GeMA – Temperature tutorial examples

08/10/2018 – Version 1.1

Tecgraf
PUC-RIO

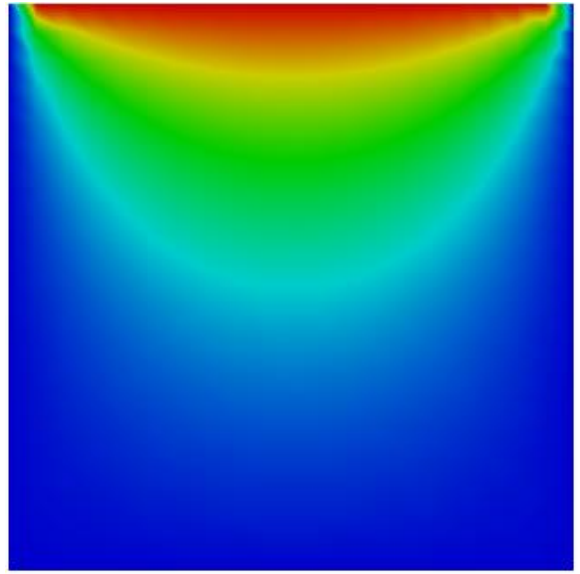
Example set purpose

This examples show:

- How to setup models for both steady state and transient temperature analysis
- Several different orchestration techniques, useful in many contexts
- How to setup mesh attributes / cell properties with values calculated by user functions
- How to initialize values for element integration points
- How to setup time dependent boundary conditions

The examples

- Examples 1 and 2 presents a steady state and a transient analysis of heat conduction on a square plate subjected to Dirichlet (prescribed temperature) boundary conditions.
  - This examples are similar to the ones presented on the GeMA tutorial. Their main change is a simplified way of building the mesh.



- Example 3 presents a transient analysis of a rod, heated on one side and insulated on the others.

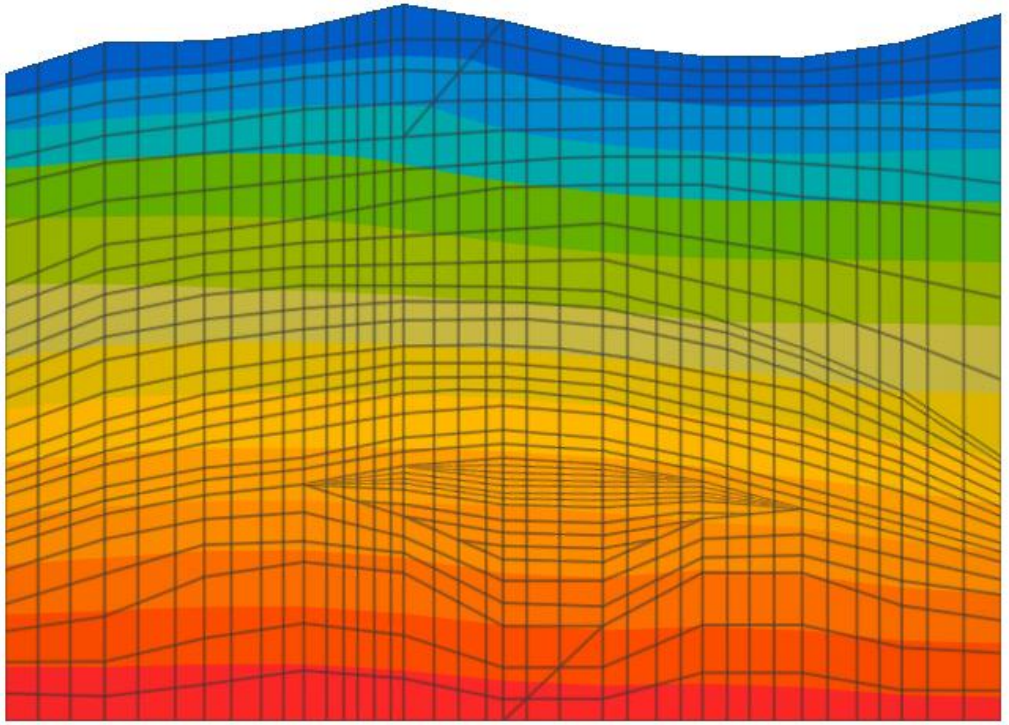
The examples

- Example 4 presents the steady state temperature profile for a set of sedimentary layers subjected to a heat flux at their base and a prescribed surface temperature.

- Example 5 builds on the previous example by considering porosity and nonlinearities arising from making the conductivity a function of the temperature.

- Example 6 presents a transient analysis for a scenario where the surface temperature is a function of the bathymetry and changes over time.

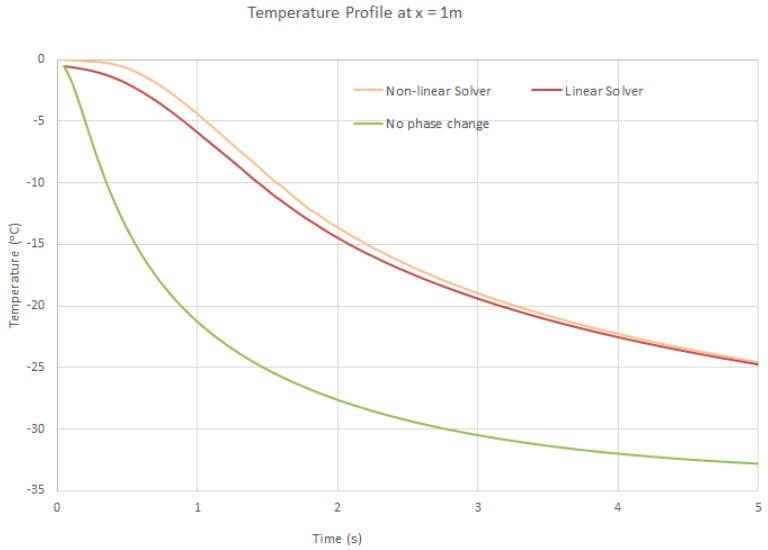
- Example 7 updates the porosity initialization function to traverse mesh cells in parallel.



Together, this 4 examples show how some complex models can be created by combining user provided functions with advanced orchestration techniques.

The examples

- Example 8 presents a phase change analysis for a solidification problem, solved by using the effective heat capacity method and the non-linear FEM solver.



On this presentation, the first example will present and analyze the complete model source. Other examples will present only relevant parts. The complete source for all models are available at the example files.

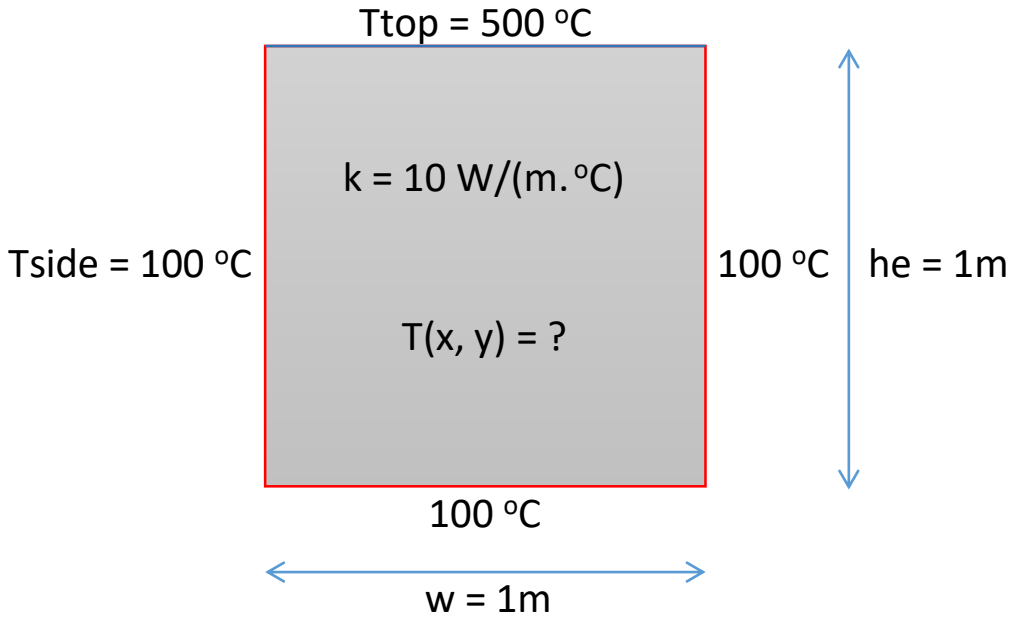
Although this examples try to explain all involved concepts and syntaxes, they are not a substitute for reading the GeMA tutorial and additional documentation.

As a convention, all the given examples will generate result files on the “out” directory.

1 – STEADY STATE HEAT CONDUCTION ON A SQUARE PLATE

The Problem

- Heat conduction on a square plate subjected to prescribed temperatures on its borders:



- Analytical solution used to validate results:

$$T(x,y) = T_{side} + (T_{top} - T_{side})\frac{2}{\pi}\sum_{n=1}^{\infty}\frac{(-1)^{n+1}+1}{n}\sin\left(\frac{n\pi x}{w}\right)\frac{\sinh\left(\frac{n\pi y}{w}\right)}{\sinh\left(\frac{n\pi h}{w}\right)}$$

Reference: "Fundamentals of the Finite Element Method for Heat and Fluid Flow", Lewis at al.

