v$
- **rhol** – density on left in Eq. (6.55)
- **rhor** – density on right in Eq. (6.56)
- **gammal** – left specific heat ratio $\gamma \equiv c_p/c_v$
- **pl** – pressure on left in Eq. (6.55)
- **ur** – velocity on right in Eq. (6.56)

class exactpack.solvers.riemann.**LeBlanc**(***params*)

Test6: This is the LeBlanc problem, which is a strong shock, strong rarefaction version of the basic rarefaction-contact-shock problem. This is a good test of a method's robustness.

Parameters

- **pr** – pressure on right in Eq. (6.56)
- **ul** – velocity on left in Eq. (6.55)
- **interface_loc** – initial interface location r_0
- **gammar** – right specific heat ratio $\gamma \equiv c_p/c_v$

- **rhol** – density on left in Eq. (6.55)
- **rhor** – density on right in Eq. (6.56)
- **gammal** – left specific heat ratio $\gamma \equiv c_p/c_v$
- **pl** – pressure on left in Eq. (6.55)
- **ur** – velocity on right in Eq. (6.56)

6.11.1 exactpack.solvers.riemann.kamm

A Fortran based Riemann solver from Jim Kamm.

class exactpack.solvers.riemann.kamm.**Riemann**(***params*)
Computes the solution to the Riemann problem.

Parameters

- **pr** – pressure on right in Eq. (6.56)
- **ul** – velocity on left in Eq. (6.55)
- **gammal** – left specific heat ratio $\gamma \equiv c_p/c_v$
- **gammar** – right specific heat ratio $\gamma \equiv c_p/c_v$
- **rhol** – density on left in Eq. (6.55)
- **rhor** – density on right in Eq. (6.56)
- **interface_loc** – initial interface location r_0
- **pl** – pressure on left in Eq. (6.55)
- **ur** – velocity on right in Eq. (6.56)

6.11.2 exactpack.solvers.riemann.timmes

A Fortran based Riemann solver.

This is a Python interface to the Riemann solution code from Frank Timmes' website.

class exactpack.solvers.riemann.timmes.**Riemann**(***params*)
Computes the solution to the Riemann problem.

Parameters

- **pr** – pressure on right in Eq. (6.56)
- **ul** – velocity on left in Eq. (6.55)
- **pl** – pressure on left in Eq. (6.55)
- **rhol** – density on left in Eq. (6.55)
- **rhor** – density on right in Eq. (6.56)
- **interface_loc** – initial interface location r_0
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$
- **ur** – velocity on right in Eq. (6.56)

6.12 The Reinicke Meyer-ter-Vehn Problem

The Reinicke Meyer-ter-Vehn problem.

The Reinicke Meyer-ter-Vehn (RMTV) problem is defined in *[Reinicke1991]* and *[Kamm2000a]*.

The energy source of the problem considered here is sufficiently large that heat conduction dominates the fluid flow, and a thermal front leads a hydrodynamic shock. The equations of motion are

$$\begin{aligned}
\frac{\partial \rho}{\partial t} + u\frac{\partial \rho}{\partial r} + \frac{\rho}{r^{k-1}}\frac{\partial}{\partial r}\left(r^{k-1}u\right) &= 0 \\
\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial r} + \frac{1}{\rho}\frac{\partial P}{\partial r} &= 0 \\
\frac{\partial e}{\partial t} + u\frac{\partial e}{\partial r} - \frac{P}{\rho^2}\left(\frac{\partial \rho}{\partial t} + u\frac{\partial \rho}{\partial r}\right) &= \frac{1}{\rho r^{k-1}}\frac{\partial}{\partial r}\left(r^{k-1}\chi\frac{\partial T}{\partial r}\right) ,
\end{aligned}$$

where $k = 1, 2, 3$ for planar, cylindrical, or spherical geometry, and the thermal conductivity is parameterized by

$$\chi(\rho, T) = \chi_0\, \rho^a T^b \;\text{ where } a \leq 0 \text{ and } b \geq 1\ . \tag{6.57}$$

The γ-law Equation of State (EOS) for the gas is written

$$P = \Gamma\rho T \;\text{ and }\; e = \frac{\Gamma T}{\gamma - 1}\ , \tag{6.58}$$

where Γ is the Gruneisen coefficient. The initial cold ($T = 0$) density profile is

$$\rho_0(r) = g_0 r^\kappa G(\xi) \;\text{ where } \kappa < 0,$$

with $G(\xi)$ being a dimensionless profile function of the dimensionless self-similarity position variable ξ.

6.12.1 exactpack.solvers.rmtv.timmes

A Fortran based RMTV solver.

This is a Python interface to the Riemann solution code from Frank Timmes website. This code is released under LA-CC-05-101.

class exactpack.solvers.rmtv.timmes.**Rmtv**(***params*)

Computes the solution to the RMTV problem.

Parameters

- **bigamma** – The Gruneisen gass coefficient Γ defined in Eq. (6.58)
- **aval** – power a in the thermal conductivity (6.57)
- **xis** – dimensionless position of the shock front
- **xif** – dimensionless position of the heat front
- **beta0** – eigenvalue of the problem
- **chi0** – coefficient χ_0 in the thermal conductivity (6.57)
- **bval** – power b in the thermal conductivity (6.57)
- **rf** – position of the heat front
- **gamma** – ratio of specific heats $\gamma \equiv c_v/c_p$
- **g0** – heat front scaling parameter

6.13 The Steady Detonation Reaction Zone Problem

The Steady Detonation Reaction Zone Problem.

The steady detonation reaction zone (SDRZ) problem tests reactive high explosives (HE) burn capability, and was first published by *[Fickett1974]*. This problem describes a steady supported Chapman-Jouguet (C-J) detonation with finite reaction-zone thickness. (Supported means that a piston is driving the reaction products from behind at exactly the C-J velocity.) The problem assumes a single forward reaction from species A to species B of the form $A \rightarrow B$, where λ is the mass fraction of species B, so that λ evolves from 0.0 to 1.0 as the detonation progresses.

For specified detonation velocity D, the equations of motion have a solution that is steady in the frame attached to the shock. With the equation-of-state $p(\rho, e, \lambda)$, the Rankine-Hugoniot relations can be solved to obtain $p, \rho,$ and u as functions of λ alone. The reaction rate $r(\rho, e, \lambda)$ is then also a function of λ alone:

$$\frac{d\lambda}{dt} = r(\lambda) \ .$$

The value of x, which is the distance from the shock for a Lagrangian particle that begins at the shock location, is derived from

$$\frac{dx}{dt} = D - u(\lambda) \ .$$

Both species A species B, are described by the polytropic gas equation of state with heat of reaction q,

$$p = (\gamma - 1)\rho(e + \lambda q) \ .$$

6.13.1 General Solution

There exists a general solution, given $D, \rho_0, \gamma, q, \lambda(t)$. A set of constants exist (independent of t),

$$\begin{aligned} {D_j}^2 &= 2(\gamma^2 - 1)q \\ f &= (D/D_j)^2 \\ p_j &= \rho_0 D^2/(\gamma + 1) \\ \rho_j &= \frac{\rho_0(\gamma + 1)}{\gamma} \ , \end{aligned}$$

as well as a set of time-dependent functions,

$$\begin{aligned} g(t) &= \sqrt{(1 - \lambda(t)/f)} \\ p(t) &= f p_j (1 + g(t)) \\ \rho(t) &= \rho_j/(1 - g(t)/\gamma) \\ u(t) &= (1 - \rho_0/\rho(t))D \\ cs(t) &= \sqrt{\gamma p(t)/\rho(t)} \end{aligned}$$

6.13.2 Special case for SDRZ problem

Assume $D = D_j$ and the rate function

$$d\lambda/dt = 2\sqrt{1 - \lambda}$$

and thus

$$\lambda = t(2 - t) \ .$$

Then the constants are

$$p_j = \rho_0 D^2/(\gamma+1)$$
$$\rho_j = \frac{\rho_0(\gamma+1)}{\gamma} ,$$

and the time-dependent functions are

$$g(t) = 1 - t$$
$$p(t) = p_j(2-t)$$
$$\rho(t) = \frac{\rho_j \gamma}{\gamma + t - 1}$$
$$u(t) = (1 - \rho_0/\rho(t))D$$
$$cs(t) = \sqrt{\gamma p(t)/\rho(t)} .$$

Then, for $t \leq 1$:

$$x(t) = \frac{\rho_0 D_j}{\rho_j}\left[\left(1 - \frac{1}{\gamma}\right)t + \frac{t^2}{2\gamma}\right] ;$$

otherwise:

$$x(t) = x|_{t=1} + (t-1)(D - u|_{t=1})$$

Fickett & Rivard suggest comparison at $t = 0.5$ s. Note that in the computational solution, care must be taken to ensure that $\lambda(t)$ is monotonic and does not exceed a value of 1.0.

