User-defined Run-time Customisation for Eilmer Simulations

Rowan Gollan, Peter Jacobs and Ingo Jahn

19 July 2018

Outline

- User-defined hooks in Eilmer: why and what
- Implementation
    - Mental model of interaction
    - Contracts between user and calling code
    - Tour of helper variables, functions and modules
    - Some restrictions/warnings on use
- Examples
    - User-defined source terms
    - User-defined boundary conditions AND user-defined grid motion
- Troubleshooting tips

The Why of User-defined Customisation

User-defined hooks are pieces of customised code that users can build to do a certain modelling job that isn't available in the standard implementation. In Eilmer, users write the custom code in Lua.

From the user's perspective:

- Provide an easy interface to simulation customisation without needing to learn D programming and details of the compilation process
- Provide a means to customise modelling that does not rely on developer's timeline
- Provide a means to prototype and test models that might later be included in the standard modelling kit

From the developer's perspective:

- Delivers flexibility of modelling to users
- Allows one-off experimentation of ideas without major changes to D source code infrastructure
- Allows one-off modelling jobs without the ongoing maintenance burden of inclusion in the mainline source code
- Simplifies addition of new modelling features in the standard kit if users can provide a debugged and tested implementation as a user-defined hook

The What of User-defined Customisation

- User-defined functions are called at particular points in the simulation to do some customisation of the modelling
- The functions are defined in a Lua script
- The customisations allow modifications to source terms, boundary conditions and prescription of grid motion.
- There is also a supervisory user-defined function available

Steps to use:

1. Build a Lua script that defines a customisation function according to certain rules[1]
2. In the main input script, configure the simulation program to look for and use your customisation script

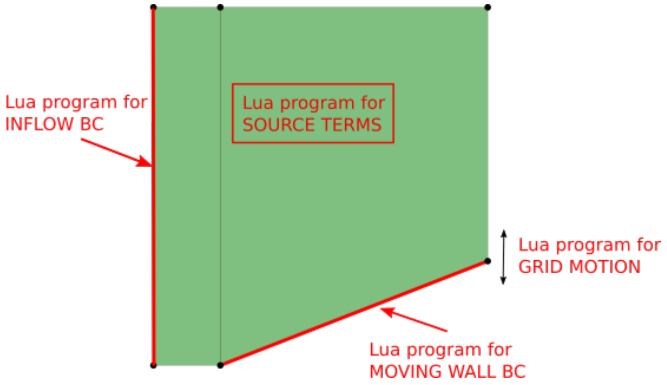
[1] See the appendix of the Eilmer user manual

Implementation: mental model for user

Some things for the user to keep in mind:

- Each “customisation” you apply is its own isolated Lua program
- Each “customisation” is loaded at the start of the simulation (initialisation)
    - You may do some initial configuration of your customisation during start-up
    - Alternatively, you can wait until the function is first called
- On each timestep, Eilmer looks for the specific customisation function and calls it, possibly many times, expecting a particular return value
- Aside from the specific function, Eilmer does not care what else is in your script. It will silently ignore anything superfluous to the job at hand
- The “customisation” programs are reentrant and hold state between calls. If you change a global value in your program, that change will persist across calls.

Implementation: mental model for user


Lua program for INFLOW BC
Lua program for SOURCE TERMS
Lua program for GRID MOTION
Lua program for MOVING WALL BC

- Each “customisation” you apply is its own isolated Lua program
- Each “customisation” is loaded at the start of the simulation (initialisation)
- On each timestep, Eilmer looks for the specific customisation function and calls it, possibly many times, expecting a particular return value
- Aside from the specific function, Eilmer does not care what else is in your script. It will silently ignore anything superfluous to the job at hand
- The “customisation” programs are reentrant and hold state between calls. If you change a global value in your program, that change will persist across calls.

