TTiP Documentation

Release 0.0.0

Andrew Lister

May 28, 2020

Contents:

TTiP - Thermal Transfer in Plasma is a tool to solve heat transfer problems specifically aimed at the plasma physics domain.

This tool uses Firedrake as the backend solver. The install for this can take sme time, alternatively a docker image can be used as described on the Installing page.

CHAPTER 1

Installing TTiP

1.1 Firedrake

TTiP uses Firedrake to solve the underlying equations. To install this, please see the install page in their documentation here. Alternatively, firedrake has a docker image available here, this image contains a build of Ubuntu with firedrakes dependencies pre-installed, and will be much faster than building from scratch.

Firedrake in turn uses the widely used PETSc which will be installed with Firedrake but can take some time.

It is recommended to use the default install for this, which will create a directory called "firedrake" where all the necessary packages can be found. In particular this contains a virtualenv which will be required to run TTiP.

To activate the virtualenv run:

```
. firedrake/bin/activate
```

This will mean python can access the firedrake modules.

1.2 TTiP

Once inside the firedrake virtualenv, install any additional packages that are required with (in the root of this repository):

```
pip install .
```

To check it works try running:

```
ttip --help
```

CHAPTER 2

Using TTiP

TTiP is controlled entirely by the problem definition config file and so running the software is as easy as:

```
ttip my_problem.ini
```

provided "my_problem.ini" is a correctly defined problem as per *Configuration File*.

2.1 Troubleshooting

If the above does not work there are several things to check:

- The firedrake environment is active (usually this is shown in the terminal). If it is not, run:

  ```
  . <firedrake_dir>/bin/activate
  ```

  where <*firedrake_dir*> is the location of the firedrake installation.

- The "ttip" command has been installed correctly:

  ```
  ls <firedrake_dir>/bin/ttip
  ```

  If not try reinstalling as per instructions in *Installing TTiP*.

CHAPTER 3

Configuration File

3.1 Functions

TTiP allows the creation of functions using either one of the built in function builders, or by writing expressions.

The simplest functions can be defined as:

```
<name>: <expression>
```

e.g.:

```
radial: 1/sqrt(x^2 + y^2)
```

Functions can also be interim functions. An interim function is one that is used as a component in other functions but is not used directly by the section. To define an interim function, start the function name with an '_':

```
_<name>: <expression>
```

Interim functions can then be used by name:

```
_foo: 10 + x*y
bar: foo^2
```

TTiP also offers predefined function builders which offer some useful functionality. Function builders are used to create specific functions in the TTiP configuration file and are defined by setting the various properties with respect to a name, using the form:

```
<name>.<property>: <value>
```

Each function builder must define the *type* property, along with the relevent properties for the selected type. e.g.:

```
foo.type: gaussian
foo.mean: 0.5
foo.sd: 0.1
foo.scale: 10
```

Available function builders are:

- *Condition*
- *Constant*
- *File*
- *Gaussian*

3.1.1 Condition

The condition function allows the user to create a stepped function using an operator alongside a left and right hand side (lhs and rhs respectively).

Args:

- operator (str): The operator to use.
- lhs (str): An expression for the left hand side.
- rhs (str): An expression for the right hand side.

The exact formula is:

$$\text{Condition}(x, \text{operator}, \text{lhs}, \text{rhs}) = \begin{cases} 1, & \text{lhs operator rhs} \\ 0, & \text{otherwise} \end{cases}$$

Where:

- operator is an operator ('<', '>', '<=', '==', ...)
- lhs is an expression for the left hand side
- rhs is an expression for the right hand side

e.g.

$$\text{Condition}(\vec{x}, <, x^2 + y^2, 10) = \begin{cases} 1, & x_1^2 + x_2^2 < 10 \\ 0, & \text{otherwise} \end{cases}$$

3.1.2 Constant

The Constant function creates a uniform function of *value* across the whole domain.

Args:

- value (numerical): The constant value.

The exact formula is:

$$\text{constant}(x, \text{value}) = \text{value}$$

Where:

- value is a scalar

3.1.3 File

The File function creates an function interpolated from data in a file.

Args:

- path (str): The path to the file to interpolate.

The file must be a csv (optionally with a commented header row), where the first n columns represent the coords and the final column is the value at the coordinate.

e.g. For a 2D problem:

```
# x, y, val
0.0, 0.0, 10.0
0.1, 0.0, 9.0
0.2, 0.0, 8.0
0.3, 0.0, 7.0
...
```

Cubic interpolation is used for 1-D or 2-D problems, while linear interpolation is used for higher dimensional problems.