Test Case

Use results from Table 10.1 of *[Fickett1974]* with:

$$0 \leq t \leq 1.2 \quad \text{every} \quad 0.1$$
$$D = 0.85$$
$$\rho_0 = 1.6$$
$$\gamma = 3 .$$

Results should match:

t	x	p	u	ρ	cs	D	λ
$[\mu s]$	$[cm]$	$[Mbar]$	$[cm/\mu s]$	$[g/cm^3]$	$[cm/\mu s]$	$[cm/\mu s]$	
0.0	0.0000000	0.578000	0.425000	3.200000	0.7361216	0.85	0.00
0.1	0.0435625	0.549100	0.403750	3.047619	0.7352009	0.85	0.19
0.2	0.0892500	0.520200	0.382500	2.909091	0.7324317	0.85	0.36
0.3	0.1370625	0.491300	0.361250	2.782609	0.7277931	0.85	0.51
0.4	0.1870000	0.462400	0.340000	2.666667	0.7212489	0.85	0.64
0.5	0.2390625	0.433500	0.318750	2.560000	0.7127467	0.85	0.75
0.6	0.2932500	0.404600	0.297500	2.461538	0.7022152	0.85	0.84
0.7	0.3495625	0.375700	0.276250	2.370370	0.6895617	0.85	0.91
0.8	0.4080000	0.346800	0.255000	2.285714	0.6746666	0.85	0.96
0.9	0.4685625	0.317900	0.233750	2.206897	0.6573776	0.85	0.99
1.0	0.5312500	0.289000	0.212500	2.133333	0.6375000	0.85	1.00
1.2	0.6587500	0.289000	0.212500	2.133333	0.6375000	0.85	1.00

Note: Corrected value 0.70125 to 0.65875 for x value at t=1.2 (Error in original table and formula)

6.13.3 exactpack.solvers.sdrz.sdrz

Exact solution for the Steady Detonation Reaction Zone problem

class exactpack.solvers.sdrz.sdrz.**SteadyDetonationReactionZone**(***kwargs*)

Computes the solution to the Steady Deonation Reaction Zone (SDRZ) problem. The problem default values are selected to be consistent with the original problem definition in Fickett and Rivard [Fickett1974].

The problem has a rate law defined as $\frac{d\lambda}{dt} = 2\sqrt{1-\lambda}$

Default values are: $D = 0.85cm/\mu s, \rho = 1.6, \gamma = 3.0$

Parameters

- **geometry** – 1=planar
- **D** – Detonation velocity of the HE (cm/us)
- **gamma** – adiabatic index
- **rho_0** – Initial density of the HE (g/cc)

Set default values if necessary, check for valid inputs, and compute polygon vertices for the parameter values.

run_tvec(*tvec*, *useExactLambda=True*)

Evaluates the solution to the Steady Detonation Reaction Zone problem at a given vector of times tvec. If useExactLambda is True, the solution uses an exact formulation of the reaction progress variable, otherwise the rate law is integrated numerically.

6.14 The Sedov Problem

The Sedov Problem.

The Sedov problem.

The Sedov blast wave problem *[Sedov1959] [Kamm2000]* models a rapid point-source explosion in an invisid, non-heat conducting polytropic gas, caused by a release of of energy E_0 at the origin $r = 0$ at time $t = 0$. The independent fluid variables are (i) the mas density $\rho(r,t)$, (ii) the velocity of the gas $u(r,t)$, and (iii) the pressure $P(r,t)$, each at spatial location r and time t. The specific internal energy $e(r,t)$ is related to the other fluid variables by the equation of state (EOS) for an ideal gas at constant entropy,

$$P = (\gamma - 1)\rho e \,,$$

where $\gamma \equiv c_p/c_v$ is the ratio of specific heats at constant pressure and volume. The Euler equations take the form

$$\begin{aligned} \frac{\partial \rho}{\partial t} + u\frac{\partial \rho}{\partial r} + \frac{\rho}{r^{k-1}}\frac{\partial}{\partial r}\left(ur^{k-1}\right) &= 0 \\ \frac{\partial u}{\partial t} + u\,\frac{\partial u}{\partial r} + \frac{1}{\rho}\frac{\partial P}{\partial r} &= 0 \\ \frac{\partial}{\partial t}\left(P\rho^{-\gamma}\right) + u\,\frac{\partial}{\partial r}\left(P\rho^{-\gamma}\right) &= 0 \,, \end{aligned}$$

where $k = 1, 2, 3$ for planar, cylindrical, and spherical coordinates respectively. The initial density $\rho(r,0) = \rho_0$ is taken to be constant in r (with the default value $\rho_0 = 1\,\mathrm{g/cm^3}$), and the initial velocity and pressure $u(r,0) = 0$ and $P(r,0) = 0$ everywhere in space (for $r \neq 0$). At $t = 0^+$, an infinitely strong shock wave forms at the origin, moving radially outward, with energy

$$E_0 = \int dV \left[\frac{1}{2}\,\rho u^2 + \frac{P}{\gamma - 1}\right] ,$$

where the integration runs over the volume behind the shock at time t, i.e. the integration runs over $0 \leq r \leq r_{\rm shock}(t)$. The volume element takes the form $dV = 4\pi\, r^2 dr$ for $k = 3$ (spherical), $dV = 2\pi\, r dr$ for $k = 2$ (cylindrical), and $dV = dr$ for $k = 1$ (planar). We take the gas constant to have the value $\gamma = 7/5 = 1.4$, and for the spherically symmetric case $k = 3$ we also take $E_0 = 0.851072\,\mathrm{erg}$, in which case the shock arrives at $r_{\rm shock} = 1\,\mathrm{cm}$ at the final time $t = 1\,\mathrm{s}$. Similarly, for $k = 2, 1$, we take $E_0 = 0.311357, 0.0673185$, so that the shock arrives at $r_{\rm shock} = 0.75, 0.5$ at $t = 1$, respectively (these conventions allow all three cases to be plotted on the same graph at the final time).

By default, *exactpack.solvers.sedov* attempts to load *exactpack.solvers.sedov.timmes*. If that load fails (due to FORTRAN linking errors, for example), it loads *exactpack.solvers.sedov.doebling*

NOTE: The values of density and specific internal energy produced by the Doebling solver at small values of radius, for the standard Sedov case, are not trustworthy. See the solver documentation for more detail.

NOTE: Currently, the Kamm solver does not return the correct value of the physical variables at the shock location, for at least some cases. The development team recommends not using the Kamm solver until this is resolved. (see unit test exactpack.tests.test_sedov_kamm.TestSedovKammShock)

class exactpack.solvers.sedov.**PlanarSedov**(*interpolation_points=5000*, ***kwargs*)
The standard planar Sedov problem, with a default value of $\gamma = 7/5$ and $E_0 = 0.0673185$ erg.

Parameters **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

Initialization of the Sedov solver class.

Parameters **interpolation_points** (*integer*) – Interpolate the Sedov solution from a representation using this number of points (faster, but not quite as accurate). If 0, then don't do interpolation.

class exactpack.solvers.sedov.**CylindricalSedov**(*interpolation_points=5000*, ***kwargs*)
The standard cylindrical Sedov problem, with a default value of $\gamma = 7/5$ and $E_0 = 0.3113572$ erg.

Parameters **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

Initialization of the Sedov solver class.

Parameters **interpolation_points** (*integer*) – Interpolate the Sedov solution from a representation using this number of points (faster, but not quite as accurate). If 0, then don't do interpolation.

class exactpack.solvers.sedov.**SphericalSedov**(*interpolation_points=5000*, ***kwargs*)
The standard spherical Sedov problem, with a default value of $\gamma = 7/5$ and $E_0 = 0.851072$ erg.

Parameters **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

Initialization of the Sedov solver class.

Parameters **interpolation_points** (*integer*) – Interpolate the Sedov solution from a representation using this number of points (faster, but not quite as accurate). If 0, then don't do interpolation.

6.14.1 exactpack.solvers.sedov.timmes

A Fortran based Sedov solver.

This is a Python interface for the Sedov problem. The Fortran source code was adapted from Frank Timmes website.

class exactpack.solvers.sedov.timmes.**Sedov**(*interpolation_points=5000*, ***kwargs*)
Computes the solution to the Sedov problem.

Parameters

- **geometry** – 1=planar, 2=cylindrical, 3=spherical

- **omega** – initial density power-law exponent, $\rho \equiv \rho_0 r^{-\omega}$
- **eblast** – total amount of energy deposited at the origin at time zero
- **rho0** – initial density
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

Initialization of the Sedov solver class.

Parameters **interpolation_points** (*integer*) – Interpolate the Sedov solution from a representation using this number of points (faster, but not quite as accurate). If 0, then don't do interpolation.

6.14.2 exactpack.solvers.sedov.kamm

A Fortran based Sedov solver.

NOTE: Currently, the Kamm solver does not return the correct value of the physical variables at the shock location, for at least some cases. The development team recommends not using the Kamm solver until this is resolved. (see unit test exactpack.tests.test_sedov_kamm.TestSedovKammShock)

This is Python interface for the Sedov problem. The Fortran source is Jim Kamm's original Fortran Sedov solver, from which Timmes' source code was based. The Kamm code also provides output in dimensionless self similar variables λ, V, f, g, h. To access the selfsimilar variables: solver = Sedov() followed by solution = solver.self_similar(lam, t), where lam is a spatial array in dimensionless selfsimilar form.

The selfsimilar output has been verified against the values reported in Sedov's book and Tables 1, 2, and 3 of the paper LA-UR-00-6055. Timmes' code has been chosen for the default, as it has better behavior in certain asymptotic regimes.

class exactpack.solvers.sedov.kamm.**Sedov**(*interpolation_points=5000*, ***kwargs*)

Computes the solution to the Sedov problem.

Parameters

- **geometry** – 1=planar, 2=cylindrical, 3=spherical
- **omega** – initial density power-law exponent, $\rho \equiv \rho_0 r^{-\omega}$
- **eblast** – total amount of energy deposited at the origin at time zero
- **rho0** – initial density
- **gamma** – specific heat ratio $\gamma \equiv c_p/c_v$

Initialize the Sedov solver class.

Parameters **interpolation_points** (*integer*) – interpolate the Sedov solution from a representation using this number of points (faster, but not quite as accurate). If 0, then don't do interpolation.

6.14.3 exactpack.solvers.sedov.doebling

A Python implementation of the Timmes Sedov solver in double precision.

This is a pure Python implentation of the Timmes Sedov solver in ExactPack. It uses SciPy optimization to find values of v that minimize the error in lambda, and SciPy integration to evaluate the energy integrals.