Example 5.2.1

Key Example Points

- This example presents the basic structure for a GeMA simulation

- In particular it shows how to:
  - Structure a GeMA model
  - Create a simple, “grid like”, mesh using auxiliary functions
  - Setup material values
  - Setup a Dirichlet boundary condition by prescribing border temperatures
  - Work with units
  - Create user functions to validate the results against analytical results
  - Setup a simple orchestration for solving the problem through the Finite Element Method
  - Save and print results
    - Files are saved both in Neutral file and Vtk formats

Simulation file: SteadyStateHeatConduction.lua

Simulation file: SteadyStateHeatConduction.lua

This file is the main simulation file. It stores a model description and loads the two auxiliary files storing the **simulation model** and the **simulation solution**. Splitting the simulation on those files is a convention to separate the description of **what** will be simulated (the model file) from the description of **how** it will be simulated (the solution file).

```
-- The model description
Simulation
{
  name         = 'Steady state conduction',
  modelVersion = '1.0',
  description  = 'A simple heat conduction model for a square plate with different\n'..
                 'prescribed border temperatures.\n'..
                 'Reference: "Fundamentals of the Finite Element Method for Heat and Fluid Flow"\n'..
                 '           Lewis at al, example 5.2.1'
}

-- Load model and solution files for this problem
dofile('$SIMULATIONDIR/$SIMULATIONNAME_model.lua')
dofile('$SIMULATIONDIR/$SIMULATIONNAME_solution.lua')
```

Comments begin with `--` and run until the end of the line

String concatenation operation used for creating a multi-line description

Loads the file “SteadyStateHeatConduction_solution.lua” from the same directory as the simulation file

Macro expanding to the directory of this simulation

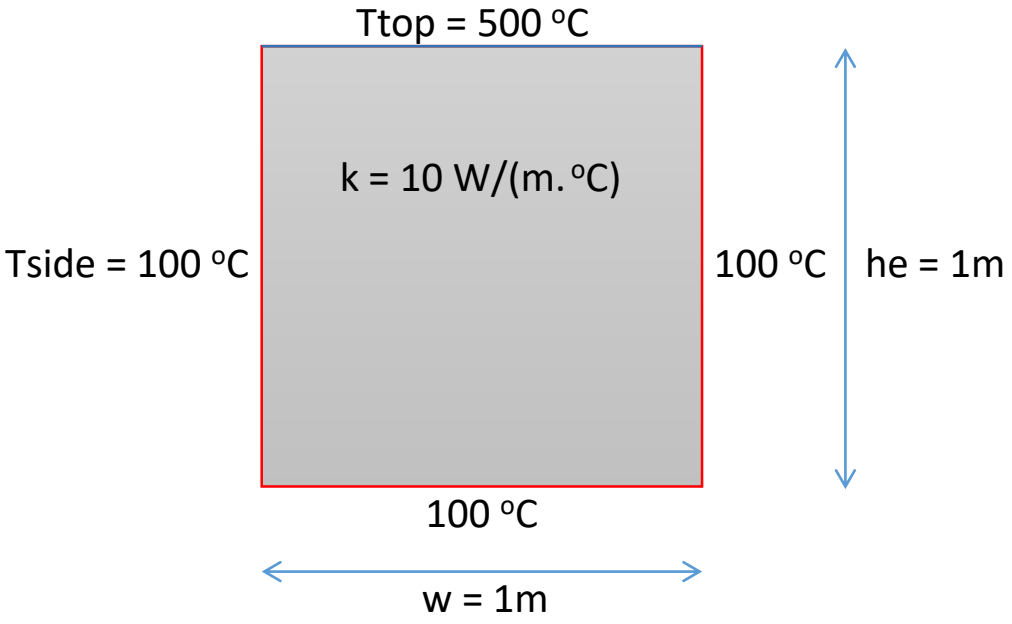
Macro expanding to the name of this simulation file, without extension

Model file: Model parameters

The model file begins by creating some constants with model parameters in order to simplify changing the plate geometry, conductivity and prescribed temperatures.

```
local cond =  10   -- Plate conductivity in W/(m.degC)
local Tt   = 500   -- Top temperature in degC
local Ts   = 100   -- Bottom and side temperatures in degC

local nw   = 20    -- The number of plate cells in the x direction
local nh   = 20    -- The number of plate cells in the y direction

local w    = 100   -- Plate width in cm
local he   = 100   -- Plate height in cm
```

Although the Thermo physics operates with SI units, this example will use a mesh with coordinates in cm and temperatures in degC. The system will perform the need conversions automatically.



Model file: State variable

Next, the simulation state variable is defined. State variables are the nodal values calculated by the simulation and represent the model degrees of freedom. For a thermal simulation, the state variable should be a scalar value named **T**, as expected by the Thermo physics plugin.

```
-- State variables
StateVar{id = 'T', description = 'Temperature', unit = 'degC', format = '8.2f'}
```

The state variable name

The unit in which the temperature data will be stored in the mesh

The format used when printing temperature values (width + precision). Uses the same syntax as the C printf() function

Model file: Material properties

Material properties are given by property sets. A property set is a table with material properties as columns. Each cell references one line for each property set in the model. For this simulation, only one table is needed with columns for the material conductivity and the plate thickness. This columns should be named **k** and **h**, as expected by the Thermo physics.

```
-- Cell properties
PropertySet
{
  id          = 'MatProp',
  typeName    = 'GemaPropertySet',
  description = 'Material parameters',

  properties  = {
    {id = 'k', description = 'Conductivity',      unit = 'W/(m.degC)'},
    {id = 'h', description = 'Element thickness', unit = 'cm'},
  },

  values = {
    {k = cond, h = 1.0},
  }
}
```

The property set name

The unit in which property values will be given. Although k is given with a length unit of 'm' and t as 'cm', there is no problem since automatic conversion will make them compatible.

Table column definitions

The property (table column) name

Table line. If the model had more materials, the values table would have several lines

Property name

Constant cond defined on the beginning of the file

Model file: Analytical solution

Before creating the mesh, a node user function is created for calculating the error between the analytical and the numerical solutions. It will be used to provide values for a node attribute.

```
-- A function to calculate the analytical model result at the given
-- coordinate. Expects x and y to be given in cm.
local function anaT(x, y)
  local sum = 0
  for i = 1, 200 do
    local u   = i * math.pi/w
    local num = ((-1)^(i+1) + 1) * math.sin(u*x) * math.sinh(u*y)
    local den = i * math.sinh(u*he)
    sum = sum + num/den
  end
  return Ts + (Tt-Ts) * (2/math.pi) * sum
end
```

$$T(x,y) = T_{side} + (T_{top} - T_{side})\frac{2}{\pi}\sum_{n=1}^{\infty}\frac{(-1)^{n+1}+1}{n}\sin\left(\frac{n\pi x}{w}\right)\frac{\sinh\left(\frac{n\pi y}{w}\right)}{\sinh\left(\frac{n\pi h}{w}\right)}$$

```
-- A node user function for calculating the error between
-- the analytical and the numerical solutions
NodeFunction { id = 'errf',
               parameters = { {src = 'coordinate', unit = 'cm'},
                              {src = 'T',          unit = 'degC'},
                            },
               method = function(nodePos, nodeT)
                 return math.abs(anaT(nodePos[1], nodePos[2]) - nodeT)
               end
             }
```

The desired parameter unit. Values are converted if needed

The node coordinate

The node temperature from state var T

x and y coordinates

The match between ‘coordinate’ and ‘nodePos’, ‘T’ and ‘nodeT’ is done by their positions in the parameters list and in the function arguments

Model file: Mesh

Since the simulation domain is a square plate, the discretization mesh can be easily created by using auxiliary functions. Refer to the “meshLib” examples for further details and options.

```
dofile('$SCRIPTS/meshLib.lua') -- Loads the 'meshLib' auxiliary functions

local xpoints = meshLib.regularSpacing(0.0, w, nw)
local ypoints = meshLib.regularSpacing(0.0, h, nh)
```

Creates the set of x coordinates used to define the position of grid columns. This call creates a regularly spaced table of coordinates from 0.0 to the plate width (given by w) with nw+1 points (nw cells).

```
local nodes, elements, borders = meshLib.build2DGrid('quad4', xpoints, ypoints, nil, {MatProp = 1})
```

Returned tables storing the mesh geometry

Element type

Property set name

Default property set line associated with mesh elements

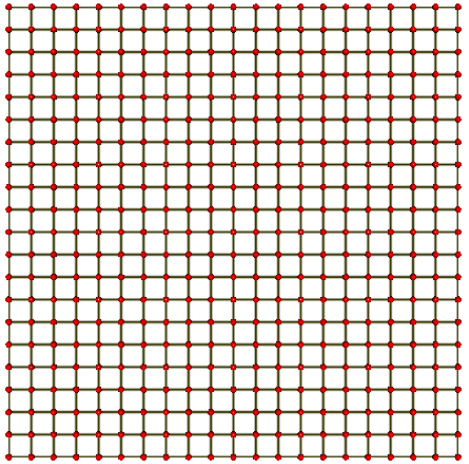
```
Mesh
{
  -- General mesh attributes
  id          = 'mesh',
  typeName    = 'GemaMesh.elem',
  description = 'Plate mesh discretization',

  -- Mesh dimensions
  coordinateDim  = 2,
  coordinateUnit = 'cm',
```

The mesh name

The plugin name used to create the mesh (GemaMesh) + the mesh type (elem)

The mesh dimension (2D)

The unit in which node coordinates are given



… continues on the next slide

Model file: Mesh (continued)

… continued from previous slide

```
  -- State vars stored in this mesh (per node)
  stateVars = {'T'},

  -- Node attributes
  nodeAttributes = {
    {id = 'Err',  description = 'Error between expected and calculated values', unit = 'degC',
                  functions = true, defVal = 'errf', format = '10.4'}
  },

  -- Mesh node coordinates
  nodeData = nodes,

  -- Element data
  cellProperties = {'MatProp'},
  cellData       = elements,

  -- Boundary data
  boundaryEdgeData = borders,
}
```

Associates this mesh with state variable T

List with user created node attributes.