3.1.4 Gaussian

The Gaussian function creates a symmetric function with a peak equal to the *scale* at the location defined by *mean*. As the distance from this peak increases, the function decays exponentially. This decay is controlled by *sd*.

Args:

- mean (numerical): The mean for the function.
- sd (numerical): The standard deviation of the function.
- scale (numerical): The amount to scale the function by.

The exact formula is:

$$\text{gaussian}(x, \text{mean}, \text{sd}, \text{scale}) = \text{scale} \times e^{-\frac{1}{2}\left(\frac{x-\text{mean}}{\text{sd}}\right)\cdot\left(\frac{x-\text{mean}}{\text{sd}}\right)}$$

Where:

- mean is a vector (comma seperated list) of length dim(x), or a scalar value that will be broadcast to a vector.
- sd is a vector (comma seperated list) of length dim(x), or a scalar value that will be broadcast to a vector.
- scale is a scalar value

3.2 File Sections

Each section is defined properly in the example config:

```
[PHYSICS]
# The physics section is used to enable/disable physical limits.

# limit_conductivity (bool): Enable the lower limit on conductivity?
# limit_flux (bool): Enable flux limiting?
limit_conductivity: on
```

(continues on next page)

(continued from previous page)

```
limit_flux: on

[SOLVER]
# The solver section defines which solver to use and where to store the
# results.

# file_path (string): The path to store the result in.
# method (string): The method to use for time dependant problems.
#     Avalilable options are: ForwardEuler, BackwardEuler, CrankNicolson, and Theta.
# theta (float): **Theta only** The theta parameter for a theta model solve.
file_path: ttip_results/result.pvd
method: CrankNicolson

[MESH]
# The mesh is defined by a type and parameters.
# Mesh types include all UtilityMeshes in firedrake as well as the option to
# read from a file.
#
# To create a mesh from file, stored at <filename>, use:
# type: Mesh
# params: <filename>
#
# For information on Utility meshes, see the firedrake docs:
# https://firedrakeproject.org/firedrake.html#module-firedrake.utility_meshes

# type (string): Which Mesh builder to use.
# params (comma seperated list): The params for the given mesh builder.
# element (string): The type of element to use in the mesh.
# order (int): The order of the element to use in the mesh.
type: Box
params: 20, 20, 20, 4e-5, 4e-5, 4e-5

element: CG
order: 1


[PARAMETERS]
# Any parameters are defined by name.
# For example the below defines the density parameter.
# A parameter will only be used if one of the mixins requests it.
# Parameters follow the same pattern as other functions which can be read about
# below.
#
# No defaults are provided. Options are given below:
# atomic_number
# coulomb_ln
# electron_density
# ion_density
# ionisation

[SOURCES]
# Any sources that are defined here will be summed to produce a final source
# term.
# Sources follow the same pattern as other functions which can be read about
# below.
#
# The default is no source term.
```

(continues on next page)

(continued from previous page)

```
# default_source: 0

[BOUNDARIES]
# Boundary conditions can be defined for each boundary surface in the mesh.
# Problems can have no boundary conditions by leaving this section empty,
# otherwise each surface must have 1 boundary condition, to define a condition
# on all surfaces use 'all'.

# <name>.boundary_type (string): The type of boundary condition
#                                (dirichlet, neuman, robin)
# <name>.surface (comma separated list or 'all'): The surface the condition
#                                                 applies for.
# If using dirichlet, you should also define:
#     - g (the value on the boundary). This should be either a scalar value,
#       or a function defined as described at the bottom of this document.
#       e.g. bound.g.type: Constant ...
# If using neuman (not available yet!), you should also define:
#     - g (the derivative on the boundary). This should be a scalar or
#       function.
# If using robin (not available yet!), you should also define:
#     - alpha
#     - g
#fixed.boundary_type: dirichlet
#fixed.g: 100
#fixed.surface: 1, 2, 3, 4, 5, 6

[TIME]
# Time is used to define any time dependency in the problem.
# If this section is left blank, the problem is assumed to be steady state.
# Only 2 (any 2) of the settings in this section are needed, and the third can
# be calculated.

# steps (int): The number of steps to take.
# dt (float): The length of each time step.
# max_t (float): The maximum time to step until.
#steps: 10
#dt: 1e-12
#max_t: 1e-13

[INITIALVALUE]
# Initial values follow the same pattern as sources.
#
# Defaults to 0 across the mesh.
# default_initial_value: 0


# Functions:
# To define a function in TTiP, we use the following format
# <name>.type: <type>
# <name>.<param1>: <value1>
# ...
#
# For details on available functions and their parameters, please see the main
# documentation for TTiP.
```

CHAPTER 4

TTiP - Contributor Documentation

Welcome to the contributor documentation for TTiP.