This solver uses double precision (64 bit), while the Timmes and Kamm solvers use quad precision (128 bit). At small values of radius, the changes in the similarity variable become too small to track with double precision arithmetic. Therefore, the values of the physical variables are interpolated in that regime. The practical effect of this is that the

density and specific internal energy values, for the Standard Sedov case at small radius, are less accurate in this solver than in the Kamm or Timmes solvers, and should not be trusted. It may be, however, that the impact of this solution inaccuracy on a global error norm is low, due to the magnitude of the solution at small radius being so much smaller than the magnitude of the solution at larger radius. An asymptotic solution for the Sedov functions at small values of radius may be able to mitigate this problem and should be developed.

class `exactpack.solvers.sedov.doebling.`**`Sedov`**(***kwargs*)

Computes the solution to the Sedov problem. The solver reports the shock jump conditions in the *exactpack.base.ExactSolution.jumps* attribute of the return value.

Parameters

- **`geometry`** – 1=planar, 2=cylindrical, 3=spherical
- **`rho0`** – initial density
- **`omega`** – initial density power-law exponent, $\rho \equiv \rho_0 r^{-\omega}$
- **`gamma`** – specific heat ratio $\gamma \equiv c_p/c_v$
- **`eblast`** – total amount of energy deposited at the origin at time zero

`sedov_funcs_standard`(*v*)

Given similarity variable v, compute Sedov functions: f, g, h, λ, and $\frac{d\lambda}{dv}$

`efun01`(*v*)

Integrand for first Sedov energy integral

`efun02`(*v*)

Integrand for second Sedov energy integral

`physical`(*f_fun*, *g_fun*, *h_fun*)

Returns physical variables from single values of Sedov functions

6.15 The Su-Olson Problem

The Su-Olson Problem.

The Su-Olson problem *[Su1996]* is a one-dimensional, half-space, non-Equilibrium Marshak burn wave. The radiative transfer model is a one-group diffusion approximation with a Marshak radiation boundary condition. The radiation temperature field (in energy units) and the matter energy density field are denoted by $T = T(z,t)$ and $E = E(z,t)$ for $0 \leq z < \infty$. Fluid motion is neglected, and the equations of motion become

$$\begin{aligned} \frac{\partial E}{\partial t} - \frac{\partial}{\partial z}\left[\frac{c}{3\kappa(T)}\,\frac{\partial E}{\partial z}\right] &= c\,\kappa(T)\Big[a\,T^4 - E\Big] \\ c_v(T)\,\frac{\partial T}{\partial t} &= c\,\kappa(T)\Big[E - a\,T^4\Big], \end{aligned}$$

where $\kappa(T)$ is the opacity of the material, c is the speed of light, a is the radiation constant, and $c_v(T)$ is the specific heat at constant volume. The equations can be solved for the (non-physical) model in which

$$\begin{aligned} c_v(T) &= \alpha\,T^3 \\ \kappa(T) &= \kappa_0\ . \end{aligned} \tag{6.59}$$

When the matter radiates as a black body, so that $E = aT^4$, we see that $c_v = \partial E/\partial T = 4aT^3$; therefore the coefficient α is related to the radiation constant by

$$\alpha = 4a\ .$$

The Marshak boundary condition at $z = 0$ is

$$E(0,t) - \frac{2}{3\kappa(0,t)} \frac{\partial E(0,t)}{\partial z} = \frac{4}{c} F_{\text{in}},$$

where F_{in} is incident radiation energy flux incident on the $z = 0$ surface, and the $z \to \infty$ boundary condition is $E(\infty, t) = 0$. The initial conditions are $E(z, 0) = T(z, 0) = 0$.

6.15.1 exactpack.solvers.suolson.timmes

A Fortran based Su-Olson solver.

This is a Python interface to the Riemann solution code suo02.f from Frank Timmes' website. This code is released under LA-CC-05-101.

class exactpack.solvers.suolson.timmes.**SuOlson**(***kwargs*)

Computes the solution to the Su-Olson problem.

Parameters

- **alpha** – coefficient α in Eq. (6.59) for the specific heat ($\alpha = 4a$)
- **trad_bc_ev** – radiation boundary condition, i.e. $T(0,t) =$ trad_bc_ev
- **opac** – constant opacity κ_0 in Eq. (6.59)

Initialize the Su-Olson solver class.

CHAPTER

SEVEN

TESTING

This chapter documents the internal self-tests of the library. The tests are all implemented as Python unittests, and can be run using the standard unittest framework. The easiest way to build and test the library is through the setup.py script:

```
python setup.py test
```

7.1 The Blake Problem

Unittests for the spherical, isotropic, linear elastic Blake solver in exactpack.solvers.Blake.

class exactpack.tests.test_blake.**TestBlakeParamErrWarnChecks**(*methodName='runTest'*)
Test Blake instance parameter checks.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_defaults()
Test default param values are present in the default solver.

test_assign_non_elastic_params()
Test that the non-elastic parameters are passed to the instance.

test_check_cavity_radius()
Test cavity_radius parameter positive check.

test_check_geometry()
Test geometry parameter range check.

test_check_pressure_scale_pos()
Test pressure_scale parameter positive check.

test_check_ref_density()
Test ref_density parameter positive check.

test_pscale_warn_check()
Test the pressure_scale parameter range warning.

test_check_blake_debug()
Test that blake_debug parameter is boolean.

class exactpack.tests.test_blake.**TestBlakeSolution**(*methodName='runTest'*)
Tests exactpack.blake.Blake solver.

These tests confirm proper solution values in some specific cases.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_blake_dflt_regress()
Regression test of default solver instance.

Compares current output on `TestBlakeSolution.intl_test_grid` (three points) using the default solver instance against regression standard output, `TestBlakeSolution.blk_dflt_std`.

test_blake_non_dflt_regress()
Regression test of solver instance with param values given.

Same as test_blake_dflt_regress() but a full set of default parameter values are supplied.

class exactpack.tests.test_blake.**TestBlakeRunChecks**(*methodName='runTest'*)
Test `Blake` run-time checks.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_radii_positive_check()
Test execution radial coordinates positive check.

class exactpack.tests.test_blake.**TestBlakeVsKamm**(*methodName='runTest'*)
Test the *exactpack.solvers.blake.blake.Blake* solver against J. Kamm's F90 Blake solver output.

J. Kamm's Fortran90 spherical Blake solver[2] has been run using our default parameter set, Brock[1] on a 16-point radial grid. This test runs exactly the same problem on the same grid at 64-bit precision. Our grid differs from that of Brock in that (SI units):

- Our domain extends to 1.2 rather than 1.0.
- Our grid spacing is significantly larger: $dr = 0.075$.

The grid is "cell-center" in that the points are offset from domain the boundaries by $dr/2 = 0.0375$. A feature of the Brock setup is that the radius of the leading edge of the waveform is at exactly $r = 0.9$ for the snapshot at $t = 0.00016$.

The Kamm solver output fields include: position, displacement, stress_rad, stress_hoop, pressure, stress_diff = abs(stress_rad - stress_hoop); it does not separately calculate strains. The Kamm fields are incorporated as source here and each field is also computed by the *ExactPack* Blake solver (EPB). This test demonstrates that to within a small tolerance, the two solvers agree. Kamm's solver was developed independently of EPB, in a different source language and using slightly different analytic forms for the stresses.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_blake_vs_kamm()
Test spherical Blake fields and compare against Kamm output.

7.2 The Cogshell Problems

Tests for the Cog1 problem.

class exactpack.tests.test_cog.**TestCog1**(*methodName='runTest'*)
Test for Coggeshall problems: cog1

[2] Kamm, James R., and Lee A. Ankeny. *Analysis of the Blake Problem with RAGE*. Report LA-UR-09-01255, Los Alamos National Laboratory (LANL), Los Alamos, NM 87545: LANL (2009).

[1] Brock, Jerry S. *Blake Test Problem Parameters*. Report LA-UR-08-3005, Los Alamos National Laboratory (LANL), Los Alamos, NM 87545: LANL (2008).

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_cog1()
cog1 problem:

test_geometry_error_cog1()
cog1 problem:

class exactpack.tests.test_cog.**TestCog2**(*methodName='runTest'*)
Test for Coggeshall problems: cog2

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_cog2()
cog2 problem:

test_geometry_error_cog2()
cog2 problem:

class exactpack.tests.test_cog.**TestCog3**(*methodName='runTest'*)
Test for Coggeshall problems: cog3

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_cog3()
cog3 problem:

test_geometry_error_cog3()
cog3 problem:

class exactpack.tests.test_cog.**TestCog4**(*methodName='runTest'*)
Test for Coggeshall problems: cog4

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_cog4()
cog4 problem:

test_geometry_error_cog4()
cog4 problem:

class exactpack.tests.test_cog.**TestCog5**(*methodName='runTest'*)
Test for Coggeshall problems: cog5

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_cog5()
cog5 problem:

test_geometry_error_cog5()
cog5 problem:

class exactpack.tests.test_cog.**TestCog6**(*methodName='runTest'*)
Test for Coggeshall problems: cog6

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_cog6()
cog6 problem:

test_geometry_error_cog6()
cog6 problem:

class exactpack.tests.test_cog.**TestCog7**(*methodName='runTest'*)
Test for Coggeshall problems: cog7

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_cog7()
cog7 problem:

test_geometry_error_cog7()
cog7 problem:

class exactpack.tests.test_cog.**TestCog8**(*methodName='runTest'*)
Test for Coggeshall problems: cog8

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_cog8()
cog8 problem:

test_geometry_error_cog8()
cog8 problem:

class exactpack.tests.test_cog.**TestCog9**(*methodName='runTest'*)
Test for Coggeshall problems: cog9

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_cog9()
cog9 problem:

test_geometry_error_cog9()
cog9 problem:

class exactpack.tests.test_cog.**TestCog10**(*methodName='runTest'*)
Test for Coggeshall problems: cog10

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_cog10()
cog10 problem:

test_geometry_error_cog10()
cog10 problem:

class exactpack.tests.test_cog.**TestCog11**(*methodName='runTest'*)
Test for Coggeshall problems: cog11

