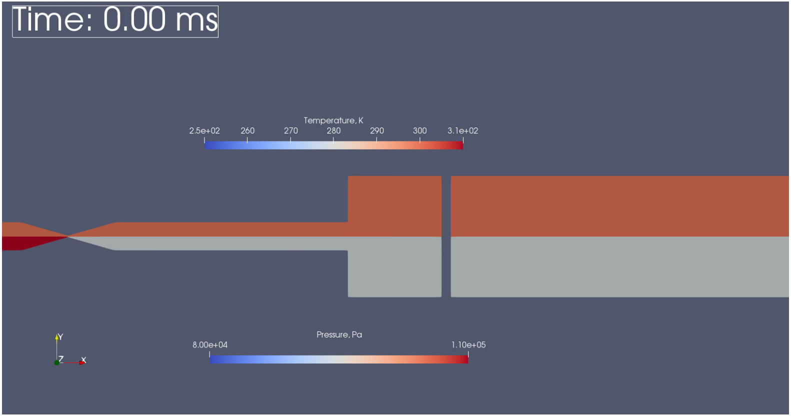
Aerodynamically Driven Moving Mesh Simulations

Fabian Zander, Ingo Jahn, Rowan Gollan,
Peter Jacobs, ... (and more)

Institute for Advanced Engineering and Space Sciences, USQ
fabian.zander@usq.edu.au
(07) 4631 1195

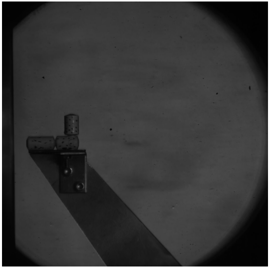
Moving Meshes

- We have been working on moving grids for TUSQ tunnel moving piston (FZ) and moving model (IJ) simulations supported by the dev team (RG, PJ,...)



Shock Surfing Space Debris

- Space debris dispersion is a current hot topic which we are pursuing
- One aspect of dispersion is the shock surfing concept - presented at CfH by Prof. Buttsworth recently
- Hence the flying cube simulation which will be presented here



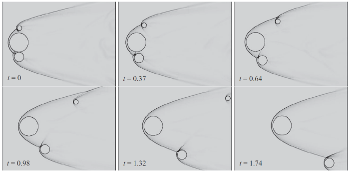


Figure 1: Shock Surfing (Laurence & Deiterding, JFM, Vol 676, 2011).

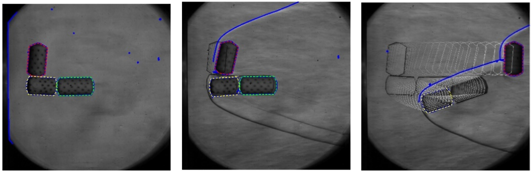


Figure 2: ISS test in TUSQ.

Experimental Work

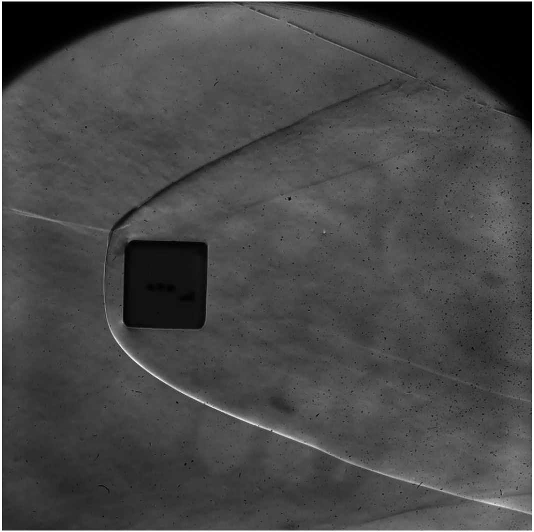


Figure 3: Schlieren of flying cube in TUSQ. Work by Hartmann, Buttsworth, Noller, & Birch.