Node attribute name

Err supports functions and its default value is the errf node function.

Sets the table with node coordinates

Associates this mesh with property set MatProp

Sets the table with element definitions

Sets the table with mesh border definitions

Model file: Boundary conditions

To complete the model file, boundary conditions for prescribing temperatures on plate borders are needed. For that, the Thermo physics provides the ‘node temperature’ boundary condition type where temperature values are associated with mesh nodes. Like property sets, boundary conditions are like a table with several columns. On this case, only one column for the temperature, named **T**, is needed.

```
BoundaryCondition {
  id   = 'Border temperature',
  type = 'node temperature',
  mesh = 'mesh',

  properties  = {
    {id = 'T',  description = 'External temperature applied on the node', unit = 'degC'},
  },

  nodeValues = {
    {'gridLeft',   Ts},
    {'gridBottom', Ts},
    {'gridRight',  Ts},
    {'gridTop',    Tt},
  }
}
```

Boundary condition name

Boundary condition type for prescribed temperature values

Associates this boundary condition with the mesh named ‘mesh’

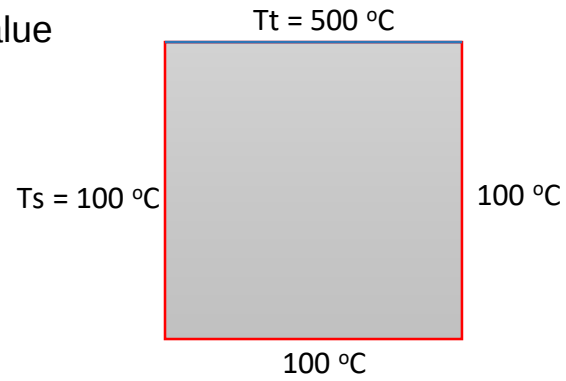
B.C. column definitions

B.C. column name

Boundary conditions. Each table line associates a node set to a temperature value

Temperature constants defined on the beginning of the file

Node set names automatically created by the “meshLib”, storing border edges



Solution file: Numerical solver

The first section of the solution file defines which numerical solver will be used to solve the equation system created by the FEM method. On this example, we will use a direct matrix solver provided by the ArmadilloSolver plugin.

```
NumericalSolver {
  id          = 'solver',
  typeName    = 'ArmadilloSolver',
  description = 'Direct matrix solver',
}
```

The solver name

The plugin name used to create the numerical solver

Solution file: Physics

Physics are the objects that provide the set of mathematical equations used to solve the simulation. For solving a temperature problem, this example will use the Thermo physics plugin.

```
PhysicalMethod {
  id         = 'HeatPhysics',
  typeName = 'ThermoFemPhysics',
  type       = 'fem',

  mesh                 = 'mesh',
  boundaryConditions = {'Border temperature'},
}
```

The physics name

The plugin name used to create the physics

Associates this physics with the mesh named ‘mesh’

Associates this physics with a set of boundary conditions

Solution file: Orchestration script

Finally, the orchestration script, provided by the ProcessScript() Lua function, drives the simulation by calling the FEM process to execute the simulation.

```
function ProcessScript()
  -- Prints model mesh information before running the simulation
  io.printMesh('mesh')

  -- Solves the model using the FEM method for linear problems
  fem.solve({'HeatPhysics'}, 'solver')
  -- Print results evaluating the error between the analytical and numerical solutions
  print('\n\nCalculated results:')
  io.printMeshNodeData('mesh', {'coordinate', 'T', 'Err'}, {eval_functions=true})
  -- Save results on the out sub-directory of the same directory hosting the simulation model
  -- Saved file will have the same name as the simulation file
  -- The saveMeshFile() call expects the 'out' sub-directory to exist
  io.saveMeshFile('mesh', '$SIMULATIONDIR/out/$SIMULATIONNAME.nf', 'nf', {'T', 'Err'})
  -- Save result using Vtk format
  dofile('$SCRIPTS/vtkLib.lua')
  vtkLib.saveMeshFile('mesh', {'T', 'Err'}, nil, '$SIMULATIONDIR/out/$SIMULATIONNAME')
end
```

The set of physics used to solve the problem

The numerical solver used to solve the equation system

Printed nodal values are the node coordinates, the state variable T and the node attribute Err

Functions should be evaluated since we want to print the error value, and not the function name associated with Err

Expanded macros

Output file type (Neutral File)

Saved nodal values are the state variable T and the node attribute Err

Load Vtk lib.

Create Paraview (Vtk) type output

Saved node values

No saved Gauss points values

Output file

Running the simulation

- At the command prompt type:

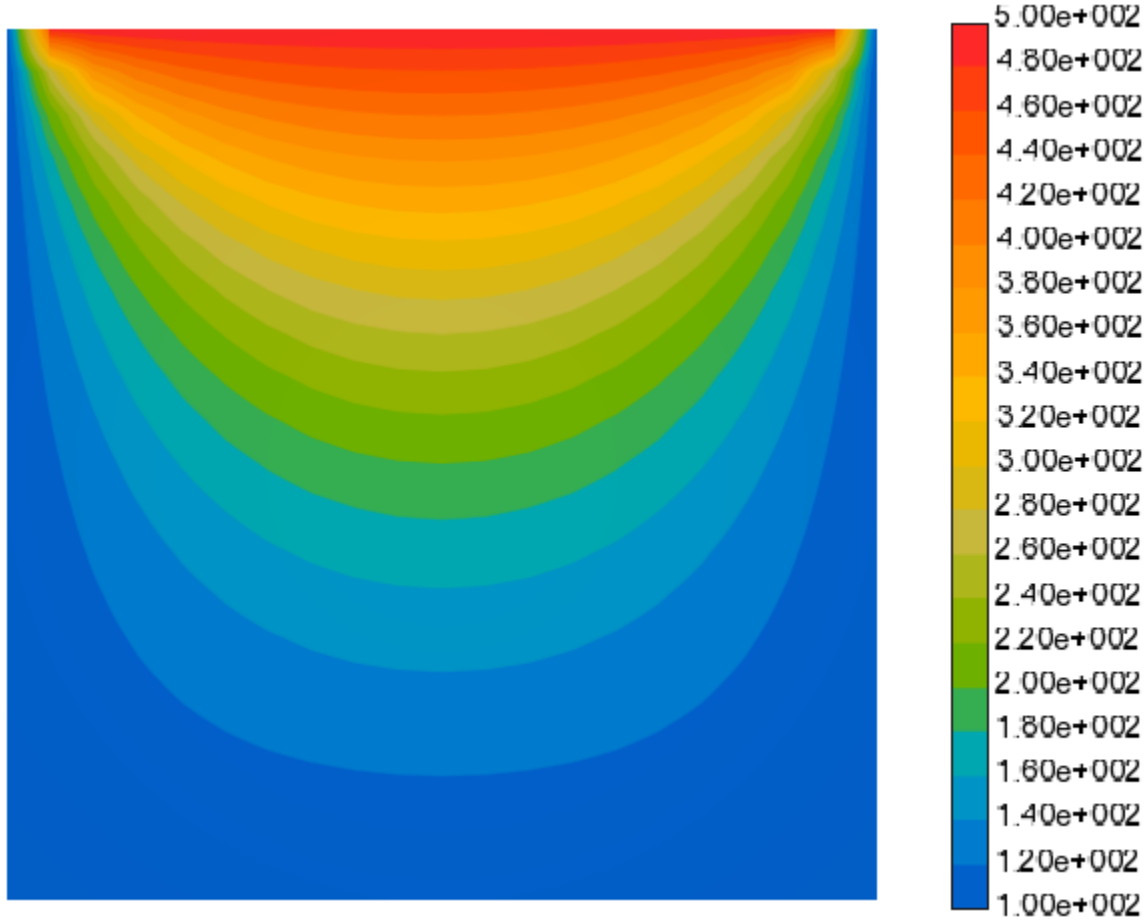
```
C:\> gema SteadyStateHeatConduction.lua
```

- After running:
  - The initial mesh will be printed to the console, followed by simulation results
  - The same contents printed to the console can be found on the ‘runlog.txt’ file, created on the current directory
  - Simulation results will also be saved, in neutral file format, to the file SteadyStateHeatConduction.nf on the “out” sub-directory at the same location of the example file. Please make sure that the “out” directory exists and you have write permission on it before running the simulation.
  - Results will also be saved in Vtk format (files with .pvd and .vtu extensions)