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_cog11()
cog11 problem:

test_geometry_error_cog11()
cog11 problem:

class exactpack.tests.test_cog.**TestCog12**(*methodName='runTest'*)
Test for Coggeshall problems: cog12

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_cog12()
cog12 problem:

test_geometry_error_cog12()
cog12 problem:

class exactpack.tests.test_cog.**TestCog13**(*methodName='runTest'*)
Test for Coggeshall problems: cog13

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_cog13()
cog13 problem:

test_geometry_error_cog13()
cog13 problem:

class exactpack.tests.test_cog.**TestCog14**(*methodName='runTest'*)
Test for Coggeshall problems: cog14

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_cog14()
cog14 problem:

test_geometry_error_cog14()
cog14 problem:

class exactpack.tests.test_cog.**TestCog16**(*methodName='runTest'*)
Test for Coggeshall problems: cog16

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_cog16()
cog16 problem:

test_geometry_error_cog16()
cog16 problem:

class exactpack.tests.test_cog.**TestCog17**(*methodName='runTest'*)
Test for Coggeshall problems: cog17

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_cog17()
cog17 problem:

test_geometry_error_cog17()
cog17 problem:

class exactpack.tests.test_cog.**TestCog18**(*methodName='runTest'*)
Test for Coggeshall problems: cog18

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_cog18()
cog18 problem:

test_geometry_error_cog18()
cog18 problem:

class exactpack.tests.test_cog.**TestCog19**(*methodName='runTest'*)
Test for Coggeshall problems: cog19

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_cog19()
cog19 problem:

test_geometry_error_cog19()
cog19 problem:

class exactpack.tests.test_cog.**TestCog20**(*methodName='runTest'*)
Test for Coggeshall problems: cog20

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_cog20()
cog20 problem:

test_geometry_error_cog20()
cog20 problem:

class exactpack.tests.test_cog.**TestCog21**(*methodName='runTest'*)
Test for Coggeshall problems: cog21

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_cog21()
cog21 problem:

Tests for the Cog8 problem.

class exactpack.tests.test_cog8.**TestCog8**(*methodName='runTest'*)
Test for exactpack.cog.cog8_timmes.Cog8.

The tests consist of comparing the values against the exact solution at $r = 0.2$ and $t = 2.0$.

Note: Cog8 currently crashes if the spatial interval is a single point; therefore, these tests really use $r = [0.2, 0.3]$ and the the first spatial point is selected. This is because Timmes' solution code returns cell averaged values. Fix this.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_density_timmes()
cog8 problem: density timmes

test_density()
cog8 problem: density

test_temperature_timmes()
cog8 problem: temperature timmes

test_temperature()
cog8 problem: temperature

test_sie_timmes()
cog8 problem: sie timmes

test_sie()
cog8 problem: sie

test_pressure_timmes()
cog8 problem: pressure timmes

test_pressure()
cog8 problem: pressure

test_velocity_timmes()
cog8 problem: velocity timmes

test_velocity()
cog8 problem: velocity

test_geometry_error()
Cog8 Problem: Test for valid value of geometry

7.3 The DSD Problems

Tests for the DSD verification problems exact solution solvers.

Test problems consist of comparison of the calculated burntime for a 2D point with the known analytic solution. Solver parameter inputs are tested individually, both for validity and for new solutions.

A time value is passed to the ExactPack solver in order to maintain a consistent format of input with the other solvers. Because the burn time solution is not time dependent, this value is ignored by the solver.

class exactpack.tests.test_dsd.**TestRateStick**(*methodName='runTest'*)
Tests for *exactpack.solvers.dsd.ratestick.RateStick*.

Solution tests consist of comparing the calculated burn time to the analytic solution at a fixed point. Input tests check that invalid input raises the appropriate error expression.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_burntime_IC1_planar()
Tests burntime solution for $IC = 1$ at 2D point.

Uses default parameter values for all parameters (except $xnodes$ and $ynodes$).

test_burntime_IC1_cylindrical()
Tests burntime solution for $IC = 1$ at 2D point.

Uses default parameter values for all parameters (except $xnodes$ and $ynodes$).

test_burntime_IC2_planar()
Tests burntime solution for $IC = 2$ at 2D point.

Uses default parameter values for all parameters (except $xnodes$ and $ynodes$).

test_burntime_IC3_planar()
Tests burntime solution for $IC = 2$ at 2D point.

Uses default parameter values for all parameters (except $xnodes$ and $ynodes$).

test_xylist_matches_xnodes_ynodes(**args*, ***kwargs*)
Test for valid combination of $xnodes$, $ynodes$ and size of $xylist$.

test_xylist_matches_R(**args*, ***kwargs*)
Test for $xylist$ nodes at $x = R$.

test_xylist_matches_zero(**args*, ***kwargs*)
Test for $xylist$ nodes at $x = 0$.

test_geometry_error()
Test for valid value of geometry.

test_radius_neg_error()
Test for valid value of rate stick radius, R.

test_radius_zero_error()
Test for valid value of rate stick radius, R.

test_omega_C_neg_error()
Test for valid value of DSD edge angle, $omega_c$.

test_omega_C_zero_error()
Test for valid value of DSD edge angle, $omega_c$.

test_omega_C_big_error()
Test for valid value of DSD edge angle, $omega_c$.

test_omega_C_top_error()
Test for valid value of DSD edge angle, $omega_c$.

test_D_CJ_neg_error()
Test for valid value of HE detonation velocity, D.

test_D_CJ_zero_error()
Test for valid value of HE detonation velocity, D.

test_alpha_neg_error()
Test for valid value of det velocity deviance coefficient, $alpha$.

test_IC_error()
Test for valid value of initial condition, IC.

test_IC1limit_error()
Test for valid value of detonation radius, r_d, if IC=1.

test_t_f_neg_error()
Test for valid value of final time, t_f.

test_t_f_zero_error()
Test for valid value of final time, t_f.

test_xnodes_neg_error()
Test for valid value of number of x-nodes, $xnodes$.

test_xnodes_zero_error()
Test for valid value of number of x-nodes, $xnodes$.

test_ynodes_neg_error()
Test for valid value of number of y-nodes, $ynodes$.

test_ynodes_zero_error()
Test for valid value of number of y-nodes, $ynodes$.

class exactpack.tests.test_dsd.**TestCylindricalExpansion**(*methodName='runTest'*)
Tests for *exactpack.solvers.dsd.cylexpansion.CylindricalExpansion*.

Solution tests consist of comparing the calculated burn time to the analytic solution at a fixed point. Input tests check that invalid input raises the appropriate error expression.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_burntime_base_HE1()
Tests burntime solution at point in HE1 region.

Uses default parameter values.

test_burntime_base_HE2()
Tests burntime solution at point in HE2 region.

Uses default parameter values.

test_burntime_base_indet()
Tests burntime solution at point in HE2 region.

Uses default parameter values.

test_burntime_1()
Tests burntime solution at points in each HE region.

Uses default parameter values for r_2, D_{CJ_1}, D_{CJ_2}, $alpha_1$, $alpha_2$, t_d. The inner radius of HE1 is $r_1 = 1.2$.

test_burntime_2()
Tests burntime solution at points in each HE region.

Uses default parameter values for r_1, D_{CJ_1}, D_{CJ_2}, $alpha_1$, $alpha_2$, t_d. The radius of interface between HE1 and HE2 is $r_2 = 2.5$.

test_burntime_DCJ1()
Tests burntime solution at points in each HE region.

Uses default parameter values for r_1, r_2, D_{CJ_2}, $alpha_1$, $alpha_2$, t_d. The CJ detonation velocity of HE1 is $D_{CJ_1} = 1.0$.

test_burntime_DCJ2()
Tests burntime solution at points in each HE region.

Uses default parameter values for r_1, r_2, D_{CJ_1}, $alpha_1$, $alpha_2$, t_d. The CJ detonation velocity of HE1 is $D_{CJ_2} = 0.5$.

test_burntime_alpha1()
Tests burntime solution at points in each HE region.

Uses default parameter values for r_1, r_2, D_{CJ_1}, D_{CJ_2}, $alpha_2$, t_d. The CJ detonation velocity of HE1 is $alpha_1 = 0.05$.

test_burntime_alpha2()
Tests burntime solution at points in each HE region.

Uses default parameter values for r_1, r_2, D_{CJ_1}, D_{CJ_2}, $alpha_1$, t_d. The CJ detonation velocity of HE1 is $alpha_2 = 0.05$.

test_burntime_tdet()
Tests burntime solution at points in each HE region.

Uses default parameter values for r_1, r_2, D_{CJ_1}, D_{CJ_2}, $alpha_1$, $alpha_2$. Detonation time of HE1 is $t_d = 3.0$.

test_geometry_error()
Test for valid value of geometry.

test_r1_neg_error()
Test for valid value of inner radius, r_1.

test_r1_zero_error()
Test for valid value of inner radius, r_1.

test_r2_neg_error()
Test for valid value of HE interface radius, r_2.

test_r2_zero_error()
Test for valid value of HE interface radius, r_2.

test_r2_smaller_error()
Test for valid value of HE interface radius, r_2.

test_DCJ1_neg_error()
Test for valid value of inner region HE CJ detonation velocity, D_{CJ_1}.

test_DCJ1_zero_error()
Test for valid value of inner region HE CJ detonation velocity, D_{CJ_1}.

test_DCJ2_neg_error()
Tests for valid value of outer region HE CJ detonation velocity, D_{CJ_2}.

test_DCJ2_zero_error()
Tests for valid value of outer region HE CJ detonation velocity, D_{CJ_2}.

test_alpha1_neg_error()
Test for valid value of inner region det velocity deviance coefficient, $alpha_1$.

test_alpha2_neg_error()
Test for valid value of outer region det velocity deviance coefficient, $alpha_2$.

class exactpack.tests.test_dsd.**TestExplosiveArc**(*methodName='runTest'*)
Tests for *exactpack.solvers.dsd.explosivearc.ExplosiveArc*.

