README

MATVEICHEV Alexey

November 6, 2017

Contents

1 Solar pond simulation

Dimensions are based upon Ormat's solar pond at Ein Boqeq (`http://ieeexplore.ieee.org/document/6369581/`):

area 75347 square feet (7009.15 m^2)

depth 8 feet (2.44 m)

1.1 Mesh

To avoid `blockMesh` mess, all meshes are prepared in Gmsh (`http://gmsh.info`).

1.1.1 Cuboid

Mesh with square base and 2.44 m height. Length of the square side is 83.72 m. To have ability to dense mesh towards boundaries, square sides are split into two parts.

1.1.2 Cylinder

Radius of the cylinder is 47.2 m, height is 2.44 m.

1.2 Boundary conditions

For the simulation we need to set boundary conditions for `T`, `U`, `p_rgh`. If we need turbulence, `k`, `epsilon`, and `nut` boundary conditions should also be set.

1.2.1 Top boundary

For scalar fields (`T`, `p_rgh`, `k`, `epsilon`, `nut`) boundary condition is zero gradient (though for turbulence this approach is arguable), for vector fields it is slip.

1.2.2 Bottom boundary

For velocity BC is noSlip, for temperature and pressure it is zero gradient, for `k`, `epsilon`, and `nut` wall functions are used.

1.3 Initial conditions

Initially pond is at rest (U = 0), with mean temperature of 25 degrees Celsius. To create motion inside the pond artificial blob of warm water is placed in the middle of the reservoir.

1.4 Top surface heat transfer models

Models are taken from Fresh Surface Water - Volume II

1.4.1 Radiative heating by solar energy

The simplest of all expressions:

$$q_s = Q_{sm}\,(1 - R_t) \tag{1}$$

Where Q_{sm} is solar radiation flux at the earth surface, R_t is reflectivity of water surface (0.03). Q_{sm} is user supplied parameter, which is in general in the range $150 - 300\ W/m^2$. For simulation value of $164\ W/m^2$ is taken (`http://zebu.uoregon.edu/disted/ph162/l4.html`).

1.4.2 Radiative heat losses

Radiation is described using simple Stefan-Boltzmann law with constant emissivity. So heat flux ($J \cdot m^{-2} \cdot s^{-1}$) from surface of the water is given by:

$$q_r = -\sigma \varepsilon T_s^4 \tag{2}$$

Where ε is water emissivity, for the simulations value of 0.97 is used.

1.4.3 Evaporation

Simple empirical law is used to describe evaporation.

$$q_e = -\left(A\left(T_s - T_a\right) + b_0 u\right)\left(e_s - e_2\right) \tag{3}$$

Where
$A = 2.7\ W \cdot m^{-2} \cdot mbar^{-1} \cdot K^{-1/3}$
$b_0 = 3.2\ W \cdot m^{-2} \cdot mbar^{-1} \cdot (m/s)^{-1}$
e_s and e_2 is vapour pressure at the water surface and 2 meter above it correspondingly (in mbar).

For calculation of vapour pressure Buck equation is used:

$$P = 0.61121 \cdot \exp\left(\left(18.678 - \frac{T}{234.5}\right)\left(\frac{T}{257.14 + T}\right)\right) \tag{4}$$

T should be in Celsius, P is in kPa.

1.5 fvOptions

We need to describe heat losses at the top surface through radiation and evaporation. Since it is impossible to account for all possible source terms in equations within single solver, fvOption framework was introduced in OpenFOAM.

In general, conservation law is expressed as:

$$\frac{\partial \psi}{\partial t} + \ldots = \sum_i S_i \tag{5}$$

And all those S_i can be described as user configured fvOptions. The most flexible of them is `codedSource`, which will be used for radiation and evaporation heat transfer at the top surface.

