firefish Documentation

Release 0.0.1dev

Cambridge University Spaceflight

June 21, 2016

Contents

1 Lid-driven cavity flow **3**

2 Supersonic flow over wedge **9**

3 Snappy Hex Mesh Example **13**

4 Join STL Example **15**

5 Kinematics Example **17**

6 Reference **19**

6.1 OpenFOAM case directory manipulation . . . 19
6.2 IO . . . 22
6.3 Geometry manipulation . . . 22
6.4 Mesh generation . . . 24
6.5 Fin-flutter . . . 25
6.6 Kinematics . . . 27

7 CU Spaceflight Simulation Software **29**

Python Module Index **31**

This section documents the example scripts shipped with the source code.

CHAPTER 1

Lid-driven cavity flow

The `openfoam_cavity_tutorial.py` script provides an example of low-level manipulation of OpenFOAM cases. In this example we shall re-create the initial example from the OpenFOAM users' guide. It's worth reading over that section first before trying to follow the transliteration below.

Firstly, we need to import some things from the *firefish.case* module:

```
from firefish.case import Case, FileName, FileClass
```

The *Case* class encapsulates an OpenFOAM case directory. We don't want to overwrite an existing case and so we write a little convenience wrapper function:

```
def create_new_case(case_dir):
    # Check that the specified case directory does not already exist
    if os.path.exists(case_dir):
        raise RuntimeError(
            'Refusing to write to existing path: {}'.format(case_dir)
        )

    # Create the case
    return Case(case_dir)
```

The *mutable_data_file()* method will return a *context manager* which can be used to manipulate an OpenFOAM data file. The data file is created if it does not yet exist, its contents are parsed into a dictionary and the dictionary is returned from the context manager. One the context is left the dictionary is re-written to disk.

The upshot of this is that the programmer is insulated from manipulating OpenFOAM data files directly. Let's write the *controlDict* file from the tutorial:

```
def write_initial_control_dict(case):
    # Control dict from tutorial
    control_dict = {
        'application': 'icoFoam',
        'startFrom': 'startTime',
        'startTime': 0,
        'stopAt': 'endTime',
        'endTime': 0.5,
        'deltaT': 0.005,
        'writeControl': 'timeStep',
        'writeInterval': 20,
        'purgeWrite': 0,
        'writeFormat': 'ascii',
        'writePrecision': 6,
        'writeCompression': 'off',
        'timeFormat': 'general',
```

```
        'timePrecision': 6,
        'runTimeModifiable': True,
    }

    with case.mutable_data_file(FileName.CONTROL) as d:
        d.update(control_dict)
```

Well-known file names are available through the `FileName` class.

The *blockMeshDict* file is the next one to be created. This is an example of a relatively complex file. The complexity is somewhat hidden by the mapping to and from the Python domain but there is still some subtlety. Notice particularly the rather odd way in which the *boundary* dictionary is specified:

```
def write_block_mesh_dict(case):
    block_mesh_dict = {
        'convertToMeters': 0.1,

        'vertices': [
            [0, 0, 0], [1, 0, 0], [1, 1, 0], [0, 1, 0],
            [0, 0, 0.1], [1, 0, 0.1], [1, 1, 0.1], [0, 1, 0.1],
        ],

        'blocks': [
            (
                'hex', [0, 1, 2, 3, 4, 5, 6, 7], [20, 20, 1],
                'simpleGrading', [1, 1, 1],
            )
        ],

        'edges': [],

        # Note the odd way in which boundary is defined here as a
        # list of tuples.
        'boundary': [
            ('movingWall', {
                'type': 'wall',
                'faces': [ [3, 7, 6, 2] ],
            }),
            ('fixedWalls', {
                'type': 'wall',
                'faces': [
                    [0, 4, 7, 3],
                    [2, 6, 5, 1],
                    [1, 5, 4, 0],
                ],
            }),
            ('frontAndBack', {
                'type': 'empty',
                'faces': [
                    [0, 3, 2, 1],
                    [4, 5, 6, 7],
                ],
            }),
        ],

        'mergePatchPairs': [],
    }

    with case.mutable_data_file(FileName.BLOCK_MESH) as d:
```

```
        d.update(block_mesh_dict)
```

At this point in the tutorial we're ready to run the *blockMesh* command which is one function call away:

```
case.run_tool('blockMesh')
```

We're close to being able to run the *icoFoam* utility. The transport properties need to be defined:

```
from firefish.case import Dimension

with case.mutable_data_file(FileName.TRANSPORT_PROPERTIES) as tp:
   tp['nu'] = (Dimension(0, 2, -1, 0, 0, 0, 0), 0.01)
```