The user-defined function contract

```
function specifiedName(args)
    -- do something interesting
    -- perhaps make use of args
    -- then return a value


    return requiredVal
end
```

The user-defined function contract

1. Eilmer sets up **args** specific to location in time and space

```
function specifiedName(args)
    -- do something interesting
    -- perhaps make use of args
    -- then return a value


    return requiredVal
end
```

The user-defined function contract

1. Eilmer sets up `args` specific to location in time and space
2. Eilmer looks for and executes a function of *`specifiedName`*, passing `args`

```
function specifiedName(args)
    -- do something interesting
    -- perhaps make use of args
    -- then return a value


    return requiredVal
end
```

The user-defined function contract

1. Eilmer sets up `args` specific to location in time and space
2. Eilmer looks for and executes a function of `specifiedName`, passing `args`

```
function specifiedName(args)
    -- do something interesting
    -- perhaps make use of args
    -- then return a value


    return requiredVal
end
```

such as:

`sourceTerms`
`ghostCells`

The user-defined function contract

1. Eilmer sets up `args` specific to position in time and space
2. Eilmer looks for and executes a function of `specifiedName`, passing `args`

```
function specifiedName(args)
    -- do something interesting
    -- perhaps make use of args
    -- then return a value


    return requiredVal
end
```

3. User specifies calculations in function body then returns a `requiredVal`

The user-defined function contract

1. Eilmer sets up **args** specific to position in time and space
2. Eilmer looks for and executes a function of ***specifiedName***, passing **args**

```
function specifiedName(args)
    -- do something interesting
    -- perhaps make use of args
    -- then return a value


    return requiredVal
end
```

3. User specifies calculations in function body then returns a ***requiredVal***
4. Eilmer makes use of ***requiredVal*** in simulation

Tour of helper variables, functions and modules

Eilmer defines some objects at global scope inside the Lua customisation programs. Some examples are:

`nFluidBlocks`	number of fluid blocks in simulation domain
`gmodel`	access to gas model functions used in current simulation
`sampleFluidCell()`	function to give properties of a cell in the simulation domain
`sampleFluidFace()`	as above but properties at face
`Vector3`	object from Eilmer geometry package
`Matrix`	object from Eilmer bare-bones linear algebra module

These can be used to make your Lua customisations rather general. For example, if you change the gas model in your simulation, you don't need to hard-code the equation-of-state expressions: just use Eilmer's gas model functions.

Some restrictions/warnings on use

- Remember that the customisations run in isolation (in their own sandbox). Don't expect changes in one script to be reflected in another.
- The sampling functions that give information about blocks, cells and interfaces only work for blocks in the memory of the process. So, for MPI simulations, you cannot get information about blocks and cells running in a different process. In fact, you might get a segmentation fault if you try this.
- There is a run-time overhead associated with user-defined functions. If you find things are really slow, it might be time to think about an implementation in the core solver in the D language.
- The other thing that could cause slow execution is sloppy Lua programming. For example, reading/writing files is possible, but slow. Consider storing values in a Lua table.
- Be careful of memory too. Don't try to store very large amounts of data in the memory of the Lua program.
- Basically, you have the full power of a programming language at your disposal: be careful.

User-defined source terms: implementation

- User-supplied source terms **add** to internally computed source terms
- Source terms are a rate of change on a per-volume basis, and can be applied to any/all conservation equations
- User-supplied function (`sourceTerms`) is called *nStages* per time step for every cell in the domain

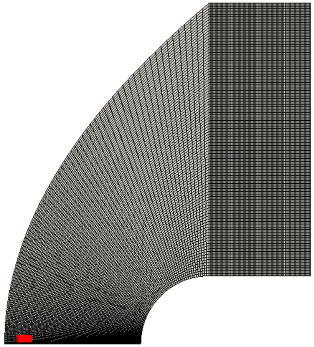
```
for s=1 to n do:
    clear flux data
    apply pre-reconstruction action
    detect shock points
    reconstruct flow data at cell interfaces
    compute convective fluxes
    apply post-convective-flux action
    apply pre-spatial-derivative action
    compute spatial derivatves
    apply post-diffusion flux action
    add source terms, if any
    compute time derivatives of conserved quantities
    update cell-average conserved quantities for stage s
    decode conserved quantities to all flow quantities
```

User-defined source terms: common uses

- To implement manufactured source terms when using the Method of Manufactured Solutions for code verification
- To give rates of species change for specialised chemistry schemes
- To model momentum loss in flow through porous media
- To model heat sources/sinks in the flow field, eg. radiation cooling effect

User-defined source terms: example

Modelling deposition of energy to modify shock location in front of a blunt slab



User-defined source terms: example script

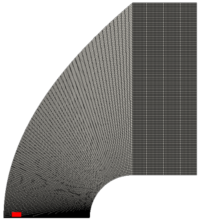
```lua
Q = 2.0e3 -- W
h = 1.0
x_on_h = 0.9
xC = -h/2 - (x_on_h * h)
xL = xC - (0.5*h/15)
xR = xC + (0.5*h/15)
yE = 0.5*h/56

function sourceTerms(t, cell)
   src = {}
   x = cell.x
   y = cell.y
   src.mass = 0.0
   src.momentum_x = 0.0
   src.momentum_y = 0.0
   src.total_energy = 0.0
   if ( y < yE ) then
      if ( x >= xL and x <= xR ) then
         src.total_energy = Q/cell.vol
      end
   end
   return src
end