Solution tests consist of comparing the calculated burn time to the analytic solution at a fixed point. Input tests check that invalid input raises the appropriate error expression.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_burntime_default()
Tests burntime solution at 2D point.

Uses default parameter values for all parameters (except $xnodes$ and $ynodes$).

test_xylist_matches_xnodes_ynodes(**args*, ***kwargs*)
Test for valid combination of $xnodes$, $ynodes$ and size of $xylist$.

test_HEx_positive(**args*, ***kwargs*)
Test for HE nodes with negative x-component‘.

test_HE_radius_zero(**args*, ***kwargs*)
Test for $xylist$ nodes at $r = 0$.

test_HE_radius_small(**args*, ***kwargs*)
Test that $xylist$ contains nodes at $r = r_2$.

test_HE_radius_large(**args*, ***kwargs*)
Test that $xylist$ contains nodes at $r = r_1$.

test_HE_theta_neg(**args*, ***kwargs*)
Test that $xylist$ contains nodes at theta = - pi/2

test_HE_theta_pos(**args*, ***kwargs*)
Test that $xylist$ contains nodes at theta = pi/2

test_geometry_error()
Test for valid value of geometry.

test_r1_neg_error()
Test for valid value of inner radius, r_1.

test_r1_zero_error()
Test for valid value of inner radius, r_1.

test_r2_neg_error()
Test for valid value of outer radius, r_2.

test_r2_zero_error()
Test for valid value of outer radius, r_2.

test_r2_smaller_error()
Test for valid value of outer radius, r_2.

test_omegain_neg_error()
Test for valid value of inner edge angle, ω_{in}.

test_omegain_zero_error()
Test for valid value of inner edge angle, ω_{in}.

test_omegain_max_error()
Test for valid value of inner edge angle, ω_{in}.

test_omegain_max2_error()
Test for valid value of inner edge angle, ω_{in}.

test_omegaout_min_error()
Test for valid value of outer edge angle, ω_{out}.

test_omegaout_max_error()
Test for valid value of outer edge angle, $\omega_s out$.

test_detloc_zero_error()
Tests for valid location of detonator, x_d.

test_detloc_pos_error()
Tests for valid location of detonator, x_d.

test_D_CJ_neg_error()
Test for valid value of HE detonation velocity, D_{CJ}.

test_D_CJ_zero_error()
Test for valid value of HE detonation velocity, D_{CJ}.

test_alpha_neg_error()
Test for valid value of det velocity deviance coefficient, $alpha$.

test_t_f_neg_error()
Test for valid value of final time, t_f.

test_t_f_zero_error()
Test for valid value of final time, t_f.

test_xnodes_neg_error()
Test for valid value of number of x-nodes, $xnodes$.

test_xnodes_zero_error()
Test for valid value of number of x-nodes, $xnodes$.

test_ynodes_neg_error()
Test for valid value of number of y-nodes, $ynodes$.

test_ynodes_zero_error()
Test for valid value of number of y-nodes, $ynodes$.

7.4 The Escape of HE Products Problem

class exactpack.tests.test_ehep.**TestEHEPAssignments**(*methodName='runTest'*)
Tests exactpack.solvers.ehep.EscapeOfHEProducts.

These tests confirm proper assignment of variables, including default values

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

class exactpack.tests.test_ehep.**TestEhepSolution**(*methodName='runTest'*)
Tests exactpack.ehep.EscapeOfHEProducts.

These tests confirm proper solution values for specific cases

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_point_on_boundary_1()
Test 1 for point_on_boundary function

test_point_on_boundary_2()
Test 2 for point_on_boundary function

test_point_on_boundary_3()
Test 3 for point_on_boundary function

test_point_on_boundary_4()
Test 4 for point_on_boundary function

test_point_on_boundary_5()
Test 5 for point_on_boundary function

test_point_on_boundary_6()
Test 6 for point_on_boundary function

test_point_on_boundary_7()
Test 7 for point_on_boundary function

test_corners1()
Tests proper calculation of polygon corners by comparing to results from hand calculations

test_corners2()
Tests proper calculation of polygon corners by comparing to results from hand calculations

test_0_regions()
Tests proper evaluation of point locations inside regions 00, 0V, and 0H, and outside of all regions

test_fickett_table6_1()
Tests proper solution values by comparing default case to the Table 6.1 in Fickett & Rivard

test_fickett_table6_2()
Tests proper solution values by comparing default case to the Table 6.2 in Fickett & Rivard

7.5 Heat Conduction Problems

Test heat conduction problems.

class exactpack.tests.test_heat.**TestHeatCylindricalSandwich**(*methodName='runTest'*)
Test Heat Cylindrical Sandwich

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_heat_cylindrical_sandwich_modes()
Checks the modes numbers for CylindricalSandwich

test_heat_cylindrical_sandwich_Rnm()
Checks the modes Rnm.

class exactpack.tests.test_heat.**TestHeatRod1d**(*methodName='runTest'*)
Test Heat Rod1D

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_heat_rod1d_boundary_conditions()
Checks Boundary Conditions for Rod1D in rod1d.py

test_heat_rod1d_regression()
Simple regression test for Rod1D in rod1d.py

class exactpack.tests.test_heat.**TestRod1DFunctions**(*methodName='runTest'*)
Test function definitions in PlanarSandwich

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_heat_rod1d_modes()
Checks the mode numbers and coefficients in PlanarSandwich

class exactpack.tests.test_heat.**TestHeatPlanarSandwich**(*methodName='runTest'*)
Test Heat Planar Sandwich

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_heat_planar_sandwich_regression()
Simple regression test for the planar sandwich

class exactpack.tests.test_heat.**TestHeatPlanarHutchens**(*methodName='runTest'*)
Test Hutchens Solutions

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_heat_planar_hutchens1_regression()
Simple regression test for Hutchens1

test_heat_planar_hutchens2_regression()
Simple regression test for Hutchens2

7.6 The Kenamond Problems

Tests for the Kenamond exact solution solvers.

Test problems consist of comparison of the calculated burntime for a 2D or 3D point with the known analytic solution. Solver parameter inputs are tested individually, both for validity and for new solutions.

A time value is passed to the ExactPack solver in order to maintain a consistent format of input with the other solvers. Because the burn time solution is not time dependent, this value is ignored by the solver.

class exactpack.tests.test_kenamond.**TestKenamond1**(*methodName='runTest'*)
Tests for *exactpack.solvers.kenamond.kenamond1.Kenamond1*.

Solution tests consist of comparing the calculated burn time to the analytic solution at a fixed point. Input tests check that invalid input raises the appropriate error expression.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_burntime_2d_base()
Tests burntime solution at 2D point.

Uses default parameter values for all parameters.

test_burntime_3d_base()
Tests burntime solution at 3D point (**geometry** $= 3$).

Uses default parameter values for HE detonation velocity and detonation time. Detonator location is specified as the 3D origin.

test_burntime_2d_detvel()
Tests burntime solution at 2D point.

Uses default parameter values for detonator location and detonation time. HE detonation velocity is $D = 2.0$.

test_burntime_3d_detvel()
Tests burntime solution at 3D point (**geometry** $= 3$).

Uses default parameter value for detonation time. Detonator location is specified as the 3D origin. HE detonation velocity is $D = 2.0$.

test_burntime_2d_dettime()
Tests burntime solution at 2D point.

Uses default parameter values for HE detonation velocity and detonator location. Detonation time is $t_d = -2.0$.

test_burntime_3d_dettime()
Tests burntime solution at 3D point (**geometry** $= 3$).

Uses default parameter value for HE detonation velocity. Detonator location is specified as the 3D origin. Detonation time is $t_d = -2.0$.

test_burntime_2d_detloc()
Tests burntime solution at 2D point.

Uses default parameter values for HE detonation velocity and detonation time. Detonator location is $x_d = (1.0, 1.0)$.

test_burntime_3d_detloc()
Tests burntime solution at 3D point (**geometry** $= 3$).

Uses default parameter values for HE detonation velocity and detonation time. Detonator location is $x_d = (1.0, 1.0, 1.0)$.

test_geometry_error()
Test for valid value of geometry.

test_D_neg_error()
Test for valid value of HE detonation velocity, D.

test_D_zero_error()
Test for valid value of HE detonation velocity, D.

test_detspec_2d_error()
Test for valid geometry of detonator, x_d.

test_detspec_3d_error()
Test for valid geometry of detonator, x_d.

class exactpack.tests.test_kenamond.**TestKenamond2**(*methodName='runTest'*)
Tests for *exactpack.solvers.kenamond.kenamond2.Kenamond2*.

Solution tests consist of comparing the calculated burn time to the analytic solution at a fixed point. Input tests check that invalid input raises the appropriate error expression.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_burntime_2d_base_t1()
Tests burntime solution at 2D point in t_1 region.

Uses default parameter values.

test_burntime_2d_base_t2()
Tests burntime solution at 2D point in t_2 region.

Uses default parameter values.

test_burntime_2d_base_t3()
Tests burntime solution at 2D point in t_3 region.

Uses default parameter values.

test_burntime_2d_base_t4()
Tests burntime solution at 2D point in t_4 region.

Uses default parameter values.

test_burntime_2d_base_t5()
Tests burntime solution at 2D point in t_5 region.

Uses default parameter values.

test_burntime_2d_base_t6()
Tests burntime solution at 2D point in t_6 region.

Uses default parameter values.

test_burntime_3d_base_t1()
Tests burntime solution at 3D point in t_1 region (**geometry** $= 3$).

Uses default parameter values for inner region radius, HE detonation velocities, detonation times and detonator locations.

test_burntime_3d_base_t2()
Tests burntime solution at 3D point in t_2 region (**geometry** $= 3$).