We also need to create the initial conditions. Notice that we have to specify a different class when creating them:

```
def write_initial_conditions(case):
    # Create the p initial conditions
    p_file = case.mutable_data_file(
        '0/p', create_class=FileClass.SCALAR_FIELD_3D
    )
    with p_file as p:
        p.update({
            'dimensions': Dimension(0, 2, -2, 0, 0, 0, 0),
            'internalField': ('uniform', 0),
            'boundaryField': {
                'movingWall': { 'type': 'zeroGradient' },
                'fixedWalls': { 'type': 'zeroGradient' },
                'frontAndBack': { 'type': 'empty' },
            },
        })

    # Create the U initial conditions
    U_file = case.mutable_data_file(
        '0/U', create_class=FileClass.VECTOR_FIELD_3D
    )
    with U_file as U:
        U.update({
            'dimensions': Dimension(0, 1, -1, 0, 0, 0, 0),
            'internalField': ('uniform', [0, 0, 0]),
            'boundaryField': {
                'movingWall': {
                    'type': 'fixedValue', 'value': ('uniform', [1, 0, 0])
                },
                'fixedWalls': {
                    'type': 'fixedValue', 'value': ('uniform', [0, 0, 0])
                },
                'frontAndBack': { 'type': 'empty' },
            },
        })
```

Before we can run *icoFoam*, we must create the mysterious *fvSolution* file:

```
def write_fv_solution(case):
    fv_solution = {
        'solvers': {
            'p': {
                'solver': 'PCG',
                'preconditioner': 'DIC',
                'tolerance': 1e-6,
```

```
                'relTol': 0,
            },
            'U': {
                'solver': 'smoothSolver',
                'smoother': 'symGaussSeidel',
                'tolerance': 1e-5,
                'relTol': 0,
            },
        },
        'PISO': {
            'nCorrectors': 2,
            'nNonOrthogonalCorrectors': 0,
            'pRefCell': 0,
            'pRefValue': 0,
        }
    }

    with case.mutable_data_file(FileName.FV_SOLUTION) as d:
        d.update(fv_solution)
```

And also the equally mysterious *fvSchemes* file:

```
def write_fv_schemes(case):
    fv_schemes = {
        'ddtSchemes': { 'default': 'Euler' },
        'gradSchemes': { 'default': 'Gauss linear', 'grad(p)': 'Gauss linear' },
        'divSchemes': { 'div(phi,U)': 'Gauss linear', 'default': 'none' },
        'laplacianSchemes': { 'default': 'Gauss linear orthogonal' },
        'interpolationSchemes': { 'default': 'linear' },
        'snGradSchemes': { 'default': 'orthogonal' },
    }

    with case.mutable_data_file(FileName.FV_SCHEMES) as d:
        d.update(fv_schemes)
```

The example script includes a `main()` function which performs all of these steps:

```
def main(case_dir='cavity'):
    # Create a new case file, raising RuntimeError if the directory already
    # exists.
    case = create_new_case(case_dir)

    # Add the information needed by blockMesh.
    write_initial_control_dict(case)
    write_block_mesh_dict(case)

    # At this point there is enough to run blockMesh.
    case.run_tool('blockMesh')

    # Update the physical properties.
    with case.mutable_data_file(FileName.TRANSPORT_PROPERTIES) as tp:
        tp['nu'] = (Dimension(0, 2, -1, 0, 0, 0, 0), 0.01)

    # Write the fvSolution and fvSchemes files.
    write_fv_solution(case)
    write_fv_schemes(case)

    # Write the initial conditions for the p and U fields.
    write_initial_conditions(case)
```

```
    # Run the icoFoam application.
    case.run_tool('icoFoam')
```

After the example script is run, *paraFoam* may be run to inspect the solution.

Supersonic flow over wedge

The `openfoam_supersonic_wedge.py` script provides an example of setting up a compressible flow solver in OpenFoam.

As in `openfoam_cavity_tutorial.py` we set up the OpenFOAM case directory using the *firefish.case* framework.

For flows with a Mach number above 0.3 compressible effects become non negligible. A compressible solver must therefore be used. In this case we use *rhoCentralFoam*. The control file must therefore be set accordingly:

```python
def write_control_dict(case, n_iter):
        """Sets up the control dictionary.
        In this example we use the rhoCentralFoam compressible solver"""

        # Control dict from tutorial
        control_dict = {
                'application': 'rhoCentralFoam',
                'startFrom': 'startTime',
                'startTime': 0,
                'stopAt': 'endTime',
                'endTime': n_iter,
                'deltaT': 0.001,
                'writeControl': 'runTime',
                'writeInterval': 1,
                'purgeWrite': 0,
                'writeFormat': 'ascii',
                'writePrecision': 6,
                'writeCompression': 'off',
                'timeFormat': 'general',
                'timePrecision': 6,
                'runTimeModifiable': True,
                'adjustTimeStep' : 'no',
                'maxCo' : 1,
                'maxDeltaT' : 1e-6,
        }

        with case.mutable_data_file(FileName.CONTROL) as d:
                d.update(control_dict)
```

The mesh needs to be set up using the *blockMeshDict*. The mesh consits of three blocks in order to model the upstream and downstream portions as well as the wedge itself. The numbering order in which the vertices are set is very important! We declare a block via:

```
hex', [0, 7, 2, 1, 8, 15, 10, 9], [40, 40, 1],
            'simpleGrading', [1, 1, 1]
```

The first line declares which vertices make up the corners of the block. This explanation best describes the order in which the vertices should be listed. The second part of the statement describes the cell density within the block in each of the three directions. The last part is used when we want the mesh density to vary within the block.