The Simulation - Different Steps/Files

Main Simulation

- Normal simulation parameters
- Define moving grid usage
- Load calculation parameters
- Specify UDF, UserPad

Load Calcs & Outputs

- Calculate aerodynamics forces
- Save UserPad values
- Output to logfile

Grid Motion

- Specify translation
- Specify rotation

The Simulation - cube.lua

cube.lua

```lua
-- Structured, spherical mesh around a cube.
-- 2021-09-26: PJ, adapted from the sphere example.
-- 2021-10-01: FZ, converted to flying cube with moving mesh

-- Setting up the gas parameters and values
config.dimensions = 3
config.axisymmetric = false
nsp, nmodes, gm = setGasModel('ideal-air-gas-model.lua')
initial = FlowState:new{p=500.0, T=300.0}
inflow = FlowState:new{p=760.0, T=71.0, velx=1005.0}
config.viscous = true

-- Set up the moving grid components
config.gasdynamic_update_scheme = "moving_grid_1_stage"
config.grid_motion = "user_defined"
config.udf_grid_motion_file = "move_grid.lua"

-- Domain is constructed by 6 blocks attached to the outside of the cube.
-- The blocks are labeled in accordance with the face they are attached to.
a_cube = 0.0127
R_mesh = 0.075
vols = {}
faces = {"east", "west", "north", "south", "top", "bottom"}
for _,f in ipairs(faces) do
   cube_face = CubePatch:new{a=a_cube, centre={0,0,0}, face_name=f}
   outer_face = SpherePatch:new{radius=R_mesh, centre={0,0,0}, face_name=f}
   if f == "east" or f == "south" or f == "top" then
      vols[f] = TwoSurfaceVolume:new{face0=cube_face, face1=outer_face, ruled_direction="k"}
   else
      -- west, north, bottom
      vols[f] = TwoSurfaceVolume:new{face0=outer_face, face1=cube_face, ruled_direction="k"}
   end
end

-- Define the 6 grids.
nFactor = 2
N_edge = 20*nFactor  -- cells along edge of cube faces that are mapped onto the sphere
N_normal = 20*nFactor  -- cells in sphere normal direction
grids = {}
for _,f in ipairs(faces) do
   if f == "east" or f == "south" or f == "top" then
      cf = RobertsFunction:new{end0=true, end1=false, beta=1.05}
   else
      cf = RobertsFunction:new{end0=false, end1=true, beta=1.05}
   end
   cfList = {edge04=cf, edge15=cf, edge26=cf, edge37=cf}
   grids[f] = StructuredGrid:new{pvolume=vols[f], cfList=cfList,
                                 niv=N_edge, njv=N_edge, nkv=N_normal}
end
```

The Simulation - cube.lua

cube.lua

cube.lua

```lua
-- and, finally, the FluidBlocks
blks = {}
nib = 3; njb = 3; nkb = 3
wall_bc = WallBC_NoSlip_FixedT:new{Twall=300.0, group='walls'}
inflow_bc = InOutFlowBC_Ambient:new{flowState=inflow}
for _,f in ipairs(faces) do
   if f == "east" or f == "south" or f == "top" then
      bcList = {top=inflow_bc, bottom=wall_bc}
   else
      bcList = {bottom=inflow_bc, top=wall_bc}
   end
   blks[f] = FBArray:new{grid=grids[f], initialState=initial,
                         bcList=bcList, nib=nib, njb=njb, nkb=nkb}
end
identifyBlockConnections()

-- Set up the run-time loads for computing the cube movement
run_time_loads={
   {group="walls", moment_centre=Vector3:new{x= 0.0, y= 0.0, z=0.0}}
      }

-- Setting up the 'standard' simulation parameters
config.max_time = 50.0e-3  -- seconds
config.max_step = 30000000
config.cfl_value = 0.5
config.stringent_cfl = true
config.dt_plot = 0.5e-3

-- These calculations are pretty tough, so allow some bad cells
config.adjust_invalid_cell_data = true
config.max_invalid_cells = 15

-- We need to do some extra work in a controlling file
config.udf_supervisor_file='udf-process.lua'

-- Run time loads are used to calculate the aerodynamic loading
config.compute_run_time_loads = true
config.run_time_loads_count = 1

-- We use a 'user_pad' to store all our information
config.user_pad_length = 8
-- 1=angle, 2=angular_velocity, x, xdot, y, ydot, z, zdot
ang_vel = 7111 * 2 * math.pi / 60 -- 7500rpm
user_pad_data = {0.0, ang_vel, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0}

-- Distribute the blocks for running MPI jobs
mpiTasks = mpiDistributeBlocks{ntasks=100, dist="load-balance"}
```