Uses default parameter values for inner region radius, HE detonation velocities, detonation times and detonator locations.

test_burntime_3d_base_t3()
Tests burntime solution at 3D point in t_3 region (**geometry** $= 3$).

Uses default parameter values for inner region radius, HE detonation velocities, detonation times and detonator locations.

test_burntime_3d_base_t4()
Tests burntime solution at 3D point in t_4 region (**geometry** $= 3$).

Uses default parameter values for inner region radius, HE detonation velocities, detonation times and detonator locations.

test_burntime_3d_base_t5()
Tests burntime solution at 3D point in t_5 region (**geometry** $= 3$).

Uses default parameter values for inner region radius, HE detonation velocities, detonation times and detonator locations.

test_burntime_3d_base_t6()
Tests burntime solution at 3D point in t_6 region (**geometry** $= 3$).

Uses default parameter values for inner region radius, HE detonation velocities, detonation times and detonator locations.

test_burntime_2d_detvel1()
Tests burntime solution at 2D point in each region.

Uses default parameter values for inner region radius, outer region HE detonation velocity, detonator locations and detonation times. HE detonation velocity in the inner region is $D_1 = 1.5$.

test_burntime_3d_detvel1()
Tests burntime solution at 3D point in each region (**geometry** $= 3$).

Uses default parameter values for inner region radius, outer region HE detonation velocity, detonator locations and detonation times. HE detonation velocity in the inner region is $D_1 = 1.5$.

test_burntime_2d_detvel2()
Tests burntime solution at 2D point in each region.

Uses default parameter values for inner region radius, inner region HE detonation velocity, detonator locations and detonation times. HE detonation velocity in the outer region is $D_2 = 2.0$.

test_burntime_3d_detvel2()
Tests burntime solution at 3D point in each region (**geometry** $= 3$).

Uses default parameter values for inner region radius, inner region HE detonation velocity, detonator locations and detonation times. HE detonation velocity in the outer region is $D_2 = 2.0$.

test_burntime_2d_innerR()
Tests burntime solution at 2D point in each region.

Uses default parameter values for HE detonation velocities, detonator locations and detonation times. The radius of the inner region is $R = 2.0$.

test_burntime_3d_innerR()
Tests burntime solution at 3D point in each region (**geometry** $= 3$).

Uses default parameter values for HE detonation velocities, detonator locations and detonation times. The radius of the inner region is $R = 2.0$.

test_burntime_2d_dets1()
Tests burntime solution at 2D point in each region.

Uses default parameter values for inner region radius, HE detonation velocities and detonation times. Detonator locations have been reversed, thus specified at **dets** $= [-10.0, -5.0, 5.0, 10.0]$.

test_burntime_3d_dets1()
Tests burntime solution at 3D point in each region (**geometry** $= 3$).

Uses default parameter values for inner region radius, HE detonation velocities and detonation times. Detonator locations have been reversed, thus specified at **dets** $= [-10.0, -5.0, 5.0, 10.0]$.

test_burntime_2d_dets2()
Tests burntime solution at 2D point in each region.

Uses default parameter values for inner region radius, HE detonation velocities and detonation times. Detonator locations 1 and 2 have been moved to the same point. D2 ignites earlier than D1, thus the t1 solution region does not exist. Detonator locations are specified at **dets** $= [9.0, 9.0, -5.0, -10.0]$.

test_burntime_3d_dets2()
Tests burntime solution at 3D point in each region (**geometry** $= 3$).

Uses default parameter values for inner region radius, HE detonation velocities and detonation times. Detonator locations 1 and 2 have been moved to the same point. D2 ignites earlier than D1, thus the t1 solution region does not exist. Detonator locations are specified at **dets** $= [9.0, 9.0, -5.0, -10.0]$.

test_geometry_error()
Test for valid value of geometry.

test_R_neg_error()
Test for valid value of inner radius, R.

test_R_zero_error()
Test for valid value of inner radius, R.

test_D1_neg_error()
Test for valid value of inner region HE detonation velocity, D_1.

test_D1_zero_error()
Test for valid value of inner region HE detonation velocity, D_1.

test_D2_neg_error()
Tests for valid value of outer region HE detonation velocity, D_2.

test_D2_zero_error()
Tests for valid value of outer region HE detonation velocity, D_2.

test_D2_smaller_error()
Tests for valid value of outer region HE detonation velocity, D_2.

test_dets_notenough_error()
Tests for valid number of detonators, **dets**.

test_dets_toomany_error()
Tests for valid number of detonators, **dets**.

test_dets_2d_d1_error()
Tests for valid location of detonators, **dets**.

test_dets_2d_d2_error()
Tests for valid location of detonators, **dets**.

test_dets_2d_d4_error()
Tests for valid location of detonators, **dets**.

test_dets_2d_d5_error()
Tests for valid location of detonators, **dets**.

test_dets_3d_d1_error()
Tests for valid location of detonators, **dets**.

test_dets_3d_d2_error()
Tests for valid location of detonators, **dets**.

test_dets_3d_d4_error()
Tests for valid location of detonators, **dets**.

test_dets_3d_d5_error()
Tests for valid location of detonators, **dets**.

test_dettimes_notenough_error()
Tests for valid number of detonation times, t_{d_i}.

test_dettimes_toomany_error()
Tests for valid number of detonation times, t_{d_i}.

test_dettimes_2d_t1_error()
Tests for valid detonation times, t_{d_1}.

test_dettimes_2d_t2_error()
Tests for valid detonation times, t_{d_2}.

test_dettimes_2d_t4_error()
Tests for valid detonation times, t_{d_4}.

test_dettimes_2d_t5_error()
Tests for valid detonation times, t_{d_5}.

test_dettimes_3d_t1_error()
Tests for valid detonation times, t_{d_1}.

test_dettimes_3d_t2_error()
Tests for valid detonation times, t_{d_2}.

test_dettimes_3d_t4_error()
Tests for valid detonation times, t_{d_3}.

test_dettimes_3d_t5_error()
Tests for valid detonation times, t_{d_4}.

class exactpack.tests.test_kenamond.**TestKenamond3**(*methodName='runTest'*)
Tests for *exactpack.solvers.kenamond.kenamond3.Kenamond3*.

Solution tests consist of comparing the calculated burn time to the analytic solution at a fixed point. Input tests check that invalid input raises the appropriate error expression.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_burntime_2d_base_los()
Tests burntime solution at 2D point in LOS region.

Uses default parameter values.

test_burntime_2d_base_shadow()
Tests burntime solution at 2D point in shadow region.

Uses default parameter values.

test_burntime_3d_base_los()
Tests burntime solution at 3D point in LOS region.

Uses default parameter values for inert region radius, HE detonation velocity and detonation time. Detonator location is specified as $(0.0, 0.0, 5.0)$.

test_burntime_3d_base_shadow()
Tests burntime solution at 3D point in shadow region.

Uses default parameter values for inert region radius, HE detonation velocity and detonation time. Detonator location is specified as $(0.0, 0.0, 5.0)$.

test_burntime_2d_detvel_los()
Tests burntime solution at 2D point in LOS region.

Uses default parameter values for inert region radius, detonator location and detonation time. HE detonation velocity is $D = 1.0$.

test_burntime_2d_detvel_shadow()
Tests burntime solution at 2D point in shadow region.

Uses default parameter values for inert region radius, detonator location and detonation time. HE detonation velocity is $D = 1.0$.

test_burntime_3d_detvel_los()
Tests burntime solution at 3D point in LOS region.

Uses default parameter values for inert region radius and detonation time. Detonator location is specified as $(0.0, 0.0, 5.0)$. HE detonation velocity is $D = 1.0$.

test_burntime_3d_detvel_shadow()
Tests burntime solution at 3D point in shadow region.

Uses default parameter values for inert region radius and detonation time. Detonator location is specified as $(0.0, 0.0, 5.0)$. HE detonation velocity is $D = 1.0$.

test_burntime_2d_dettime_los()
Tests burntime solution at 2D point in LOS region.

Uses default parameter values for inert region radius, HE detonation velocity and detonator location. Detonation time is $t_d = -2.0$.

test_burntime_2d_dettime_shadow()
Tests burntime solution at 2D point in shadow region.

Uses default parameter values for inert region radius, HE detonation velocity and detonator location. Detonation time is $t_d = -2.0$.

test_burntime_3d_dettime_los()
Tests burntime solution at 3D point in LOS region.

Uses default parameter values for inert region radius and HE detonation velocity. Detonator location is specified as $(0.0, 0.0, 5.0)$. Detonation time is $t_d = -2.0$.

test_burntime_3d_dettime_shadow()
Tests burntime solution at 3D point in shadow region.

Uses default parameter values for inert region radius and HE detonation velocity. Detonator location is specified as $(0.0, 0.0, 5.0)$. Detonation time is $t_d = -2.0$.

test_burntime_2d_detloc_los()
Tests burntime solution at 2D point in LOS region.

Uses default parameter values for inert region radius, HE detonation velocity and detonation time. Detonator location is $x_d = (0.0, -5.0)$.

test_burntime_2d_detloc_shadow()
Tests burntime solution at 2D point in shadow region.

Uses default parameter values for inert region radius, HE detonation velocity and detonation time. Detonator location is $x_d = (0.0, -5.0)$.

test_burntime_3d_detloc_los()
Tests burntime solution at 3D point in LOS region.

Uses default parameter values for inert region radius, HE detonation velocity and detonation time. Detonator location is $x_d = (0.0, 0.0, -5.0)$.

test_burntime_3d_detloc_shadow()
Tests burntime solution at 3D point in shadow region.

Uses default parameter values for inert region radius, HE detonation velocity and detonation time. Detonator location is $x_d = (0.0, 0.0, -5.0)$.

test_burntime_2d_inertR_los()
Tests burntime solution at 2D point in LOS region.

Uses default parameter values for HE detonation velocity, detonation time and detonator location. Inert region radius is $R = 4.0$.

test_burntime_2d_inertR_shadow()
Tests burntime solution at 2D point in shadow region.