We must also set the thermodynamic properties of the gas. In this case the properties have been chosen so that the gas has a ratio of specific heats of 1.4 and that, if the temperature is 1K, then the speed of sound is 1m/s. As this is a commonly used fluid it can be done using the *write_standard_thermophysical_properties* function in the *firefish.case* module. We do this via:

```
write_standard_thermophysical_properties(case, StandardFluid.DIMENSIONLESS_AIR)
```

As this is a compressible flow we must also set the initial value of the temperatue field. Notice also that for the velocity we have set a slip boundary condition on the solid walls. This is because we are using an inviscid solver. When we move to a viscous solver we must set a no slip boundary condition on the solid walls.

```
def write_initial_conditions(case):
        """Sets the initial conditions"""
        # Create the p initial conditions
        p_file = case.mutable_data_file(
                '0/p', create_class=FileClass.SCALAR_FIELD_3D
        )
        with p_file as p:
                p.update({
                        'dimensions': Dimension(1, -1, -2, 0, 0, 0, 0),
                        'internalField': ('uniform', 1),
                        'boundaryField': {
                                'inlet' : {'type' : 'fixedValue', 'value' : 'uniform 1'},
                                'outlet': {'type': 'zeroGradient'},
                                'fixedWalls': {'type': 'zeroGradient'},
                                'frontAndBack': {'type': 'empty'},
                        },
                })

        # Create the U initial conditions
        U_file = case.mutable_data_file(
                '0/U', create_class=FileClass.VECTOR_FIELD_3D
        )
        with U_file as U:
                U.update({
                        'dimensions': Dimension(0, 1, -1, 0, 0, 0, 0),
                        'internalField': ('uniform', [2, 0, 0]),
                        'boundaryField': {
                                'inlet' : {'type' : 'fixedValue',
                                                   'value' : ('uniform', [2, 0, 0])},
                                'outlet': {
                                        'type': 'zeroGradient'
                                },
                                'fixedWalls': {
                                        'type': 'slip'
                                },
                                'frontAndBack': {'type': 'empty'},
                        },
                })
                # Create the T initial conditions
        T_file = case.mutable_data_file(
                '0/T', create_class=FileClass.SCALAR_FIELD_3D
```

```
        )
        with T_file as T:
                T.update({
                        'dimensions': Dimension(0, 0, 0, 1, 0, 0, 0),
                        'internalField': ('uniform', 1),
                        'boundaryField': {
                                'inlet' : {'type' : 'fixedValue', 'value' : ('uniform', 1)},
                                'outlet': {
                                        'type': 'zeroGradient'
                                },
                                'fixedWalls': {
                                        'type': 'zeroGradient'
                                },
                                'frontAndBack': {'type': 'empty'},
                        },
                })
```

The example script includes a `main()` function which performs all of these steps. A boolean value can be passed to `main()` in order to reduce the number of iterations and so speed up automatic testing.

```
def main(case_dir='wedge', n_iter=10):
    #Try to create new case directory
    case = create_new_case(case_dir)
    # Add the information needed by blockMesh.
    write_control_dict(case, n_iter)
    write_block_mesh_dict(case)
    #we generate the mes1h
    case.run_tool('blockMesh')
    #we prepare the thermophysical and turbulence properties
    write_standard_thermophysical_properties(case, StandardFluid.DIMENSIONLESS_AIR)
    write_turbulence_properties(case)
    #we write fvScheme and fvSolution
    write_fv_schemes(case)
    write_fv_solution(case)
    write_initial_conditions(case)
    case.run_tool('rhoCentralFoam')
```

After the example script is run, *paraFoam* may be run to inspect the solution.

CHAPTER 3

Snappy Hex Mesh Example

The `snappy_hex_example.py` script provides an example of running snappyHexMesh.

In order to be able to run snappyHexMesh we need to set up a control dictionary even though it plays no part in the actual mesh generation process. Likewise, in order to use paraFoam, we need to set up fvSchemes and fvSolution.

For snappyHexMesh to work we must have an underlying block mesh. This is generated in *make_block_mesh* and follows the same procedure as in previous examples.

For the actual mesh generation we first of all load a geometry using the new Geometry class:

```
rocket = Geometry(GeometryFormat.STL,'example.stl','example',case)
```

The idea behind this class is that, when we support more geometry file types in the future, it will abstract away the need to worry about wether something is an STL or OBJ etc.

We next scale and translate the rocket:

```
rocket.scale(0.5);
rocket.translate([0.5,2,2])
```

The Geometry class also contains mesh quality settings for this particular geometry.