1.5.1 Implementation

With the flexibility comes a need to write C++ in rather inconvenient environment: OpenFOAM dictionary. Our fvOptions are just source terms in temperature equation, so, fortunately, we need to write just `addSup` method, which goes into `codeAddSup` entry of configuration, which goes into `constant/fvOptions` file. Alternatively we can implement our fvOptions as a library, yet this way has its own inconveniences.

In general, radiative heat transfer or evaporation are mentally treated as boundary condition, yet within OpenFOAM it is much easier to think of them as surface source term in corresponding equation.

```
radiation
{
  type scalarCodedSource;
  active yes;
  name radiativeHeatExchange;
```

fvOption description is stored in a dictionary and starts with type, which, in case of temperature, is `scalarCodedSource`. It is `active` and is called `radiativeHeatExchange`. The name can be arbitrary and used for code folder name and for naming source coefficients dictionary.

Next come two dictionaries: `scalarCodedSourceCoeffs`, which describes source and contain code, and `radiativeHeatExchangeCoeffs`, which can contain arbitrary settings for the source, which later can be accessed in code through `coeffs()` method. Latter allows to avoid recompilation in case of model coefficients change.

`scalarCodedSourceCoeffs` dictionary contains the following keys:

- `selectionMode`, which can be `all` or the name of cell set if we would like to apply source in certain area. In our case this parameter is ignored, since we select cells adjacent to a boundary.
- `fields`, which defines list of field name, which fvOption affects.
- `codeInclude`, which goes after the standard includes in the code template.

- `codeCorrect`, which should contain `correct` method code, if fvOption does corrections.
- `codeAddSup`, which contains code of `addSup` method. And this is the only method, necessary for source term.
- `codeSetValue`, which contains code of `setValue` method.
- `code`, which is used to trigger recompilation of the fvOption if any of the above mentioned code sections are modified.

`radiativeHeatExchangeCoeffs` dictionary contains `selectionMode` key, which repeats one from previous dictionary, and arbitrary model parameters in key-value form.

Implementation of the source term is quite straight-forward.

```
using constant::physicoChemical::sigma;

if (not isActive())
  return;
```

The first line imports sigma constant into current name space, the second line returns from the method if source is not active.

```
// Looking up model parameters
scalar Qsm = coeffs().lookupOrDefault<scalar>("Qsm", 164.0);
scalar rhow = coeffs().lookupOrDefault<scalar>("rho", 1000);
scalar Cp = coeffs().lookupOrDefault<scalar>("Cp", 4200);
scalar Rt = coeffs().lookupOrDefault<scalar>("Rt", 0.03);
scalar ew = coeffs().lookupOrDefault<scalar>("ew", 0.97);
```

The source looks up model parameters, such as average solar radiation heat flux at the water surface and water thermophysical properties.

```
// Getting source vector from equation matrix
scalarField& src = eqn.source();
const volScalarField& T = eqn.psi();
```

Then source extracts temperature equation source vector and temperature field.

```
word top_patch_name = coeffs().lookupOrDefault<word>("patch", "top");
const fvPatch& pp = mesh().boundary()[top_patch_name];
forAll(pp, i) {
  label cell_i = pp.faceCells()[i];
  scalar Ai = pp.magSf()[i];
  scalar Ts = T[cell_i];

  // Heating
  src[cell_i] -= Qsm*(1 - Rt)*Ai/rhow/Cp;

  // Cooling
  src[cell_i] -= -sigma.value()*ew*pow4(Ts)*Ai/rhow/Cp;
}
```

Then source looks up top patch, iterates over it, and adds source terms into cells adjacent to the faces of the patch. And that is all. Source code for the heat losses through evaporation is a little bit more complicated, since it has to calculate vapour pressure at the surface of the water, which depends on the temperature.

1.6 Using example case

For ease of execution four shell scripts are supplied with the case: `Prepare.cuboid`, `Prepare.cylinder`, `Run`, and `Stop`. First two prepare case for execution: generate mesh, correct `boundary` dictionary, copy `fvSchemes` and `fvSolution` corresponding to the case. `Run` executes case locally. Finally `Stop` script kills solver process on local machine.