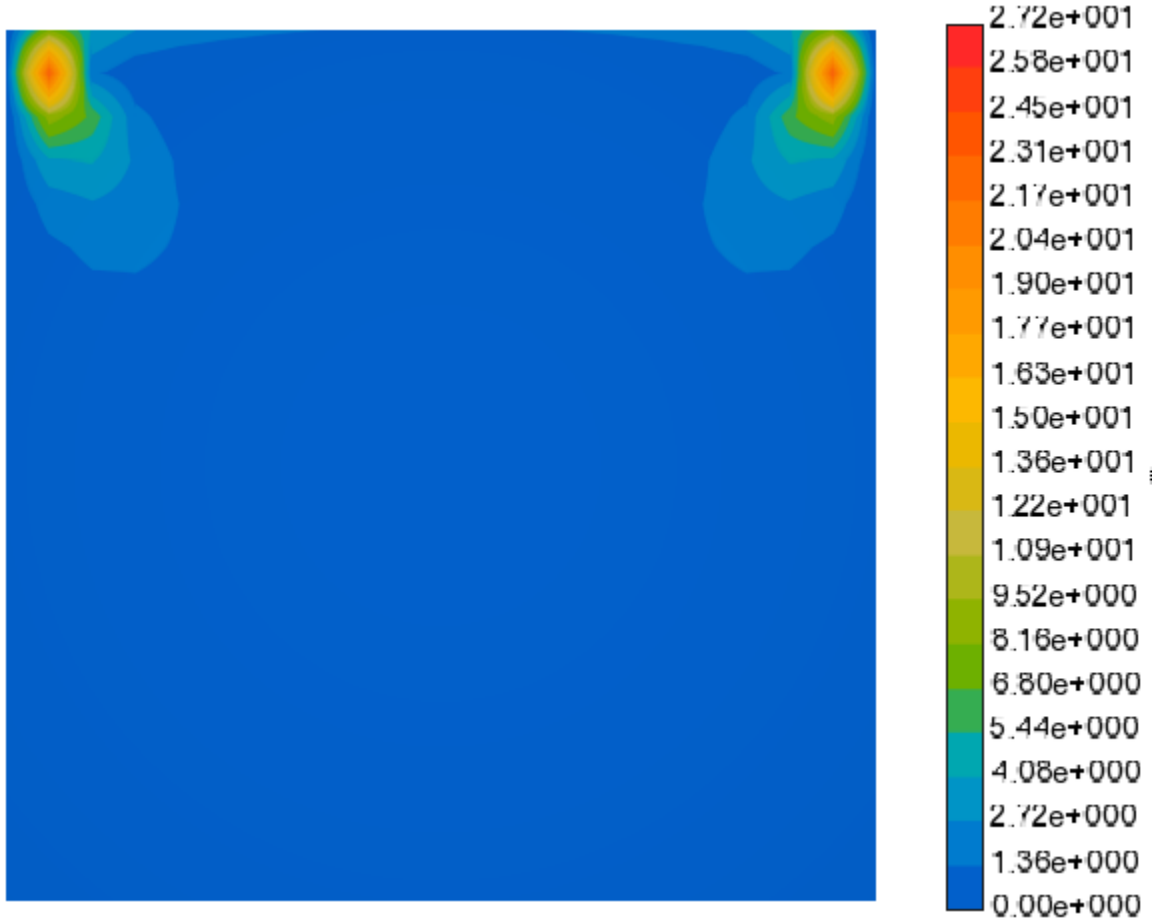
Analyzing the runlog.txt file, the careful reader will find a set of warnings associated to the boundary conditions. This happens because, in our model, top corner nodes have both a 500 degC and a 100 degC prescribed temperature. The inconsistency is solved by assuming the first prescribed temperature for the node.

Results (Neutral file - Pos3D)