Now that the geometry has been loaded we use it to initialise the *SnappyHexMesh* class:

```
snap = SnappyHexMesh(rocket,4,case)
```

This creates a new SnappyHexMesh class based on the example geometry and with a surface refinement level of 4. The SnappyHexMesh class automatically sets the mesh generation settings to a set of default values. These can be altered:

```
snap.snap=True
snap.snapTolerance = 8;
snap.locationToKeep = [0.0012,0.124,0.19]
snap.addLayers=False
```

Once the mesh generator is set up we can make the mesh via the call:

```
snap.generate_mesh()
```

Several things happen all at once here: * Surface features are extracted from the geometry (saving the STL in the trisurfaces directory if it has not already done so) * The mesh quality settings are written to a dictionary * The snappyHexMesh dictionary is written using the attributes of the SnappyHexMesh class * snappyHexMesh is run as a tool within the case directory

Once this has run the mesh can be viewed via *paraFoam*

Join STL Example

The `join_stl_example.py` script provides an example of combining multiple STL files into a single geometry and then generating a mesh through snappyHexMesh. It is worth having a look at `snappy_hex_example.py` first in order to get a more detailed overview on how SnappyHexMesh works.

It is now very straightforwards to generate a mesh made up from multiple *.STL* files.

Firstly one needs to make a list containing the paths of each STL file:

```
path_list = ['STLS/nosecone.stl', 'STLS/upperTube.stl', 'STLS/lowerTube.stl', 'STLS/finCan.stl', 'STL
```

One then needs to make a list of the human-readable names that correspond to each file:

```
part_list = ['nosecone', 'upperTube', 'lowerTube', 'finCan', 'tail']
```

When examining the mesh in paraFoam or when getting force outputs it will be these names that appear.

Once this has been done the Geometry classes can be loaded by a single call of:

```
parts = load_multiple_geometries(GeometryFormat.STL,path_list,part_list,case)
```

This produces a list of *firefish.geometry.Geometry* objects which can scaled, translated and rotated independently as required using the normal geometry functions.

We now use this list of Geometry objects to initialise SnappyHexMesh:

```
snap = SnappyHexMesh(parts,4,case)
```

As in `snappy_hex_example.py`, we can now alter the settings of Snappy Hex Mesh by altering the attributes of the SnappyHexMesh class. Once we've updated these we generate the mesh via:

```
snap.generate_mesh()
```

This generates four different meshes: the blank block mesh, the castellated mesh, the snapped mesh and a mesh with layer addition. In order to use the final mesh as the starting point of our simulation we perform some trickery to delete the meshes we don't want and move the mesh we do want into the constant folder

```
def getTrueMesh(case):
    #the proper mesh is in the final time directory, delete the one in constant
    os.chdir(case.root_dir_path)
    call (["rm", "-r", "constant/polyMesh"])
    call (["mv", "0.002/polyMesh", "constant/"])
    call (["rm", "-r", "0.001/"])
    call (["rm", "-r", "0.002/"])
    os.chdir("../")
```

Kinematics Example

The `kinematics_example.py` script uses the new *firefish.kinematics* framework to implement the kinematics example in `kinematics_basis.py`

We firstly define a class that inherits from *firefish.kinematics.KinematicBody*. We want to model a cylindrical rocket whose principal moments of inertia vary as the rocket burns its motor. In order to vary the prinicpal moments of inertia automatically during the timestepping routine we must overload the *update_moi()* function.

```
class CylinderRocket(KinematicBody):
    """A rocket that is a cylinder with  evenly distributed mass"""
    def update_moi(self):
        radius = 0.3
        height = 2

        Ixx = (self.mass/12.0)*(3*radius**2 + height**2)
        Iyy = (self.mass/12.0)*(3*radius**2 + height**2)
        Izz = self.mass*radius**2 / 2.0

        self.MoI = [Ixx,Iyy,Izz]
```

In the main function we now undergo the time stepping routine. For each time step we must pass the forces acting on the rocket along with the torques to the routine. These forces and torques must be given in the body coordinate system. The example here burns the motor for fifty seconds and then lets it coast

```
def main():
    """Run through the simulation with a 50s motor burn"""
    initialMass = 100
    initialInertias = [0, 0, 0]
    rocket = CylinderRocket(initialMass,initialInertias)
    dt =1; duration = 100; gravity = 9.81;
    simulation = KinematicSimulation(rocket,gravity,duration,dt)
    while simulation.tIndex*dt <= duration:

        thrust = 0

        if simulation.tIndex*dt <= 50:
            thrust = 2000.0

        forces = [2.0, 0.0, thrust]
        torques = [0.0, 0.0, 0.0]
        mdot = 0.1

        i = simulation.tIndex
```

```
        simulation.time_step(forces,torques,mdot)

    return simulation.posits[100,2] #z position
```

Reference

6.1 OpenFOAM case directory manipulation

This module allows the building and manipulating OpenFOAM case directories.

OpenFOAM files are mapped into Python objects using the following conventions:

- Dictionaries map to python `dict`.
- Keyword data entries map to `tuple` when the number of data entries is greater than one. Otherwise the single data entry is the keyword's value.
- Lists are mapped to Python `list`.
- Dimension are represented via the *`Dimension`* type.

class `firefish.case.`**`Case`**(*root_dir_path*, *create=True*)
Object representing an OpenFOAM case on disk.