```

```lua
function specifiedName(args)
    -- do something interesting
    -- perhaps make use of args
    -- then return a value


    return requiredVal
end
```



User-defined source terms: additions to input script

```
config.udf_source_terms = true
config.udf_source_terms_file = 'energy-deposition.lua'
```

User-defined boundary conditions: implementation

- Users can affect both convective and diffusive boundary conditions
- Users decide if they wish to supply ghost cell values, interface values or fluxes directly
- When specifying fluxes or interface values, users decide which values they supply. It is assumed that the values related to other conservation equations are already set.
- When specifying ghost cells, the user must supply all requisite values.

```
for s=1 to n do:
    clear flux data
    apply pre-reconstruction action
    detect shock points
    reconstruct flow data at cell interfaces
    compute convective fluxes
    apply post-convective-flux action
    apply pre-spatial-derivative action
    compute spatial derivatves
    apply post-diffusion flux action
    add source terms, if any
    compute time derivatives of conserved quantities
    update cell-average conserved quantities for stage s
    decode conserved quantities to all flow quantities
```

User-defined boundary conditions: common uses

- To implement time-varying and/or spatially-varying boundary conditions
- To patch inflow data from another simulation or code
- To specify exact conditions on boundaries for Method of Manufactured Solutions
- To implement fluid-structure interaction BCs where the boundary responds to the fluid state

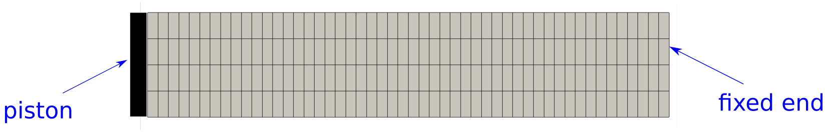
User-defined grid motion: implementation

- We cannot predict all cases of moving grids so we allow the user to control grid motion.
- Grid motion is controlled through movement of vertices.
- The user supplies the velocities of the vertices, not positions.
- Users supply a function `assignVtxVelocities` that needs to give velocities for all vertices in the domain. Vertices with zero velocity may be skipped.
- For moving whole blocks or whole domains at once, there are convenience functions:
  - `setVtxVelocitiesForBlock`
  - `setVtxVelocitiesForDomain`

Example of piston driving shock

This example employs both a user-defined boundary condition and user-defined grid motion.

It is a piston of constant velocity. The piston does not get any feedback from the gas in the cylinder, in this simple case.



User customisations:

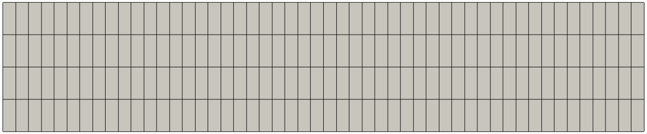
1. The moving piston face BC will be supplied as a user-defined function. The convective fluxes for all of the conservation equations will be given.
2. The grid motion will be defined by setting velocities of grid vertices. Vertices at the piston face have the piston velocity. Vertices at the fixed end have zero velocity. All vertices in between will have a velocity assigned according to a linear distribution.

User-defined BC for moving piston face

```lua
pSpeed = 293.5 -- m/s

function convectiveFlux(args)
   -- We need to get the pressure at the
   -- cell adjacent to the boundary.
   -- We aren't going to do any fancy
   -- reconstruction of the piston face
   -- pressure; we are simply going to take
   -- the adjacent cell pressure.
   cell = sampleFluidCell(blkId, args.i, args.j)
   p = cell.p
   flux = {}
   flux.mass = 0
   flux.momentum_x = p
   flux.momentum_y = 0.0
   flux.total_energy = p*pSpeed
   return flux
end
```

```
for s=1 to n do:
    clear flux data
    apply pre-reconstruction action
    detect shock points
    reconstruct flow data at cell interfaces
    compute convective fluxes
    apply post-convective-flux action
    apply pre-spatial-derivative action
    compute spatial derivatves
    apply post-diffusion flux action
    add source terms, if any
    compute time derivatives of conserved quantities
    update cell-average conserved quantities for stage s
    decode conserved quantities to all flow quantities
```