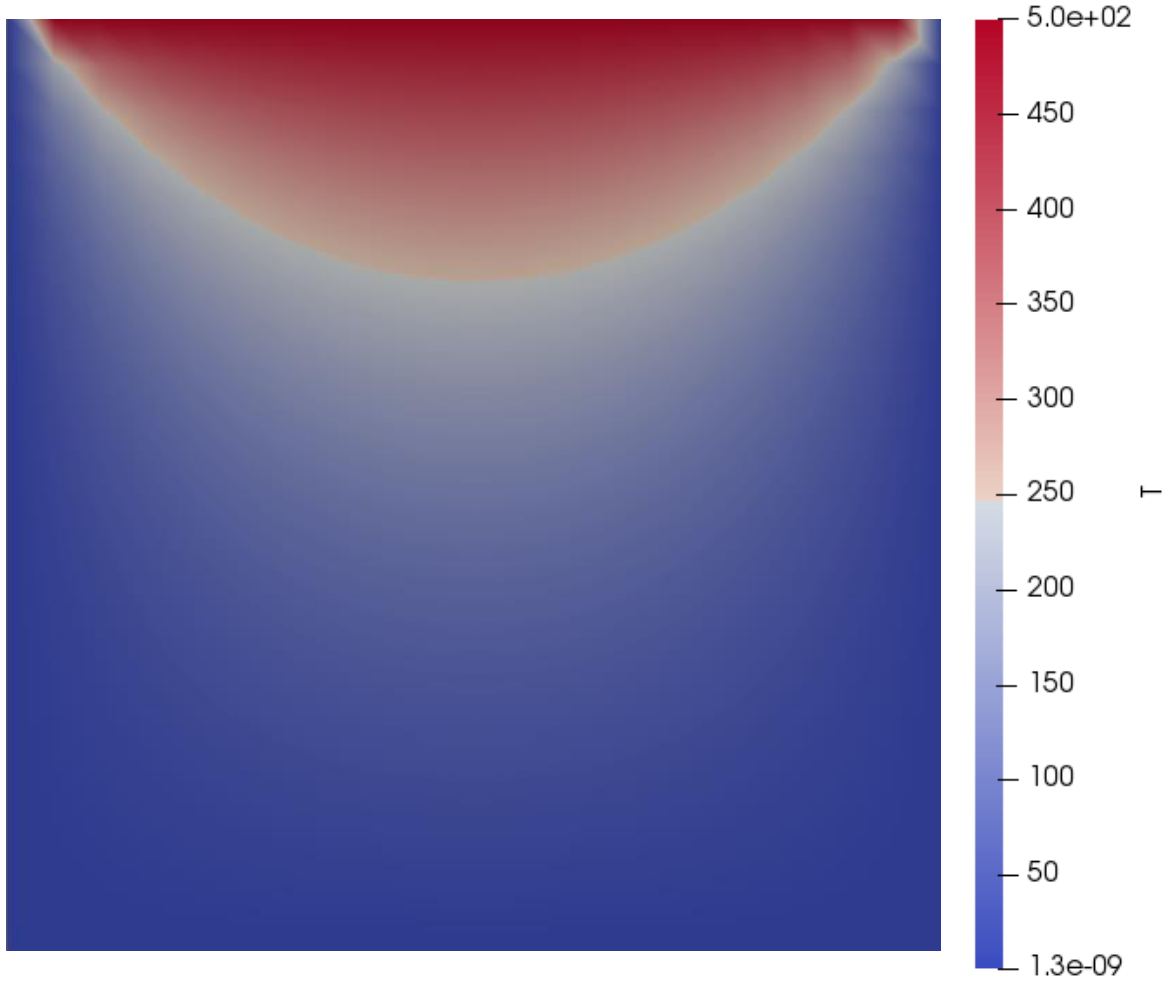
Temperature

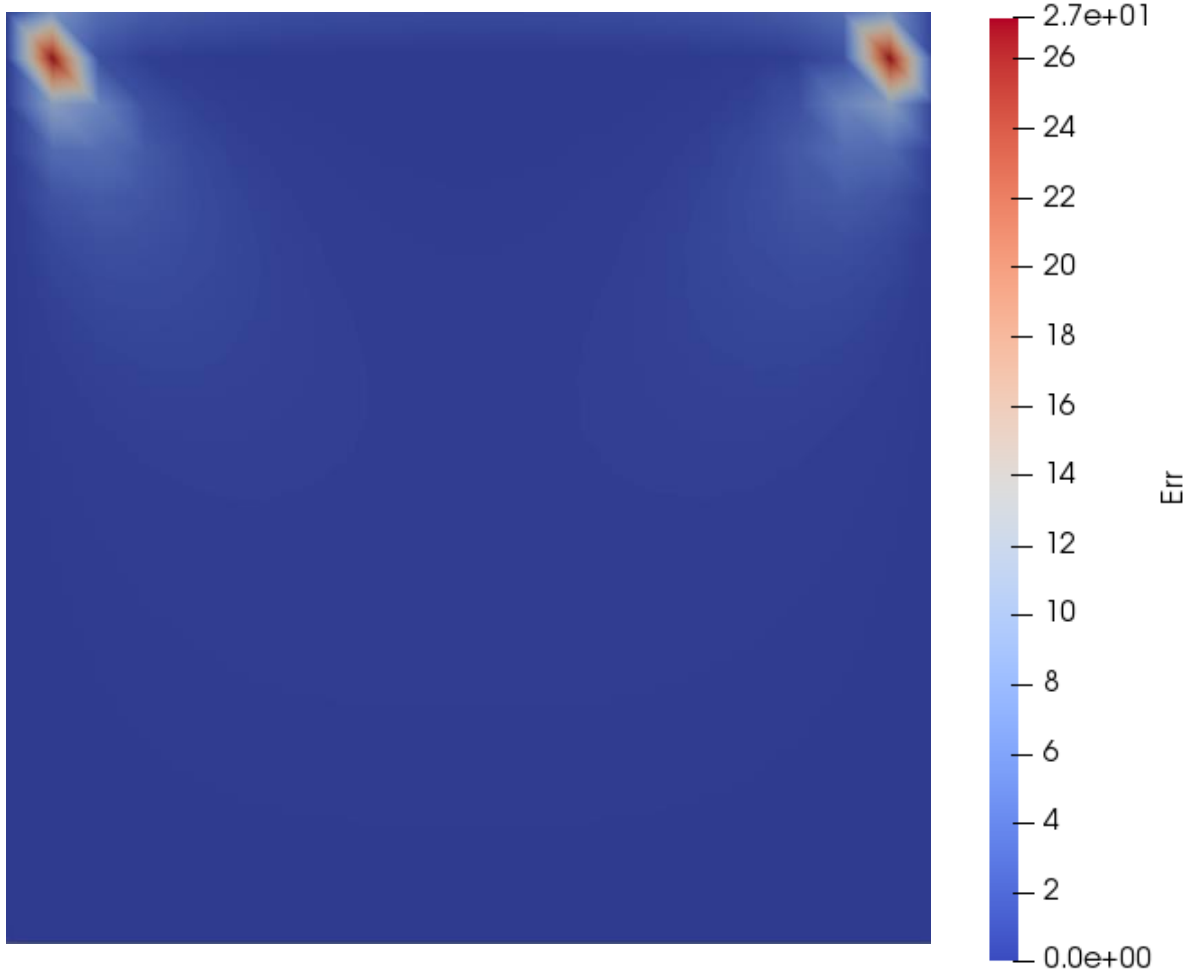


Error



Results (Vtk - Paraview)

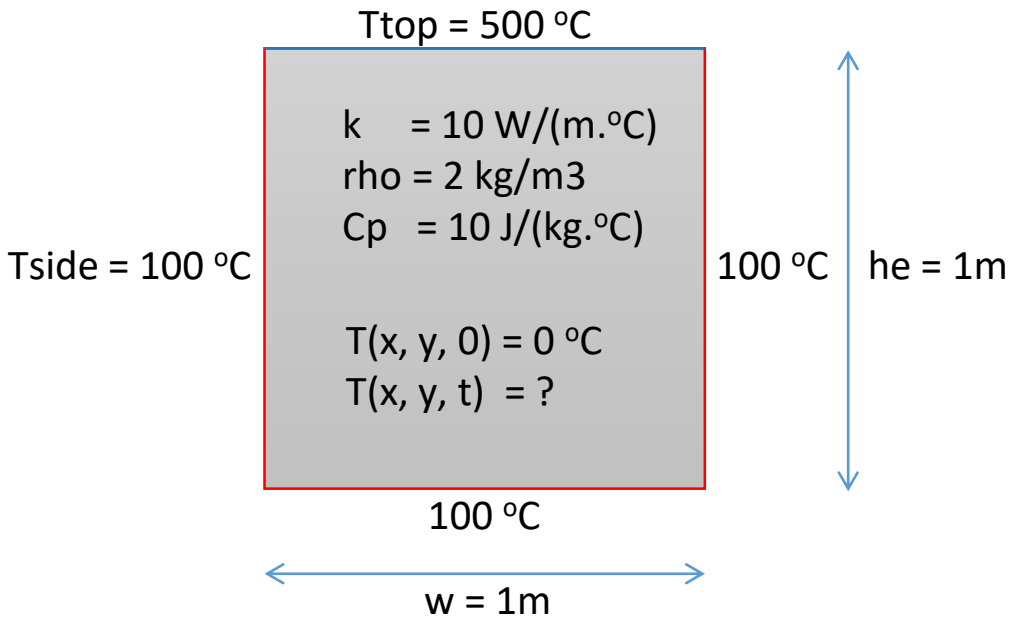




2 – TRANSIENT HEAT CONDUCTION ON A SQUARE PLATE

The Problem

- This simulation revisits the previous example, analyzing the transient problem when the same boundary conditions are suddenly imposed over a plate with an initial uniform temperature equal to 0 ºC



Reference: "Fundamentals of the Finite Element Method for Heat and Fluid Flow", Lewis at al.

Example 6.6.1

Key Example Points

- This example shows how to create a simple orchestration for transient linear problems

- This example builds heavily on the previous one and only shows key difference points. Please refer to the simulation files for the complete source.

Simulation file: TransientHeatConduction.lua

Model file

The main changes to the model file are the initialization of the plate temperature, the addition of the density and specific heat capacity to the property set and the removal of the analytical solution.

```
local cond =  10  -- Plate conductivity in W/(m.degC)
local dens =   2  -- Plate density in kg/m3
local hcap =  10  -- Plate specific heat capacity in J/(kg.degC)
local T0   =   0  -- Initial plate temperature in degC

StateVar{id = 'T', description = 'Temperature', unit = 'degC', defVal = T0, format='8.2f'}

PropertySet
{
  id          = 'MatProp',
  typeName    = 'GemaPropertySet',
  description = 'Material parameters',
  properties  = {
    {id = 'k',   description = 'Conductivity',      unit = 'W/(m.degC)'},
    {id = 'rho', description = 'Density',           unit = 'kg/m3'},
    {id = 'cp',  description = 'Specific heat',     unit = 'J/(kg.degC)'},
    {id = 'h',   description = 'Element thickness', unit = 'cm'},
  },
  values = {
    {k = cond, rho = dens, cp = hcap, h = 1.0},
  }
}
```

The default value used to initialize the state variable value at each node. When this value is missing, the state var is initialized with 0.0

Solution File

The orchestration script now includes the time loop, with calls at each time step to the FEM solver to advance the simulation. It also builds the result file iteratively.

```
function ProcessScript()
  -- Creates the FEM solver used for calculating the numerical solution at each time step
  -- for linear problems
  local solver = fem.initTransientSolver({'HeatPhysics'}, 'solver')
  local dt     = 0.001     -- The time step
  local endt   = 0.2       -- Simulation duration
  local nsteps = endt / dt -- Number of steps

  -- Creates the output simulation file
  local file = io.prepareMeshFile('mesh', '$SIMULATIONDIR/out/$SIMULATIONNAME.nf', 'nf', {'T'})
  -- Load VtkLib to enable saving results in the Vtk format
  dofile('$SCRIPTS/vtkLib.lua')

  -- Simulation time loop
  for i=1, nsteps do
    -- Set the current time in the model
    setCurrentTime(i*dt)

    -- Solve one time step
    fem.transientStep(solver, dt)

    -- Add step results to the ouptut file using neutral file and vtk formats
    vtkLib.saveMeshFile('mesh', {'T'}, nil, '$SIMULATIONDIR/out/$SIMULATIONNAME', {state = i, stateTime = i * dt})
    io.addResultToMeshFile(file, i * dt)
  end
  -- Close output file
  io.closeMeshFile(file)
end
```

The solver object returned by fem.initTransientSolver()

Advances the simulation by a time step dt.