`root_dir_path`
path to case directory

`add_tri_surface`(*name*, *geom*, *clobber_existing=False*)
Add a triangulated surface to the case.

Adds the geometry specified in *geom* to the case under the `constant/triSurface` directory. The geometry is saved in STL format.

The geometry is added with the given name. If name is `foo`, for example, it will be saved with the filename `foo.stl`.

Note: Do not add the `.stl` extension to *name*. Future versions of this method may wish to allow other ways of specifying file format.

Parameters

- **`name`** (*`str`*) – name to save surface as
- **`geom`** (*`stl.mesh.Mesh`*) – geometry representing surface
- **`clobber_existing`** (*`bool`*) – if False, do not overwrite an existing file

Raises *`CaseAlreadyExists`* – if a surface with the given name already exists and *clobber_existing* was not True.

mutable_data_file (*path*, *create_class=<FileClass.DICTIONARY: 'dictionary'>*, *create=True*)
A context manager representing a dict. Changes to the dict are written back to disk.

Parameters

- **path** (*str or FileName*) – relative path to dictionary
- **create_class** (*str or FileClass*) – specify the class of created files
- **create** (*bool*) – create file if it does not exist

```
>>> case = getfixture('tmpcase')
>>> with case.mutable_data_file('system/blockMeshDict') as d:
...     d['boundary'] = { 'foo': { 'type': 'empty' } }
>>> case.read_data_file('system/blockMeshDict')['boundary']['foo']
{'type': 'empty'}
```

```
>>> case = getfixture('tmpcase')
>>> items = { 'application': 'simpleFoam', 'description': 'mycase' }
>>> with case.mutable_data_file(FileName.CONTROL) as d:
...     d.update(items)
>>> control = case.read_data_file(FileName.CONTROL)
>>> control['application']
'simpleFoam'
>>> control['description']
'mycase'
```

read_data_file (*path*)
Read the contents of the control dictionary.

Parameters **path** (*str or FileName*) – relative path to dictionary

Raises IOError – the control dictionary could not be opened

run_tool (*tool_name*, *flags=''*)
Run an OpenFOAM tool on the case.

It is assumed that the tool accepts the standard "-case" argument.

Parameters **tool_name** (*str*) – name of tool to run (e.g. "icoFoam")

Raises

- *CaseToolRunFailed* – if the tool exits with an error
- OSError – if the tool could not be started

exception firefish.case.**CaseAlreadyExists**
Some resource already existed.

exception firefish.case.**CaseDoesNotExist**
A case directory did not exist when we expected it to.

exception firefish.case.**CaseException**
Base class for exceptions raised by firefish.case module.

exception firefish.case.**CaseToolRunFailed**
There was a failure running a tool on a case directory.

class firefish.case.**Dimension** (**dims*)
Represents a value's dimensions in OpenFOAM cases.

A dimension represents the units used to describe a physical value e.g. one might measure velocity in metres per second or kilometres per hour.

In OpenFoam these dimensions must be built up from the standard SI units of kilograms, metres, seconds, Kelvins, moles, Amps and candelas.

To construct a dimension we raise each unit to a given exponent and multiply them all together e.g. metres per second is m s $^{-1}$

Some commonly used units such as the Newton are not SI. We must therefore express them as a combination of SI units e.g. we know F=ma and so the Newton must be kg m s $^{-2}$

In order to do this in OpenFoam we must pass it a tuple containing the list of exponents for each fundamental SI unit. These are given in the order *kg, m, s, K, mol, A, cd*.

e.g. for acceleration (m s $^{-2}$)

```
>>> d = Dimension(0, 1, -2, 0, 0, 0, 0)
```

e.g. for pressure (kg m $^{-1}$ s $^{-2}$)

```
>>> d = Dimension(1, -1, -2, 0, 0, 0, 0)
```

Parameters **`PFDataStructs.Dimension`** – A tuple containing the exponents to be used for each SI unit. These are given in the order *kg, m ,s, K, mol, A, cd*

Example usage:

```
>>> d = Dimension(0, 1, -2, 0, 0, 0, 0)
>>> str(d) # PyFOAM data file representation
'[ 0 1 -2 0 0 0 0 ]'
>>> d.unit
'ms^-2'
>>> repr(d)
'firefish.case.Dimension(0, 1, -2, 0, 0, 0, 0)'
```

The class also supports indexing and the sequence property

```
>>> d[2]
-2
>>> [v+1 for v in d]
[1, 2, -1, 1, 1, 1, 1]
>>> d[0] = 2
>>> d.unit
'kg^2ms^-2'
```

class `firefish.case.`**`FileClass`**
Well known OpenFOAM dictionary classes.

class `firefish.case.`**`FileName`**
An enumeration of well known OpenFOAM file locations.

class `firefish.case.`**`MeshGenerator`**
An eumeration of different mesh generation methods

class `firefish.case.`**`StandardFluid`**
An enumeration of commonly used fluids

`AIR`
generates the recommended OpenFoam thermophysicalProperties for air. The dictionary produced is taken from the rhoCentralFoam shock tube tutorial

`DIMENSIONLESS_AIR`
generates a normalised gas whith gamma=7/5 and with the property that at 1 temperature unit the speed of sound is 1 velocity unit. The dictionary produced is taken from the rhoCentralFoam wedge15Ma5 tutorial

`firefish.case.`**`read_data_file`**(*path*)
Read and parse an OpenFOAM dict into a Python dictionary.