Uses default parameter values for HE detonation velocity, detonation time and detonator location. Inert region radius is $R = 4.0$.

test_burntime_3d_inertR_los()
Tests burntime solution at 3D point in LOS region.

Uses default parameter values for HE detonation velocity and detonation time. Detonator location is specified as $(0.0, 0.0, 5.0)$. Inert region radius is $R = 4.0$.

test_burntime_3d_inertR_shadow()
Tests burntime solution at 3D point in shadow region.

Uses default parameter values for HE detonation velocity and detonation time. Detonator location is specified as $(0.0, 0.0, 5.0)$. Inert region radius is $R = 4.0$.

test_geometry_error()
Test for valid value of geometry.

test_R_neg_error()
Test for valid value of inner radius, R.

test_R_zero_error()
Test for valid value of inner radius, R.

test_D_neg_error()
Test for valid value of HE detonation velocity, D.

test_D_zero_error()
Test for valid value of HE detonation velocity, D.

test_detgeom_2d_error()
Tests for valid geometry of detonator, x_d.

test_detgeom_3d_error()
Tests for valid geometry of detonator, x_d.

test_detloc_2d_error()
Tests for valid location of detonator, x_d.

test_detloc_3d_error()
Tests for valid location of detonator, x_d.

test_pts_in_inert(**args*, ***kwargs*)
Tests that solution points are outside the inert region.

7.7 The Mader Problem

Tests for the Mader rarefaction problem.

class exactpack.tests.test_mader.**TestMaderTimmes**(*methodName='runTest'*)
Test for exactpack.mader.timmes.Mader.

The tests consist of comparing the values against the exact solution at $r = 0.7$ and $t = 6.25 \times 10^{-6}$.

Note: Mader currently returns a NaN if the spatial interval is a single point; therefore, these tests really use $r = [0.7, 0.8]$ and the the first spatial point is selected. This is because Timmes' solution code returns cell averaged values.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_velocity()
Mader problem: velocity

test_pressure()
Mader problem: pressure

test_sound()
Mader problem: sound

test_density()
Mader problem: density

test_xdet()
Mader problem: xdet

7.8 The Noh Problem

Tests for the Noh problem are relatively rudimentary. Since the solution is an analytic expression, they essentially consist of checks for typographical errors.

class exactpack.tests.test_noh.**TestNoh1**(*methodName='runTest'*)
Tests for exactpack.noh.noh1.Noh.

The tests consist of comparing the values at two points, one in front of the shock ($r = 0.3$) and one behind the shock ($r = 0.1$), to the analytic solutions at a fixed time ($t = 0.6$) and $\gamma = 5/3$.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_velocity_error()
Noh Problem: Test for valid value of velocity

test_preshock_density()
Noh problem: Pre-shock density

test_preshock_energy()
Noh problem: Pre-shock internal energy

test_preshock_velocity()
Noh problem: Pre-shock velocity

test_preshock_pressure()
Noh problem: Pre-shock pressure

test_postshock_density()
Noh problem: Post-shock density

test_postshock_energy()
Noh problem: Post-shock internal energy

test_postshock_velocity()
Noh problem: Post-shock velocity

test_postshock_pressure()
Noh problem: Post-shock pressure

test_geometry_error()
Noh Problem: Test for valid value of geometry

class exactpack.tests.test_noh.**TestNohWrappers**(*methodName='runTest'*)
Test wrappers for Noh in specific geometries.

Test the wrapper functions for specific geometries from exactpack.noh.noh1, by comparing the results computed via the wrappers to those from the general solver.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_planar()
Planar Noh wrapper

test_cylindrical()
Cylindrical Noh wrapper

test_spherical()
Spherical Noh wrapper

class exactpack.tests.test_noh.**TestNohTimmes**(*methodName='runTest'*)
Tests for exactpack.noh.timmes.Noh.

The tests consist of comparing the values at two points, one in front of the shock ($r = 0.3$) and one behind the shock ($r = 0.1$), to the analytic solutions at a fixed time ($t = 0.6$) and $\gamma = 5/3$.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_preshock_density()
Noh problem: Pre-shock density

test_preshock_energy()
Noh problem: Pre-shock internal energy

test_preshock_velocity()
Noh problem: Pre-shock velocity

test_preshock_pressure()
Noh problem: Pre-shock pressure

test_postshock_density()
Noh problem: Post-shock density

test_postshock_energy()
Noh problem: Post-shock internal energy

test_postshock_velocity()
Noh problem: Post-shock velocity

test_postshock_pressure()
Noh problem: Post-shock pressure

test_geometry_error()
Noh Problem: Test for valid value of geometry

7.9 The Noh2 Problem

Tests for the Noh problem are relatively rudimentary. Since the solution is an analytic expression, they essentially consist of checks for typographical errors.

class exactpack.tests.test_noh2.**TestNoh2**(*methodName='runTest'*)
Tests for the Noh2 problem

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_noh2()
Regression test for Noh2

test_noh2_cog()
Noh2 is a speical case of Cog1, and this test compares them.

regression_test_planar_noh2()
Regression test for Planar Noh2

regression_test_cylindrical_noh2()
Regression test for Cylindrical Noh2

regression_test_spherical_noh2()
Regression test Spherical Noh2

7.10 The Riemann Problem

Tests for the Riemann problem.

class exactpack.tests.test_riemann.**TestRiemannKamm**(*methodName='runTest'*)
Test for exactpack.riemann.kamm.Riemann.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_Riemann_test1_Sod()
Riemann problem 1: Sod

test_Riemann_test2_Einfeldt()
Riemann problem 2: Einfeldt

test_Riemann_test3_StationaryContact()
Riemann problem 3: Stationary Contact

test_Riemann_test4_SlowShock()
Riemann problem 4: Slow Shock

test_Riemann_test5_ShockContactShock()
Riemann problem 5: Shock-Contact-Shock

test_Riemann_test6_LeBlanc()
Riemann problem 6: Le Blanc

7.11 The Reinicke Meyer-ter-Vehn Problem

Tests for the Reinicke Meyer-ter-Vehn (RMTV) problem.

class exactpack.tests.test_rmtv.**TestRmtvTimmes**(*methodName='runTest'*)
Test for exactpack.rmtv.timmes.Rmtv.

The comparisions are made at $r = 0.015$.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_density()
Rmtv problem: density

test_temperature()
Rmtv problem: temperature

test_energy(**args*, ***kwargs*)
Rmtv problem: energy

test_pressure(**args*, ***kwargs*)
Rmtv problem: pressure

test_velocity(**args*, ***kwargs*)
Rmtv problem: velocity

7.12 The Steady Detonation Reaction Zone Problem

tests the solver implementation for the Steady Detonation Reaction zone test problem (sdrz for short)

class exactpack.tests.test_sdrz.**TestSDRZAssignments**(*methodName='runTest'*)
Tests exactpack.solvers.sdrz.SteadyDetonationReactionZone.

These tests confirm proper assignment of variables, including default values

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

class exactpack.tests.test_sdrz.**TestSDRZSolution**(*methodName='runTest'*)
Tests exactpack.contrib.sdrz.SteadyDetonationReactionZone.

These tests confirm proper solution values for specific cases

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_fickett_table10_1()
Tests proper solution values by comparing default case to the Table 10.1 in Fickett & Rivard

test_sdrz_reaction_progress_limit()
Tests that burn rate is limited to be monotonic and cannot exceed 1.0

class exactpack.tests.test_sdrz.**TestSDRZNumericalIntegration**(*methodName='runTest'*)
Tests exactpack.contrib.sdrz.SteadyDetonationReactionZone.

These tests verify that the numerical integration scheme is working correctly.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

class exactpack.tests.test_sdrz.**TestSDRZFullSolution**(*methodName='runTest'*)
Tests exactpack.contrib.sdrz.SteadyDetonationReactionZone.

These tests verify that the full solution to SDRZ is behaving as expected.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

7.13 The Sedov Problem

The following test verifies the dimensionless self-similar variables of Kamm's Fortran code against the values published in Sedov's book and *[Kamm2000]*, (Tables I, II, and III, for the spherical, cylindrical, and planar cases respectively).

class exactpack.tests.test_sedov_kamm.**TestSedovKammSelfSim**(*methodName='runTest'*)
Tests for the Kamm's Sedov problem in self-similar variables.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_self_similar_sph()
Kamm's spherical Sedov: self-similar variables

test_self_similar_cyl()
Kamm's cylindrical Sedov: self-similar variables

test_self_similar_pla()
Kamm's planar Sedov: self-similar variables

test_shock_state_interpolated()
Sedov Kamm Problem: pre and post shock values, with interpolation to large number of points

test_geometry_error_sedov_kamm()
Sedov Kamm Problem: Test for valid value of geometry

class exactpack.tests.test_sedov_kamm.**TestSedovKammShock**(*methodName='runTest'*)
Tests Kamm Sedov for correct pre and post shock values.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_preshock_state()
Tests density, velocity, pressure, specific internal energy, and sound speed immediately before the shock.

test_postshock_state(**args*, ***kwargs*)
Tests density, velocity, pressure, specific internal energy, and sound speed immediately after the shock.

Currently, the Kamm solver does not return the correct value of the physical variables at the shock location, for at least some cases

The following is a test of Timmes' Sedov code.

class exactpack.tests.test_sedov_timmes.**TestSedovTimmesVsKamm**(*methodName='runTest'*)
This test compares Timmes vs Kamm, the latter of which has been verified against Sedov's book and LA-UR-00-6050.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_sedov_timmes_vs_kamm(**args*, ***kwargs*)
Checks Timmes' against Kamm's Sedov solver. Expected to fail because Kamm solver does not return correct values at shock for this choice of parameters.

class exactpack.tests.test_sedov_timmes.**TestSedovTimmesShock**(*methodName='runTest'*)
Tests Timmes Sedov for correct pre and post shock values.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_preshock_state()
Tests density, velocity, pressure, specific internal energy, and sound speed immediately before the shock.

test_postshock_state()
Tests density, velocity, pressure, specific internal energy, and sound speed immediately after the shock.

test_interpolate()
Sedov test: interpolation to large number of points.

test_geometry_assignment()
Sedov test: geometry assignment.