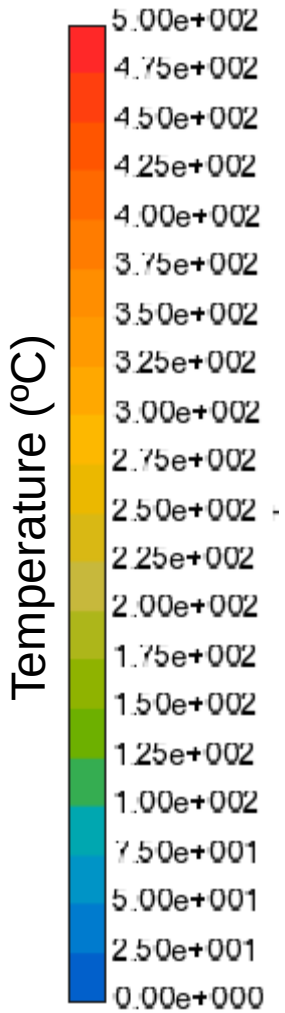
Current time

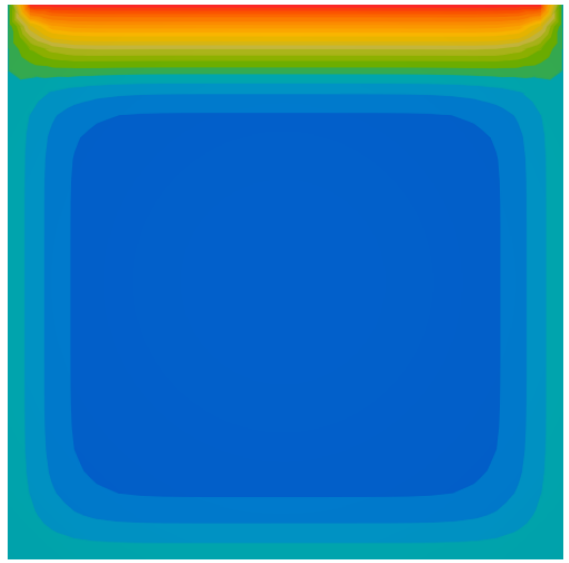
Time step

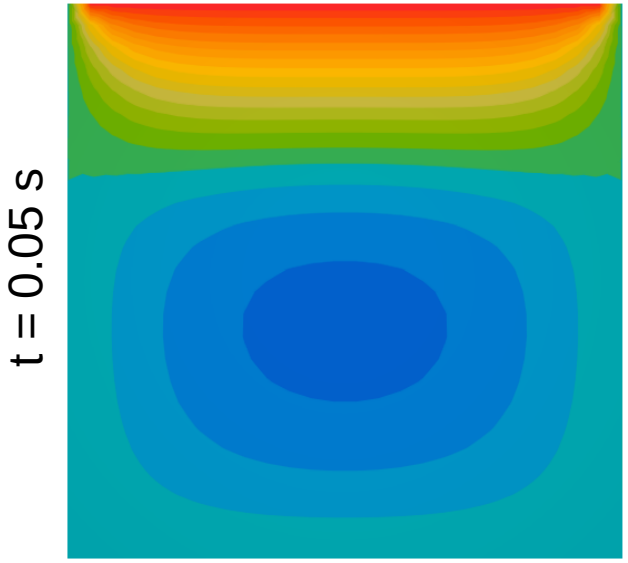
Current time

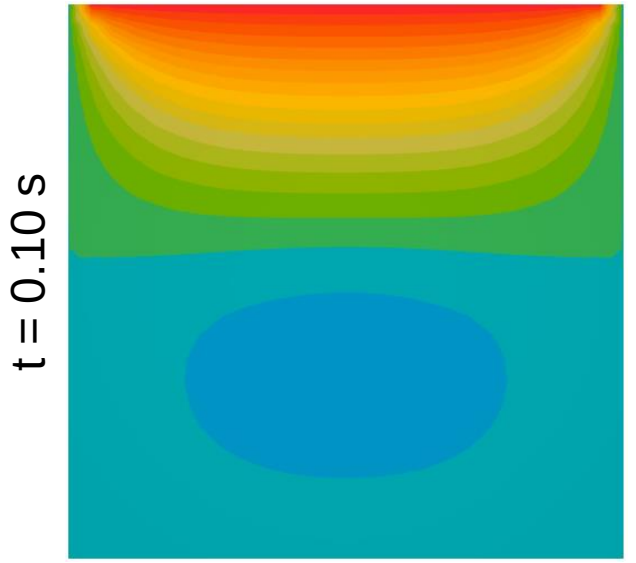
The file object returned by io.prepareMeshFile()

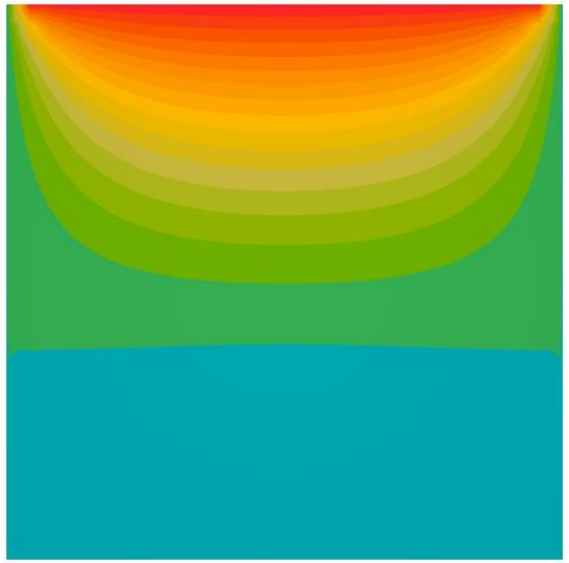
Results (Neutral file - Pos3D)