Parameters **`path`** (*`str`*) – path to the OpenFOAM dict on disk

Returns A dict representing a Python transliteration of the dict.

Raises `IOError` – the path could not be read from

`firefish.case.`**`write_standard_thermophysical_properties`**(*case*, *fluid*)
" Writes a thermophysicalProperties dict in the given case for the specified fluid.

Parameters

- **`case`** (`firefish.case.Case`) – the case in which to write the dict
- **`fluid`** (`firefish.case.StandardFluid`) – the fluid to use

6.2 IO

This module contains IO utility functions for amateur rocketry file formats.

class `firefish.io.`**`Engine`**
An individual engine record.

The thrust curve data is represented by a pandas DataFrame object, with the following columns: time (seconds), force (Newtons), mass (grams).

`manufacturer`
A string containing the manufacturer, or None

`code`
A string containing the maufacturer's product code, or None

`comments`
A string containing any comments, or None

`data`
A pandas DataFrame, see above

exception `firefish.io.`**`RSEParseError`**
Raised when there is an error parsing a RockSim file.

`firefish.io.`**`rse_load`**(*path*)
Load a RockSim format engine database from disk.

Parameters **`path`** (*`str`*) – path name to .rse file

Returns A list of Engine instances.

Raises *`RSEParseError`* – when the .rse file is invalid

6.3 Geometry manipulation

This module deals with the loading, saving and manipulation of geometry.

Most manipulation functions deal with instances of `stl.mesh.Mesh`. See the numpy-stl documentation for more information.

class `firefish.geometry.`**`Geometry`**(*geomType*, *path*, *name*, *case*)
This class encapsulates the geometry functionality

extract_features()
Extracts surface features from geometry using the surfaceFeatureExtract tool

recentre()
Recentres the geometry

save(*path*)
copies the stl from source directory into path

Args: path: the path to copy the stl file into

scale(*factor*)
Scales geometry by factor

Parameters **factor** – The factor to scale the gometry by

translate(*delta*)
Translates geometry by delta

Parameters **delta** – The vector to translate the geometry by

class firefish.geometry.**GeometryFormat**
An enumeration of different geometry formats

class firefish.geometry.**MeshQualitySettings**
Controls the mesh quality settings associated with the gometry

write_settings(*case*)
Writes the quality settings to a separate dict that can be included

firefish.geometry.**load_multiple_geometries**(*geomType*, *paths*, *names*, *case*)
Loads multiple geometries of the same type and returns as a list

Parameters

- **geomType** (firefish.geometry.GeometryFormat) – indicates what type these geometries are
- **paths** – list of paths to each geometry file eg. stls/foo.stl
- **names** – the list of names of each geometry e.g. body, fin etc.
- **case** (firefish.case.Case) – the case to place each geometry in

firefish.geometry.**stl_bounds**(*geom*)
Compute the bounding box of the geometry.

Parameters **geom** (*stl.mesh.Mesh*) – STL geometry

Returns A pair giving the minimum and maximum X, Y and Z co-ordinates as three-dimensional array-likes.

firefish.geometry.**stl_copy**(*geom*)
Copy a geometry.

Use this function sparingly. Geometry can be quite heavyweight as data structures go.

Parameters **geom** (*stl.mesh.Mesh*) – STL geometry

Returns A deep copy of the geometry.

firefish.geometry.**stl_geometric_centre**(*geom*)
Compute the centre of the bounding box.

Parameters **geom** (*stl.mesh.Mesh*) – STL geometry

Returns An array like giving the X, Y and Z co-ordinates of the centre.

`firefish.geometry.`**`stl_load`**(*path*)
Convenience function to load a `stl.mesh.Mesh` from disk.

Note: The `save()` method on `stl.mesh.Mesh` may be used to write geometry to disk.

Parameters **`path`** (*`str`*) – pathname to STL file

Returns an new instance of `stl.mesh.Mesh`

`firefish.geometry.`**`stl_recentre`**(*geom*)
Centre a geometry such that its bounding box is centred on the origin.

This function modifies the passed geometry.

Equivalent to:

```
translate(geom, -geometric_centre(geom))
```

Parameters **`geom`** (*`stl.mesh.Mesh`*) – STL geometry

Returns The passed geometry to allow for easy chaining of calls.

`firefish.geometry.`**`stl_scale`**(*geom*, *factor*)
Scale geometry by a fixed factor.

This function modifies the passed geometry. If the scale factor is a single scalar it is applied to each axis. If it is a 3-vector then the elements specify the scaling along the X, Y and Z axes.

Parameters

- **`geom`** (*`stl.mesh.Mesh`*) – STL geometry
- **`factor`** (*`scalar or array like`*) – scale factor to apply

Returns The passed geometry to allow for easy chaining of calls.