TTiP is built to be extendable in 2 main ways:

- *Problem definition*: The formula to be solved by the finite element code is highly extendable through the use of Mixin classes.
- *Functions*: The available functions for defining parameters can be extended through the use of function_builder classes.

These are both described more thoroughly in the linked pages above, as well as throughout the source code.

4.1 Code Structure

The source code for TTiP is structured into the following sections:

- *cli*: Entry points to the software.
- *core*: Main functionality.
- *function_builders*: Classes to create functions from config inputs.
- *parsers*: Code for parsing the config file.
- *problem_mixins*: Extensions for the problem class.
- *resources*: Any non code assets.
- *util*: Generic code that is used in various other places.

More detail on these can be found below.

4.1.1 CLI

The cli is designed be the main access point for TTiP from the terminal. Currently this only includes a single file which serves as the main entry point for the software.

4.1.2 Core

The core folder contains code that is central to the logic of TTiP and ties the other section together. This includes the solver, base problem, and the interface to the parsers.

4.1.3 Function Builders

The function builders folder follows a factory design pattern. This folder holds the factory and all of the function builders. More information on this can be found in *Function Builders*.

4.1.4 Parsers

The parsers folder holds all code related to parsing the config file. This includes a file for each section of the config file, as well as a base class and argument parser. More information can be found in *Config Parsing* and *Expression Parsing*.

4.1.5 Problem Mixins

The problem mixins directory holds all optional extensions for the problem class. Information on these mixins is in *Problem Class*.

4.1.6 Resources

The resources directory is a place to store assets that are not source code. Currently the only file in this directory is a default config file which also serves to document the config options for a user.

4.1.7 Util

Finally, the util directory is a place for anything that doesn't obviously fit in the other directories. This is likely to be any utility methods or functionality. Currently this only holds some code to setup a logger.

4.2 Problem Class

The problem definitions in TTiP are designed to be extensible, this is done through multiple inheritance and mixin classes.

The two main parts to a problem class in TTiP are the base class and the mixins.

4.2.1 Base Class

The base class (found in `core/problem.py`) defines a set of methods that create, set (replace), and bound firedrake functions. It also defines a set of functions such as the flux and ionisation terms which are used in multiple mixins.

The formula that the problem defines is given by:

$$a - L = 0$$

a is the sum of terms containing both T and v, whereas L does not contain T explicitly (some parameters can be dependant on temperature).

4.2.2 Mixins

The mixins are fundamental to creating a sensible problem.

A variety of mixins can be found in the `problem_mixins` directory.

Mixins are used to add functionality such as time dependancies, flux limits, specific conductivity formulations, and many others.

The key concept of a mixin for the problem class is that it should edit the existing variables rather than replacing them, either by adding terms or by using the `set_function` method from the parent class (which will be available at runtime as the mixin will only ever be used with the base problem class). Some examples of this are the time dependancy mixin which adds a term to the `a` attribute, and the conductivity limiter which imposes bounds on the existing conductivity.

4.2.3 Usage

Some examples of how to define new problem classes can be seen at the bottom of the `core/problem.py` file as well as a utility function to dynamically create new problem classes.

4.2.4 Extending

To add new functionality to the problem class, define a new mixin (using the template in `problem_mixins`) this can then be used in any custom scripts. To make the new mixin more accessible, it should be added to the `create_problem_class` method in the problem file.

4.3 Config Parsing

Parsing the config file is done using a set of parsers, one for each config section in the `parsers` directory. These are called from `core/read_config.py`.

To extend the config options to add more options to a config section, add the parsing to `parse` function in the associated parser, and fetch it in the `read_config.py` section.

To add a new section, an additional parser should be created.

4.4 Function Parsing

Functions in TTiP can be divided into 2 categories, those built using expressions, and those built using predefined constructs or function builders.

Both of these have been created in such a way that they can be extended, although Function Builders are considerably easier to understand.

4.4.1 Expression Parsing

The expression parser can be found in `parsers/parse_args.py`.

This works by reading the expression from left to right and building a tree which is evaluated once complete. For example:

```
x + 2*y :=     +
              / \
             x   *
                / \
               2   y
```

The expression parser is made up of Node subclasses: Expression, List, and Terminal.

Base Node

The base class contains code to return the parent node when created.

Due to the way this works, the tree is built up in both directions. As such, when creating a node from a string, the root node should be returned. This is done by overriding `__new__` and using a custom initialiser `_init` rather than the usual `__init__` (using the standard initialiser causes objects to be reinitialised after `__new__`).