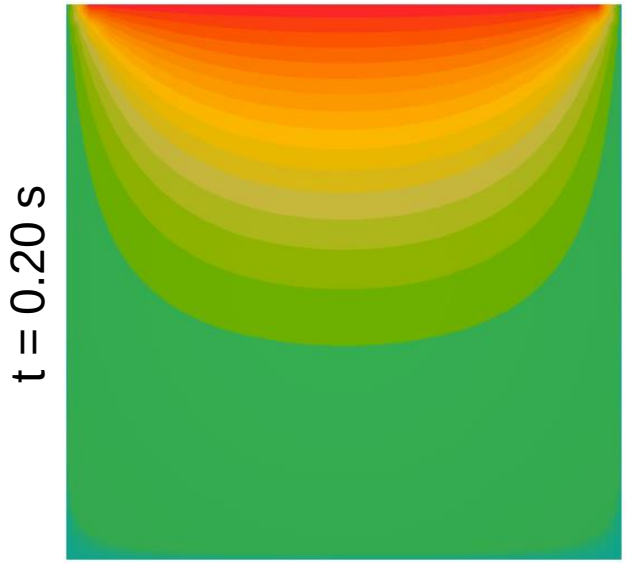


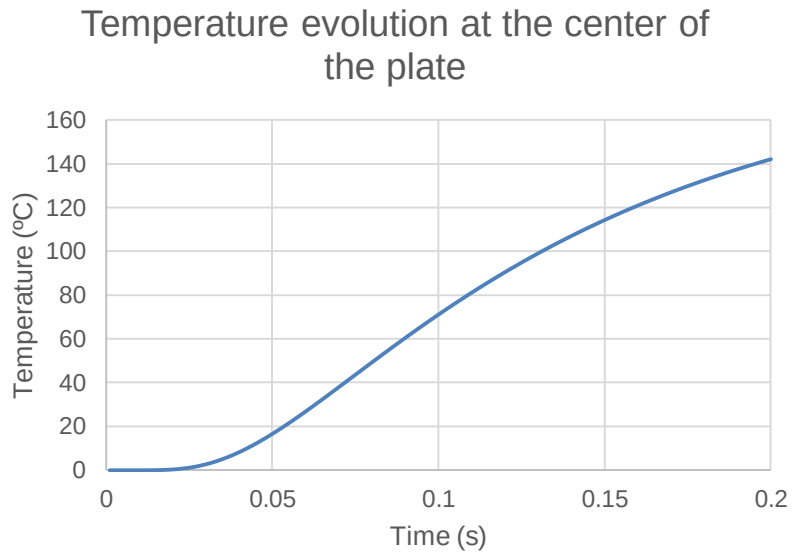




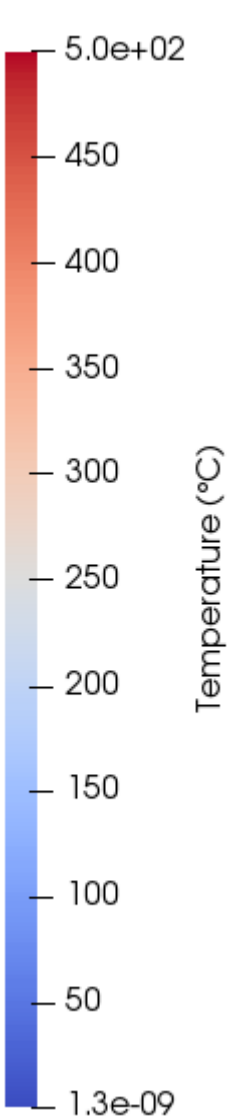


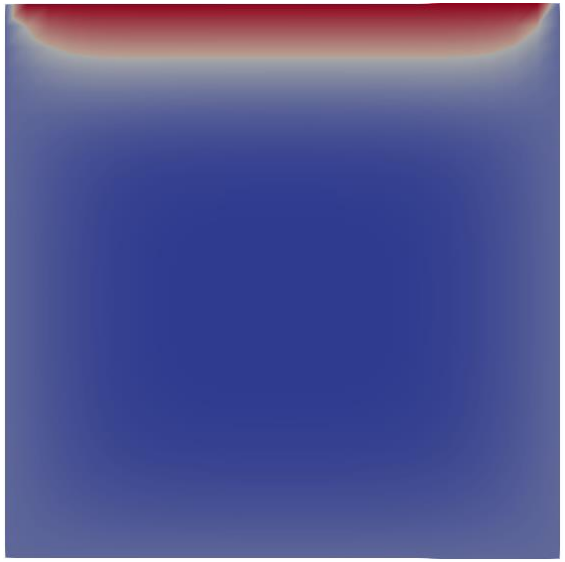


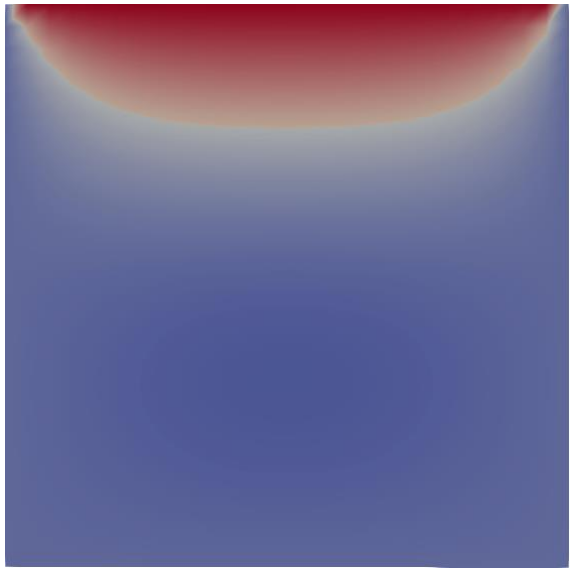


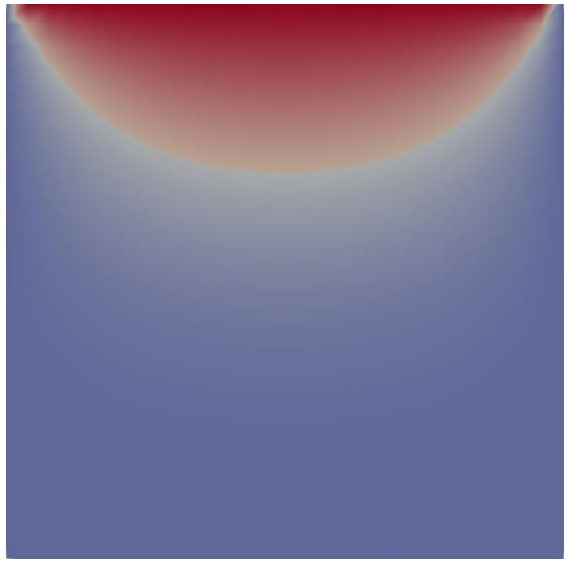


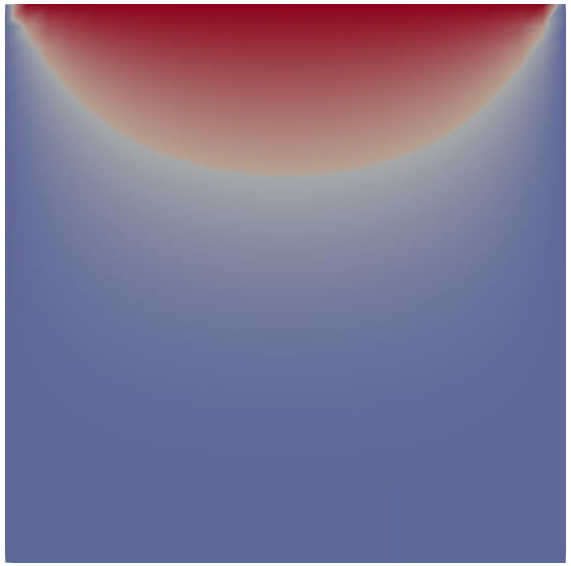
Results (Vtk - Paraview)

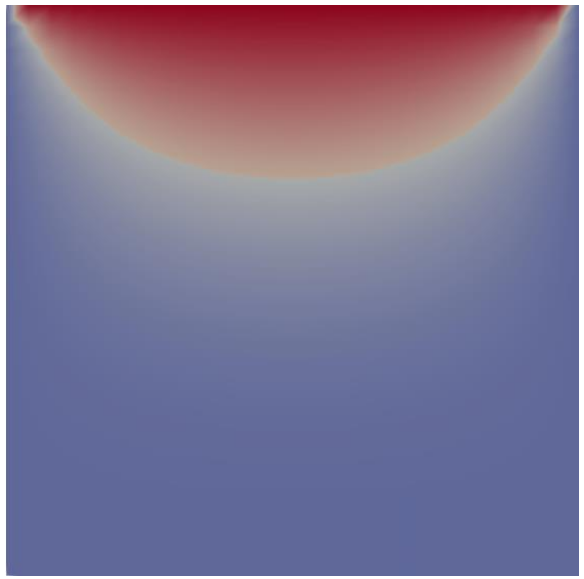


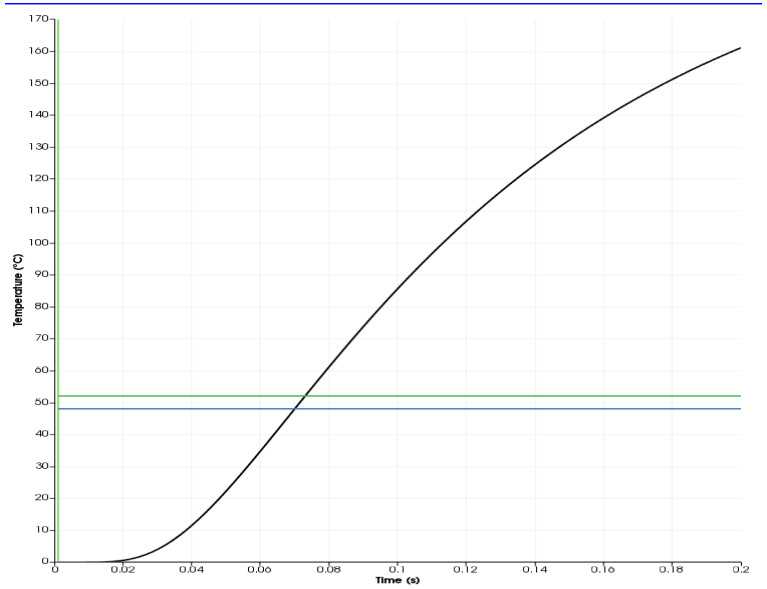








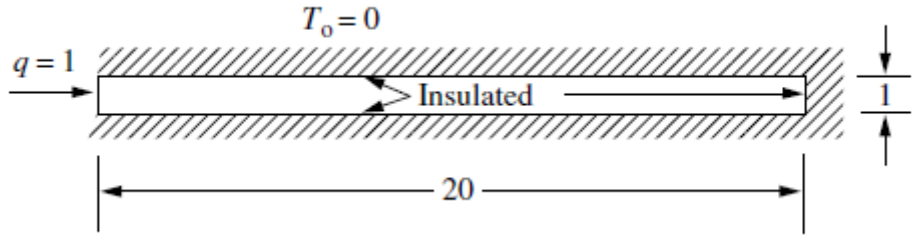




3 – TRANSIENT CONDUCTION ON A HEATED ROD

The Problem

- This example simulates the heat conduction on an insulated rod with a heat influx on one of its tips, comparing the numerical results to an analytical solution



- Analytical solution used to validate results:

$$T(x,t) = 2(t/\pi)^{1/2}\left[\exp\left(-x^2/4t\right) - (1/2)x\sqrt{\frac{\pi}{t}}erfc\left(\frac{x}{2\sqrt{t}}\right)\right]$$

Reference: "Fundamentals of the Finite Element Method for Heat and Fluid Flow", Lewis at al.

Example 6.4.2

Key Example Points

- The model structure for this simulation is similar to the previous ones. The main features not seen on previous models are:
  - Setup a Neumann boundary condition by prescribing border heat fluxes
  - Setup time-dependent node user functions
  - Print options for the FEM solver

- This example builds heavily on the previous ones and only shows key difference points. Please refer to the simulation files for the complete source.

Simulation file: HeatedRod.lua

Model file: Analytical solution

On the model file, two attributes will be added to the mesh for storing both the analytical solution and the error between the numerical and analytical calculations. Both attributes will be tied to user node functions.

```
-- The standard erfc(x) function
local function erfc(x)
   local p  =  0.3275911
   local a1 =  0.254829592
   local a2 = -0.284496736
   local a3 =  1.421413741
   local a4 = -1.453152027
   local a5 =  1.061405429
   local t  = 1/(1+p*x)

   return t * (a1 + t * (a2 + t * (a3 + t * (a4 + t * a5)))) * math.exp(-x*x)
end

-- A function to calculate the analytical model result at the given x coordinate and simulation time.
local function anaT(x, t)
  local a = 2 * math.sqrt(t/math.pi)
  local b = math.exp(-x*x/(4*t))
  local c = 0.5 * x * math.sqrt(math.pi/t) * erfc(x/(2*math.sqrt(t)))

  return a * (b - c)
end
```

$$T(x,t) = 2(t/\pi)^{1/2}\left[\exp\left(-x^2/4t\right) - (1/2)x\sqrt{\frac{\pi}{t}}erfc\left(\frac{x}{2\sqrt{t}}\right)\right]$$

Model file: Analytical solution (continued)

```
-- A node user function for calculating the error between the analytical and the numerical solutions
NodeFunction { id = 'errf',
               parameters = { {src = 'Tana'},
                              {src = 'T'},
                            },
               method = function(Tana, T) return Tana - T end
             }

-- A node user function for calculating the analytical solution
NodeFunction { id = 'ana',
               parameters = { {src = 'coordinate', dim=1},
                              {src = 'time'},
                            },
               method = function(x, t) return anaT(x, t) end
             }
Mesh
{
  ...
  nodeAttributes = {
    {id = 'Tana', description = 'Expected temperature analytically calculated',
                  functions = true, defVal = 'ana',  format = '10.5'},
    {id = 'Err',  description = 'Error between expected and calculated values',
                  functions = true, defVal = 'errf', format = '10.5'}
 },
 ...
}
```

The analytical temperature from node attribute Tana

The node temperature from state var T

The first dimension (x) of the node coordinate

The current simulation time

Err supports functions and its default value is the errf node function.