`firefish.geometry.`**`stl_translate`**(*geom*, *delta*)
Translate a geometry along some vector.

This function modifies the passed geometry.

Parameters

- **`geom`** (*`stl.mesh.Mesh`*) – STL geometry
- **`delta`** (*`array like`*) – 3-vector giving translation in X, Y and Z

Returns The passed geometry to allow for easy chaining of calls.

6.4 Mesh generation

This module provides tools for building and running SnappyHexMesh

class `firefish.meshsnappy.`**`SnappyHexMesh`**(*geometries*, *surfaceRefinement*, *case*)
Encapsulates all the snappyHexMesh settings

`add_mesh_features`(*file_list*)
test function which runs add_features in order to write the surfaceFeatureExtractDict

generate_mesh()
Generates the mesh

Note: This extracts surface features, writes the main SHM dict, a mesh quality dict and then runs SHM. We assume that an underlying block mesh has already been produced

write_snappy_dict()
Writes the SHM dictionary

Note: This is called by SnappyHexMesh when it generates the mesh

6.5 Fin-flutter

Calculation of fin flutter vs. altitude.

The transonic flutter velocity code comes from "Peak of flight" newsletter issue 291, which is itself a modified version of the equation in NACA paper 4197.

The supersonic flutter criterion is from a thesis by J. Simmons at the Air Force Institute of Technology, Ohio. (AFIT/GSS/ENY/09-J02), the torsional and bending frequencies have to be calculated for different geometries using finite element analysis in Solidworks.

This module provides a simple API for computing fin-flutter velocity as a function of altitude. These can then be plotted. For example:

```
import matplotlib.pyplot as plt
import numpy as np
from firefish import finflutter

zs = np.linspace(0, 50000, 200)
ps, _, ss = finflutter.model_atmosphere(zs)
vs = finflutter.flutter_velocity_transonic(ps, ss, root_chord=20, +
    tip_chord=10, semi_span=10, thickness=0.2)

plt.plot(zs * 1e-3, vs)
plt.grid()
plt.title('Flutter velocity versus altitude')
plt.xlabel('Altitude [km]')
plt.ylabel('Flutter velocity [ms${}^{-1}$]')
plt.show()
```

firefish.finflutter.**flutter_velocity_supersonic**(*air_densities*, *torsional_frequency*, *bending_frequency*, *mass*, *semi_span*, *radius_of_gyration*, *distance_to_COG*, *Mach_number*)

Calculate transonic flutter velocities for a given fin design. The equation is valid for freestream flow in the supersonic regime (>~M2.5)

Fin analysis have to be done for Solidworks in order to find the frequencies for bending and torsional modes, as well as the radius_of_gyration and distance_to_COG. Torsional and bending frequency are in rad/s, the semi-span, radius of gyration, and distance to COG will be given in metres.

```
>>> import numpy as np
>>> zs = np.linspace(0, 30000, 100)
```

```
>>> ps, ts, ss = model_atmosphere(zs)
>>> rhos = (ps/1000) / (0.2869 * (ts + 273.1))
>>> vels = flutter_velocity_supersonic(rhos, 380, 104, 1, 0.1, 0.2, 0.1, 3)
>>> assert vels.shape == ps.shape
```

Parameters

- **semi_span** – fin semi-span (m)
- **air_densities** – 1-d array of air density in kg/m^3 (np.array)
- **frequency** (*torsional*) – uncoupled torsional frequency (rad/s)
- **bending_frequency** – uncoupled bending frequency of the fin (rad/s)
- **mass** – mass of fin (kg)
- **Mach_number** – mach number of rocket
- **distance_to_COG** – distance of COG to axis of rotation (m)
- **radius_of_gyration** – distance at which all the mass of the fin can be though to be concenreated, =sqrt(I/M)

Returns A 1-d array containing corresponding flutter velocities in m/s.

`firefish.finflutter.`**`flutter_velocity_transonic`**(*pressures*, *speeds_of_sound*, *root_chord*, *tip_chord*, *semi_span*, *thickness*, *shear_modulus=26200000000.0*)

Calculate transonic flutter velocities for a given fin design. The equation is valid if the rocket is travelling at < M2.5 at the given altitude.

Fin dimensions are given via the root_chord, tip_chord, semi_span and thickness arguments. All dimensions are in centimetres.

Use shear_modulus to specify the shear modulus of the fin material in Pascals.

```
>>> import numpy as np
>>> zs = np.linspace(0, 30000, 100)
>>> ps, _, ss = model_atmosphere(zs)
>>> vels = flutter_velocity_transonic(ps, ss, 20, 10, 10, 0.2)
>>> assert vels.shape == ps.shape
```

Parameters

- **pressures** (*np.array*) – 1-d array of atmospheric pressures in Pascals
- **speeds_of_sound** (*np.array*) – 1-d array of speeds of sound in m/s
- **root_chord** – fin root chord (cm)
- **tip_chord** – fin tip chord (cm)
- **semi_span** – fin semi-span (cm)
- **thickness** – fin thickness (cm)
- **shear_modulus** – fin material shear modulus (Pascals)

Returns A 1-d array containing corresponding flutter velocities in m/s.