The Simulation - udf-process.lua

cube.lua | udf-process.lua ./ | move_grid.lua | udf-process.lua CODE

CODE > udf-process.lua

```lua
-- Initialise variables
local x       = 0.
local xdot    = 0.
local xdotdot = 0.
local y       = 0.
local ydot    = 0.
local ydotdot = 0.
local z       = 0.
local zdot    = 0.
local zdotdot = 0.

local upstreamForce = 0.
local downstreamForce = 0.

-- Set a file name for saving data
local outfile_name = 'cube_output.dat'

-- Configure the cube parameters
cubeLength = 25.4e-3
cubeMass = 0.1278
cubeI = cubeMass*cubeLength*cubeLength/6

-- At each time step we need to...
function atTimestepStart(sim_time, steps, delta_t)

  -- Get the variables we need from userPad
  alpha = userPad[1]
  alphadot = userPad[2]
  x = userPad[3]
  xdot = userPad[4]
  y = userPad[5]
  ydot = userPad[6]
  z = userPad[7]
  zdot = userPad[8]

  -- Grab the forces using getRunTimeLoads
  cubeForce, cubeMoment = getRunTimeLoads("walls")

```

The Simulation - udf-process.lua

```

-- Solve momentum equation to get acceleration of the cube in both axes
-- This is just for our data recording, in the simulation, we are only
-- going to rotate the cube
alphadotdot = cubeMoment.z / cubeI
alpha    = alpha + alphadot * delta_t
alphadot = alphadot + alphadotdot * delta_t
xdotdot = cubeForce.x / cubeMass
x       = x + xdot * delta_t
xdot    = xdot + xdotdot * delta_t
ydotdot = (cubeForce.y) / cubeMass
y       = y + ydot * delta_t
ydot    = ydot + ydotdot * delta_t
zdotdot = (cubeForce.z) / cubeMass - 9.81
z       = z + zdot * delta_t
zdot    = zdot + zdotdot * delta_t

-- save data to userPad for vtxSpeed Assignment in grid-motion
userPad[1] = alpha
userPad[2] = alphadot
userPad[3] = x
userPad[4] = xdot
userPad[5] = y
userPad[6] = ydot
userPad[7] = z
userPad[8] = zdot

```

The Simulation - udf-process.lua

cube.lua   udf-process.lua

udf-process.lua

```lua

  -- The rest is 'just' for outputting and saving data
  if in_mpi_context and not is_master_task then
  else
    -- print progress to screen
    if (steps % 500) == 0 then
      print('++++++++++++++++')
      print('Steps: ', steps, '   sim-time (sec): ',sim_time)
      print(string.format('alpha %.4f (rad) alphadot %.4f (rad/s)', alpha, alphadot) )
      print(string.format('x %.4f (m) xdot %.4f (m/s)', x, xdot) )
      print(string.format('y %.4f (m) ydot %.4f (m/s)', y, ydot) )
      print(string.format('z %.4f (m) zdot %.4f (m/s)', z, zdot) )
    end

    -- write data to file
    if steps == 0 then
      -- check if file exists
      local infile=io.open(outfile_name,"r")
      if infile~=nil then -- yes file already exists
        instr = infile:read("*a")
        infile:close()
        -- write contents to a new file
        outfile = io.open(outfile_name .. ".old", "w")
        outfile:write(instr)
        outfile:close()
        print('Output file=' .. outfile_name .. ' already exists. File copied to ' .. outfile_name .. '.old')
        os.remove(outfile_name) -- now delete file
      end

      file=io.open(outfile_name,"a")
      file:write('# pos1:sim_time(s) pos2:Alpha(rad) pos3:AlphaDot(rad/s) pos4:PositionX(m) pos5:VelocityX(m/s) pos6:PositionY(m) pos7:VelocityY(m/s) pos8:PositionZ(m)
      file:close()
    end
    if (steps % 500) == 0 then
      file=io.open(outfile_name,"a")
      file:write(string.format('%.18e %.18e %.18e %.18e %.18e %.18e %.18e %.18e %.18e\n', sim_time, alpha, alphadot, x, xdot, y, ydot, z, zdot) )
      file:close()
    end
  end

  return nil
end

```

The Simulation - udf-process.lua

```lua
-- The is for saving our userpad data when we write flow data so that we can do a restart
function atWriteToFile(sim_time, steps)
  -- We want to save our userPad data every time we write out a flow solution
  -- Only let the master task do this
  if in_mpi_context and not is_master_task then return end
  -- I want to save my userPad for restarts
  assert( table.save(userPad, "userPadSave.lua") == nil )
  -- I also want to save my cube data at these points to allow an
  -- arbitrary restart
  file=io.open("cube_userpad_data.dat", "a")
  file:write(string.format('%.18e %.18e %.18e %.18e %.18e %.18e %.18e %.18e %.18e\n',
    sim_time, userPad[1], userPad[2], userPad[3], userPad[4], userPad[5], userPad[6], userPad[7], userPad[8]) )
  file:close()
end