The base node also controls named terminals for the expression parser. This allows custom terminals, where a name is given to the parser from the calling code alongside a value, and reserved terminals, where the name is reserved for use by one of the Node classes (e.g. 'false').

Finally, the base node also has the functionality to return a ready state describing if it can be evaluated yet. A node is considered to be ready if all the required terminals in custom_terminals have been assigned a value.

Expression

The Expression class handles the bulk of the tree building, and within, this is done on initialisation (in the `_init` method).

In brief, this class takes a string and determines the next operator, it then creates a new tree for each of the left and right operands. If the operator has a lower or equal priority than the parents operator, a pivot is applied.

The pivot follows the following pattern:

```
  op1                 op2
 /   \               /   \
x    op2    ->     op1    z
    /   \          /   \
   y     z        x     y
```

Where x, y, and z can also be a tree.

The operands are defined by a dictionary at the class level which maps the string format to a tuple of priority, callable, and string format.

Expressions can also contain unary functions defined in the functions dictionary. These are called by a name followed by an open bracket (except in the case of negation).

New binary functions should be added by extending the operators dictionary and new unary functions should be added by extendin the functions dictionary.

Evaluation of an expression is done by first evaluating a left and right hand side, then applying an operator to them.

List

The list node is used for any string with a comma in. It assumes that the string is a comma separated list and creates an expression for each of these.

Terminal

Terminal nodes evaluate the singleton values after operators have been removed. This includes boolean, integer, float, string, and coordinate values.

Coordinate values are reserved keyword mappings that allow for expressions such as:

```
x^2 + y^2
```

Where x and y are the mesh coordinates.

4.4.2 Function Builders

The function builders can be found in the *function_builders* directory alongside a base class and a factory implementation.

Base Class (`function_builder.py`)

The base class defines some shared interfaces for all of the function builders, and should be inheritted from for a new function builder to work.

In general, the only requirement for the sub classes of this are that they override the abstract `build` method so that it returns a firedrake Function, or a numerical value. In practice, since this is designed to only be accessed by the factory (explained below), there are additional constraints:

- Function builders should not modify the signature of `__init__`. E.g. they are initialised with `(mesh, V)` and any other arguments must be defined using the assign method.
- The signature of build should not be edited. To access arguments for constructing a function, the object dictionary `_props` should be used.

Since the build method signature cannot be changed, values must be passed through a different mechanism. This is done using the `_props` dictionary and `assign` method.

Assign takes a *name* and a *value* as inputs, it then checks this pair is valid for the class by checking that the *name* is in the `properties` dictionary and that the type of *value* matches the entry in `properties`. If it is valid, `_props` is updated with the *name-value* pair.

Factory (`function_builder_factory.py`)

The factory is used to dynamically import and use only the specified function builder at runtime. This means that extending the set of function builders is done by creating a new file. The factory will then be able to find the new builder by name and load the code.

In order for the factory to work, there can only be 1 function builder per file.

Function Builders (`<type>_builder.py`)

The rest of the files in this directory are function builders and should serve as good examples for adding new ones.

To create a new function builder, you should make sure to define the following on the new subclass:

- `properties` dict: This dictionary should have the names of all properties that are required to build a function, alongside the type of each property. Types can be specified as a tuple e.g. if a value can be *int* or *float*.
- `build` method: This is the main logic in the builder.

It is also important to name the file sensibly as this is how the function builder will be selected. For a file called `foo_builder.py`, the user will be required to set the type in the config file to `foo`. If this is not appropriate in the future, the factory will have to be changed.

4.5 Logging

TTiP uses pythons default logger via a wrapper stored in `utils` this sets up the logger and ensures the correctly set up logger is used.

4.6 Testing

All tests for TTiP are done using pytest and are automated through github actions. Coverage is monitored using codecov.

4.6.1 Tests

Currently, the only testing in TTiP is unit testing. Most functions in the code are supported by multiple tests to cover every input permutation. Some functions are declaritive and are not tested as this would likely lead to code being duplicated in the tests.

4.6.2 Linting

All source code is ran through both pylint and flake8 to enfore coding standards. This is done as a github action before pull requests are merged.

4.6.3 Coverage

The coverage is calculated once the tests suite completes. When new code is added, coverage should be considered and tests added.

4.7 Documentation

The documentation for TTiP is all stored in the `docs` directory and is written to be used with sphinx.

A local copy of the documentation can be generated using:

```
cd docs
make html
```

An online copy of the documentation can be found on readthedocs, and is automatically updated when commits are added to github.

4.8 Some key links for contributors:

- Source
- Tests
- Coverage
- Docs

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

T