`firefish.finflutter.`**`model_atmosphere`**(*altitudes*)

Model atmospheric pressure, temperature and speed of sound.

Parameters **altitudes** (*np.array*) – 1-d array of geopotential altitudes in metres

Returns A triple giving corresponding 1-d arrays of estimated pressure, temperature and speed of sound. Units are Pascals, Celsius and m/s respectively.

```
>>> import numpy as np
>>> zs = np.linspace(0, 30000, 100)
>>> ps, ts, ss = model_atmosphere(zs)
>>> assert ps.shape == zs.shape
>>> assert ts.shape == zs.shape
>>> assert ss.shape == zs.shape
```

6.6 Kinematics

This module deals with kinematic models used in rocket simulation

class firefish.kinematics.**KinematicBody**(*mass*, *inertias*)

Encapsulates information about the kinematic body

update_moi()

We update moments of inertias. Any class inheriting KinematicBody must overload this if it has non-constant moments of inertia

class firefish.kinematics.**KinematicSimulation**(*body*, *gravity*, *duration*, *dt*)

Encapsulates all the simulation logic and time stepping

time_step(*forces*, *torques*, *mdot*)

Performs a single time step

Parameters

- **forces** (*[float]*) – A list of the forces on the body in N in the form [Fx,Fy,Fz]
- **torques** (*[float]*) – A lst of the moments acting on the body in Nm in the form [Mxx,Myy,Mzz]
- **modt** (*float*) – Mass flow rate of the motor. i.e. 0.1 implies the motor is ejection 0.1 kgs^-1

CHAPTER 7

CU Spaceflight Simulation Software

f

firefish.case, 19
firefish.finflutter, 25
firefish.geometry, 22
firefish.io, 22
firefish.kinematics, 27
firefish.meshsnappy, 24

A

add_mesh_features() (firefish.meshsnappy.SnappyHexMesh method), 24
add_tri_surface() (firefish.case.Case method), 19
AIR (firefish.case.StandardFluid attribute), 21

C

Case (class in firefish.case), 19
CaseAlreadyExists, 20
CaseDoesNotExist, 20
CaseException, 20
CaseToolRunFailed, 20
code (firefish.io.Engine attribute), 22
comments (firefish.io.Engine attribute), 22

D

data (firefish.io.Engine attribute), 22
Dimension (class in firefish.case), 20
DIMENSIONLESS_AIR (firefish.case.StandardFluid attribute), 21

E

Engine (class in firefish.io), 22
extract_features() (firefish.geometry.Geometry method), 22

F

FileClass (class in firefish.case), 21
FileName (class in firefish.case), 21
firefish.case (module), 19
firefish.finflutter (module), 25
firefish.geometry (module), 22
firefish.io (module), 22
firefish.kinematics (module), 27
firefish.meshsnappy (module), 24
flutter_velocity_supersonic() (in module firefish.finflutter), 25
flutter_velocity_transonic() (in module firefish.finflutter), 26

G

generate_mesh() (firefish.meshsnappy.SnappyHexMesh method), 24
Geometry (class in firefish.geometry), 22
GeometryFormat (class in firefish.geometry), 23

K

KinematicBody (class in firefish.kinematics), 27
KinematicSimulation (class in firefish.kinematics), 27

L

load_multiple_geometries() (in module firefish.geometry), 23

M

manufacturer (firefish.io.Engine attribute), 22
MeshGenerator (class in firefish.case), 21
MeshQualitySettings (class in firefish.geometry), 23
model_atmosphere() (in module firefish.finflutter), 26
mutable_data_file() (firefish.case.Case method), 19

R

read_data_file() (firefish.case.Case method), 20
read_data_file() (in module firefish.case), 22
recentre() (firefish.geometry.Geometry method), 23
root_dir_path (firefish.case.Case attribute), 19
rse_load() (in module firefish.io), 22
RSEParseError, 22
run_tool() (firefish.case.Case method), 20

S

save() (firefish.geometry.Geometry method), 23
scale() (firefish.geometry.Geometry method), 23
SnappyHexMesh (class in firefish.meshsnappy), 24
StandardFluid (class in firefish.case), 21
stl_bounds() (in module firefish.geometry), 23
stl_copy() (in module firefish.geometry), 23
stl_geometric_centre() (in module firefish.geometry), 23
stl_load() (in module firefish.geometry), 23
stl_recentre() (in module firefish.geometry), 24

stl_scale() (in module firefish.geometry), 24
stl_translate() (in module firefish.geometry), 24

T

time_step() (firefish.kinematics.KinematicSimulation method), 27
translate() (firefish.geometry.Geometry method), 23

U

update_moi() (firefish.kinematics.KinematicBody method), 27

W

write_settings() (firefish.geometry.MeshQualitySettings method), 23
write_snappy_dict() (firefish.meshsnappy.SnappyHexMesh method), 25
write_standard_thermophysical_properties() (in module firefish.case), 22