Tests the solver implementation of the Doebling Sedov code

exactpack.tests.test_sedov_doebling.**sedovFcnTable**(*solution*, *lamvec*)
helper function to find values of Sedov functions corresponding to a list of lambda values, lamvec

exactpack.tests.test_sedov_doebling.**sed_lam_min**(*v*, *solution*, *lam_want*)
helper function to find value of v corresponding to value of lambda

class exactpack.tests.test_sedov_doebling.**TestSedovDoeblingAssignments**(*methodName='runTest'*)
Tests *exactpack.solvers.sedov.doebling.Sedov*.

These tests confirm proper assignment of variables, including default values

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

class exactpack.tests.test_sedov_doebling.**TestSedovDoeblingSpecialSingularities**(*methodName='run*
Tests *exactpack.solvers.sedov.doebling.Sedov*.

These test the special sigularity cases of denom2=0 and denom3=0

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

class exactpack.tests.test_sedov_doebling.**TestSedovDoeblingFunctionsTable1**(*methodName='runTest'*)
Compare results to Kamm & Timmes, Table 1. Sedov Functions for gamma=1.4, planar geometry case

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

class exactpack.tests.test_sedov_doebling.**TestSedovDoeblingFunctionsTable2**(*methodName='runTest'*)
Compare results to Kamm & Timmes, Table 2. Sedov Functions for gamma=1.4, cylindrical geometry case

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

class exactpack.tests.test_sedov_doebling.**TestSedovDoeblingFunctionsTable3**(*methodName='runTest'*)
Compare results to Kamm & Timmes, Table 3. Sedov Functions for gamma=1.4, spherical geometry case

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

class exactpack.tests.test_sedov_doebling.**TestSedovDoeblingFunctionsTables45**(*methodName='runTest*
Compare results to Kamm & Timmes, Tables 4 & 5 Values of key variables for the gamma = 1.4 uniform density test cases at t=1s

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

class exactpack.tests.test_sedov_doebling.**TestSedovDoeblingFunctionsTable67**(*methodName='runTest'*)
Compare results to Kamm & Timmes, Tables 6 & 7. Values of key variables for the gamma = 1.4 singular test cases at t=1s

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

class exactpack.tests.test_sedov_doebling.**TestSedovDoeblingFunctionsTable89**(*methodName='runTest'*)
Compare results to Kamm & Timmes, Tables 8 & 9. Values of key variables for the gamma = 1.4 vacuum test cases at t=1s

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

class exactpack.tests.test_sedov_doebling.**TestSedovDoeblingShock**(*methodName='runTest'*)
Tests Doebling Sedov for correct pre and post shock values.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_preshock_state()
Tests density, velocity, pressure, specific internal energy, and sound speed immediately before the shock.

test_postshock_state()
Tests density, velocity, pressure, specific internal energy, and sound speed immediately after the shock.

7.14 The Su-Olson Problem

Tests for the Su-Olson problem.

class exactpack.tests.test_suolson.**TestSuOlsonTimmes**(*methodName='runTest'*)
Test for exactpack.suolson.timmes.SuOlson.

Comparisions are made at $r = 0.1$ and $t = 10^{-9}$.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_radiation_temperature()
SuOlson problem: radiation temperature

test_matter_temperature()
SuOlson problem: matter temperature

Unit testing for ExactPack.

Although (or, perhaps, because) one important use case for ExactPack is for generating solutions to use in verification of other codes, ExactPack itself needs to be tested. The modules in exactpack.test contain python unittest test cases designed to self-test the library.

CHAPTER

EIGHT

INDICES AND TABLES

8.1 Credits

ExactPack was developed by the Physics Verification project team within the Computational Physics Division (XCP) at Los Alamos National Laboratory. Scott Doebling had the original idea to collect exact solution codes under a uniform Python API. The initial implementation was done by Daniel Israel and Robert Singleton.

The current development team at Los Alamos National Laboratory includes:

- Scott Doebling
- Daniel Israel
- Robert Singleton
- C Nathan Woods
- John Walter
- Kyle Hickmann
- Gowri Srinivasan

Past contributors at Los Alamos National Laboratory include:

- Ann Kaul
- James Kamm
- Frank Timmes
- Scott Ramsey
- Jerry Brock

8.2 To Do List

Todo

Add a search directory list to *exactpack.utils.discover_solvers()*, so that users can develop other solvers outside the library.

(The original entry is located in /home/doebling/Documents/2017/exactpack/exactpack/utils.py:docstring of exactpack.utils.discover_solvers, line 12.)

8.3 Glossary

API Application program interface.

solution An object of type `exactpack.base.ExactSolution`, which represents the solution produced by a particular *solver*.

solver An object of a type derived from `exactpack.base.ExactSolver`. This is a function-like object which, when called, will return a *solution* object for the specified problem.

8.4 References

- genindex
- modindex
- search

BIBLIOGRAPHY

[Bdzil] Bdzil, J. B., R. J. Henninger, and J. W. Walter, Test Problems for DSD2D, LA-14277, 2006.

[Coggeshall1991] S. V. Coggeshall, *Analytic solutions of hydrodynamics equations*, Phys. Fluids A **3** (1991) 757.

[Dawes2015] A Dawes, *3D Multi-Material Polyhedral Methods for Diffusion*, MultiMat Conference, Warzberg, Germany (2015).

[Dawes2016] A. Dawes, C. Malone, M. Shashkov, *Some New Verification Test Problems for Multimaterial Diffusion on Meshes that are Non-Aligned with Material Boundaries.*, LA-UR-16-24696, Los Alamos National Laboratory (2016).

[Doebling2015] S. Doebling, *The Escape of High Explosive Products: An Exact-Solution Problem for Verification of Hydrodynamics Codes*, LA-UR-15-22547, Los Alamos National Laboratory (2015).

[Dykema2002] P. Dykema, S. Brandon, J. Bolstad, T. Woods, and R. Klein, *Level 1 V. & V. Test Problem 10: Escape of High Explosive Products*, UCRL-ID-150418, Lawrence Livermore National Laboratory (2002).

[Fickett1974] W. Fickett and C. Rivard, *Test Problems for Hydrocodes*, LA-5479, Los Alamos Scientific Laboratory (1974, Rev 1981).

[Fickett1979] W. Fickett and W. C. Davis, *Detonation: Theory and Experiment*, University of California Press, Berkeley, 1979.

[Gehmeyr1997] M. Gehmeyr, B. Cheng, and D. Mihalas, *Noh's constant-velocity shock problem revisited*, Shock Waves **7** (1997) 255.

[Guderley1942] G. Guderley, Luftahrtforschung **19** (1942) 302.

[Guderley2012] Scott D. Ramsey, James R. Kamm, and John H. Bolstad, *The Guderley problem revisited*, International Journal of Computational Fluid Dynamics **26** (2012) 79.

[Kamm2000] James R. Kamm, *Evaluation of the Sedov-von Neumann-Taylor Blast Wave Solution*, LA-UR-00-6055, Los Alamos National Laboratory (2000).

[Kamm2000a] James R. Kamm, *Investigation of the Reinicke & Meyer-ter-Vehn Equations: I. The Strong Conduction Case*, LA-UR-00-4304, Los Alamos National Laboratory (2000).

[Kamm2008] James R. Kamm, Jerry S. Brock, Scott T. Brandon, David L. Cotrell, Bryan Johnson, Patrick Knupp, William J. Rider, Timothy G. Trucano, and V. Gregory Weirs, *Enhanced Verification Test Suite for Physics Simulation Codes*, LA-14379, Los Alamos National Laboratory (2008).

[KaulK22016] Ann Kaul, *A More General Solution of the Kenamond HE Problem 2*, LA-UR-15-29547, Los Alamos National Laboratory (2015).

[KaulK32016] Ann Kaul, *General Solution of the Kenamond HE Problem 3*, LA-UR-15-29553, Los Alamos National Laboratory (2015).

[Kenamond2011] Kenamond, M. A., *HE Burn Table Verification Problems*, LA-UR-11-03096, Los Alamos National Laboratory (2011).

[Kirkpatrick2004] R. Kirkpatrick, C. Wingate, and J.R. Kamm, *HE Burn Test Problem*, X-3-19U (2004).

[Noh1987] W. F. Noh, *Errors for Calculations of Strong Shocks Using an Artificial Viscosity and an Artificial Heat Flux*, JCP **72** (1987) 78-120.

[Hutchens2009] G J Hutchens, *A Generalized Set of Heat Conduction Test Problems*, LA-UR-09-01692, Los Alamos National Laboratory (2009).

[Reinicke1991] P. Reinicke and J. Meyer-ter-Vehn, *The point explosion with heat conduction*, Phys. Fluids A **3** (1991) 1807.

[Sedov1959] L.I. Sedov, *Similarity and Dimensional Methods in Mechanics*, Academic Press, New York, NY, p. 147 ff. (1959).

[Su1996] Bingjing Su and Gordon L. Olson, Benchmark Results for the Non-equilibrium Marshak Diffusion Problem, J. Quant. Spectrosc. Radiat. Transfer **56** 337 (1996).

[Timmes2005] Francis X. Timmes, Galen Gisler, and George M. Hrbek, *Automated Analyses of the Tri-Lab Verification Test Suite on Uniform and Adaptive Grids for Code Project A*, LA-UR-05-6865, Los Alamos National Laboratory (2005).

PYTHON MODULE INDEX

e

INDEX

D

E

F

G

H

J

K

L

M

N

P

R

S

T

U

W