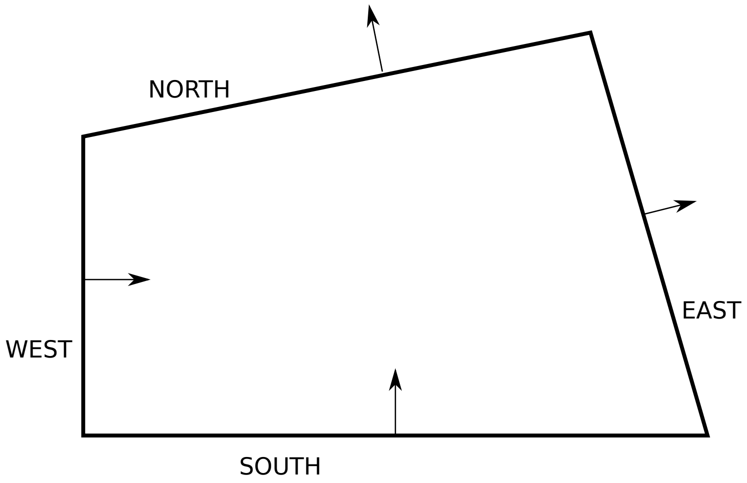


The `convectiveFlux` flux function is called, in turn, at every interface along the user-defined boundary. Here, 4 times.

User-defined boundary conditions: additions to input script

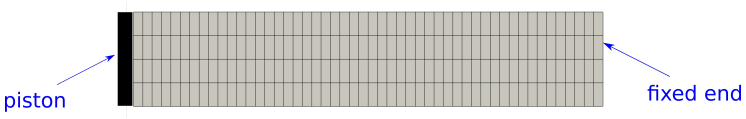
```
blk.bcList[west] = BoundaryCondition:new{type="user_defined",
  ghost_cell_data_available=false,
  convective_flux_computed_in_bc=true,
  postConvFluxAction = {UserDefinedFlux:new{fileName='piston-bc.lua',
                                            funcName='convectiveFlux'}
                       }
}
```

A note on flux directions



These are the positive sense of fluxes for the various boundaries in a 2D structured grid.

User-defined grid motion: example script



```
pSpeed = 293.5 -- m/s
L = 0.5 -- m
H = 0.1 -- m
endDomain = L

function assignVtxVelocities(sim_time, dt)
   -- Compute present position of piston
   pPos = pSpeed * sim_time
   -- Compute current length of domain
   L = endDomain - pPos
   -- Loop over all cells, assigning vertex velocities
   imin = blockData[0].vtxImin
   imax = blockData[0].vtxImax
   jmin = blockData[0].vtxJmin
   jmax = blockData[0].vtxJmax
   for j=jmin,jmax do
      for i=imin,imax do
     -- Find position in duct
     pos = getVtxPosition(0, i, j, 0)
     -- Linearly scale vertex speed based
     -- on how far the vertex is from the
     -- fixed end of the duct.
     vtxSpeed = ((endDomain - pos.x)/L)*pSpeed
     -- Set vertex as Vector3 object
     setVtxVelocity(Vector3:new{x=vtxSpeed}, 0, i, j)
      end
   end
end

```

User-defined grid motion: additions to input script

```
config.gasdynamic_update_scheme = "moving_grid_1_stage"
config.grid_motion = "user_defined"
config.udf_grid_motion_file = "grid-motion.lua"
```

Troubleshooting tips, 1/2

- The user-defined hooks are considered features for advanced users. We do not give much in the way of error messages when things go wrong in your function. Partly because of the effort for such a small user base and partly because of the additional run-time overhead in doing checks on every user-supplied input. Recall how often the functions are called.
- The Lua `print()` function should be your friend for debugging. Remember that the user-defined functions are called <u>many</u> times. It might be helpful to also exit the program after you have some debug information: `os.exit(1)`
- If you want to check that your Lua script is syntactically correct before you start, use the Lua compiler in parse-only mode.

```
> luac -p piston-bc.lua
```

Troubleshooting tips, 2/2

- Check you are playing by the rules:
  - supplying functions with correct signatures and return types
  - not trying to access flow field data unavailable to that process
- Start small and build up complexity as you go
- Design simple tests that run in minutes before launching a multi-day simulation
- RTEM: Read the Eilmer manual (see Appendix in user guide and examples in repository)