```

The Simulation - move-grid.lua

cube.lua | udf-process.lua | move_grid.lua

move_grid.lua

```lua
-- Cube flying mesh movement for TUSQ simulation
--
-- Author: Fabian Zander
-- Last Modified: 26/08/2021

function assignVtxVelocities(sim_time, dt)
  -- I'm doing a dodgy on the movement here - one us is translation,
  -- the next us is rotation. Hence double the values for each us.

  -- I need a time which can use the modulo function
  tmpTime = math.floor(sim_time*1e6)

  if (tmpTime % 2 == 0) then
    -- First we translate
    xdot = userPad[4]*2 -- Multiply by 2 for every other us
    ydot = userPad[6]*2 -- Multiply by 2 for every other us
    zdot = userPad[8]*2
    -- Noting the use of the Domain movement here
    setVtxVelocitiesForDomain(Vector3:new{x=xdot, y=ydot, z=zdot})

  else
    -- Then we rotate
    alphadot = userPad[2]*2 -- Multiply by 2 for every other us
    x = userPad[3]
    y = userPad[5]
    z = userPad[7]
    for _,blkId in ipairs(localFluidBlockIds) do
      -- This time we rotate each block around the current 'centre' location
      setVtxVelocitiesForRotatingBlock(blkId, alphadot, Vector3:new{x=x, y=y, z=z})
    end
  end
end
```

The Eilmer Result

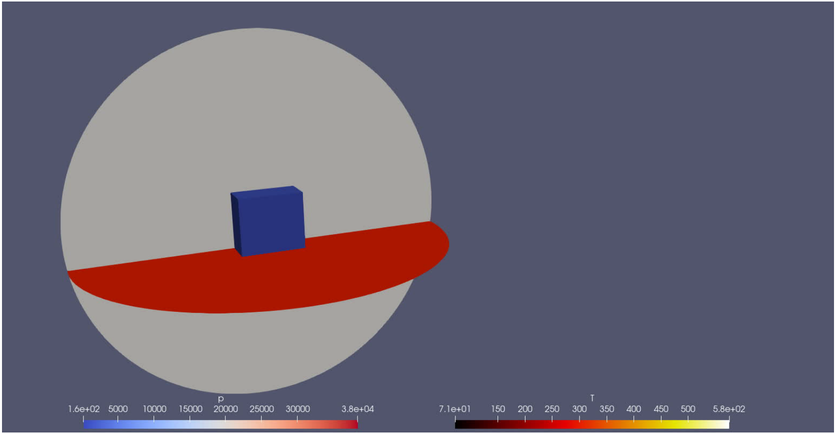
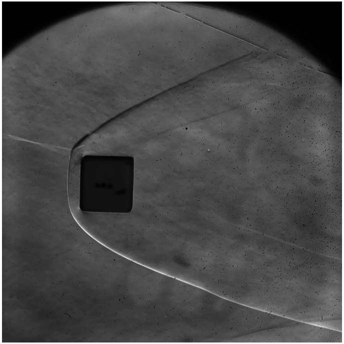
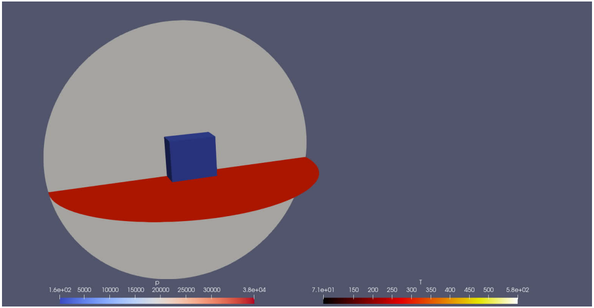


Figure 4: Eilmer CFD result of the flying cube. Numerical schlieren, temperature and surface pressure

Visual Comparison (as far as possible)





Note - different frame rates in the videos

Next Steps

- Get funding (!)
- Higher resolution
- Turbulence
- Multi-body (how?)
- Radiation coupling
- Modelling of video below (Rowan!?)