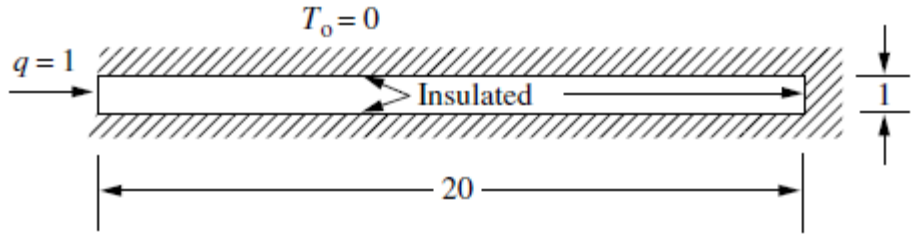
Model file: Boundary condition

For prescribing Neumann boundary conditions, the Thermo physics provides the ‘surface flux’ boundary condition type where heat flux values are associated with mesh edges. There is no need to specify insulated borders since, on the FEM method, borders with no prescribed condition will automatically be insulated.

```
BoundaryCondition {
  id    = 'Heated Border',
  type = 'surface flux',
  mesh = 'mesh',

  properties  = {
    {id = 'q',  description = 'Heat flux'},
  },

  edgeValues = {
    {'gridLeft', -1}, -- Negative value since the flow enters the rod
  }
}
```

Boundary condition type for prescribed heat flux

Boundary conditions. Each table line associates an edge set to a flux value, normal to the edges



Solution File

The orchestration for this simulation is very similar to the orchestration of the previous example. It features the global time loop for advancing the simulation. It also shows how to pass some print options to the Fem process. This options are useful when debugging new physics plugins.

```
function ProcessScript()
  -- Options for printing internal FEM matrices. Uncomment the desired flags before running.
  -- Refer to the Fem process documentation for other options
  local printOptions = {
    -- assembledMatrix = true,  -- The assembled matrix will be printed
    -- elementMatrices = true,  -- All element matrices will be printed
    -- assembledVector = true,  -- The assembled vector will be printed
    -- elementVectors  = true,  -- All element vectors will be printed
  }
```

Uncomment the desired lines above for seeing their effect on the output

```
  -- Creates the FEM solver used for calculating the numerical solution at each time step
  local solver = fem.initTransientSolver({'HeatPhysics'}, 'solver', {printOptions = printOptions})

  local dt     = 0.05      -- The time step
  local endt   = 10.0      -- Simulation duration
  local nsteps = endt / dt -- Number of steps

  -- Load VtkLib to enable saving results in the Vtk format
  dofile('$SCRIPTS/vtkLib.lua')
```

Use the print options table defined above

… continues on the next slide

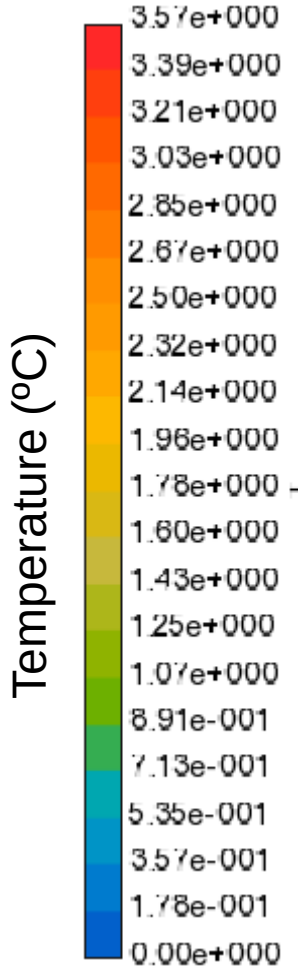
Solution File (continued)

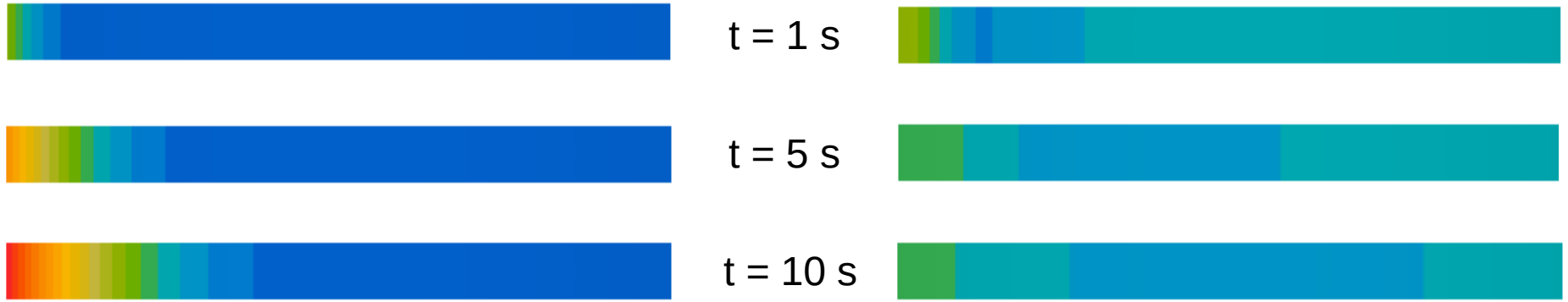
… continued from previous slide

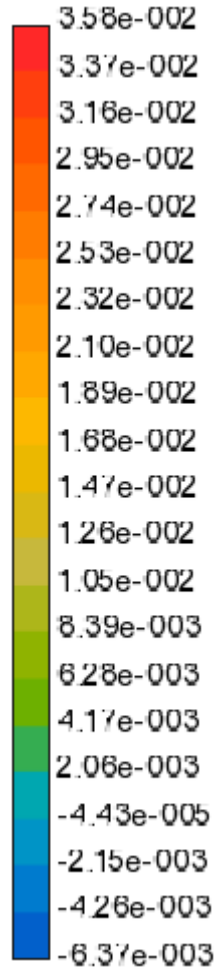
```
-- Creates the output simulation file
local file = io.prepareMeshFile('mesh', '$SIMULATIONDIR/out/$SIMULATIONNAME.nf', 'nf',
                                {'T', 'Tana', 'Err'})
-- Simulation time loop
for i=1, nsteps do
  -- Set the current time in the model
  setCurrentTime(i*dt)

  -- Solve one time step
  fem.transientStep(solver, dt)

  -- Print step solution
  io.printMeshNodeData('mesh', {'coordinate', 'T', 'Tana', 'Err'},
                       {header_title = true, eval_functions=true})

  -- Add step results to the ouptut file using neutral file and vtk formats
  io.addResultToMeshFile(file, i * dt)
  vtkLib.saveMeshFile('mesh', {'coordinate','T','Tana','Err'}, nil,
                      '$SIMULATIONDIR/out/$SIMULATIONNAME', {state = i, stateTime = i * dt})
end

-- Close output file
io.closeMeshFile(file)
end
```

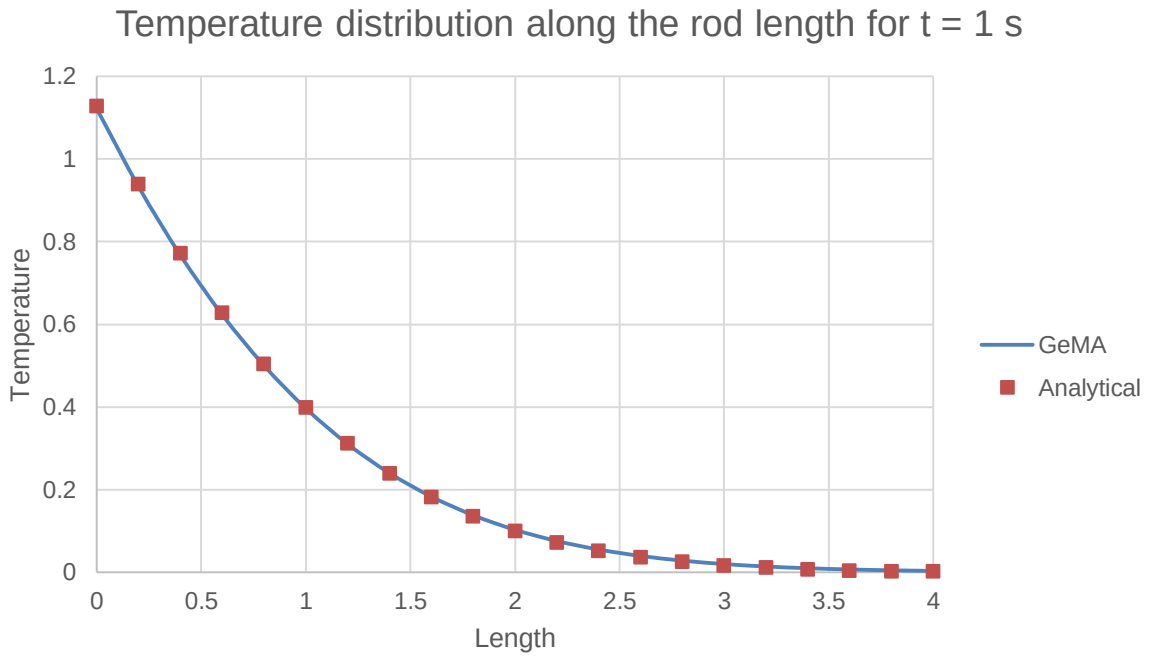
Needed for the nodal user function ‘ana’ to receive the current time

Results


Temperature (ºC)
3.57e+000
3.39e+000
3.21e+000
3.03e+000
2.85e+000
2.67e+000
2.50e+000
2.32e+000
2.14e+000
1.96e+000
1.78e+000
1.60e+000
1.43e+000
1.25e+000
1.07e+000
8.91e-001
7.13e-001
5.35e-001
3.57e-001
1.78e-001
0.00e+000


t = 1 s
t = 5 s
t = 10 s


Error (ºC)
3.58e-002
3.37e-002
3.16e-002
2.95e-002
2.74e-002
2.53e-002
2.32e-002
2.10e-002
1.89e-002
1.68e-002
1.47e-002
1.26e-002
1.05e-002
8.39e-003
6.28e-003
4.17e-003
2.06e-003
-4.43e-005
-2.15e-003
-4.26e-003
-6.37e-003


Temperature distribution along the rod length for t = 1 s
Temperature
Length
1.2
1
0.8
0.6
0.4
0.2
0
0
0.5
1
1.5
2
2.5
3
3.5
4
GeMA
Analytical

4 – STEADY STATE TEMPERATURE ANALYSIS OF SEDIMENTARY LAYERS

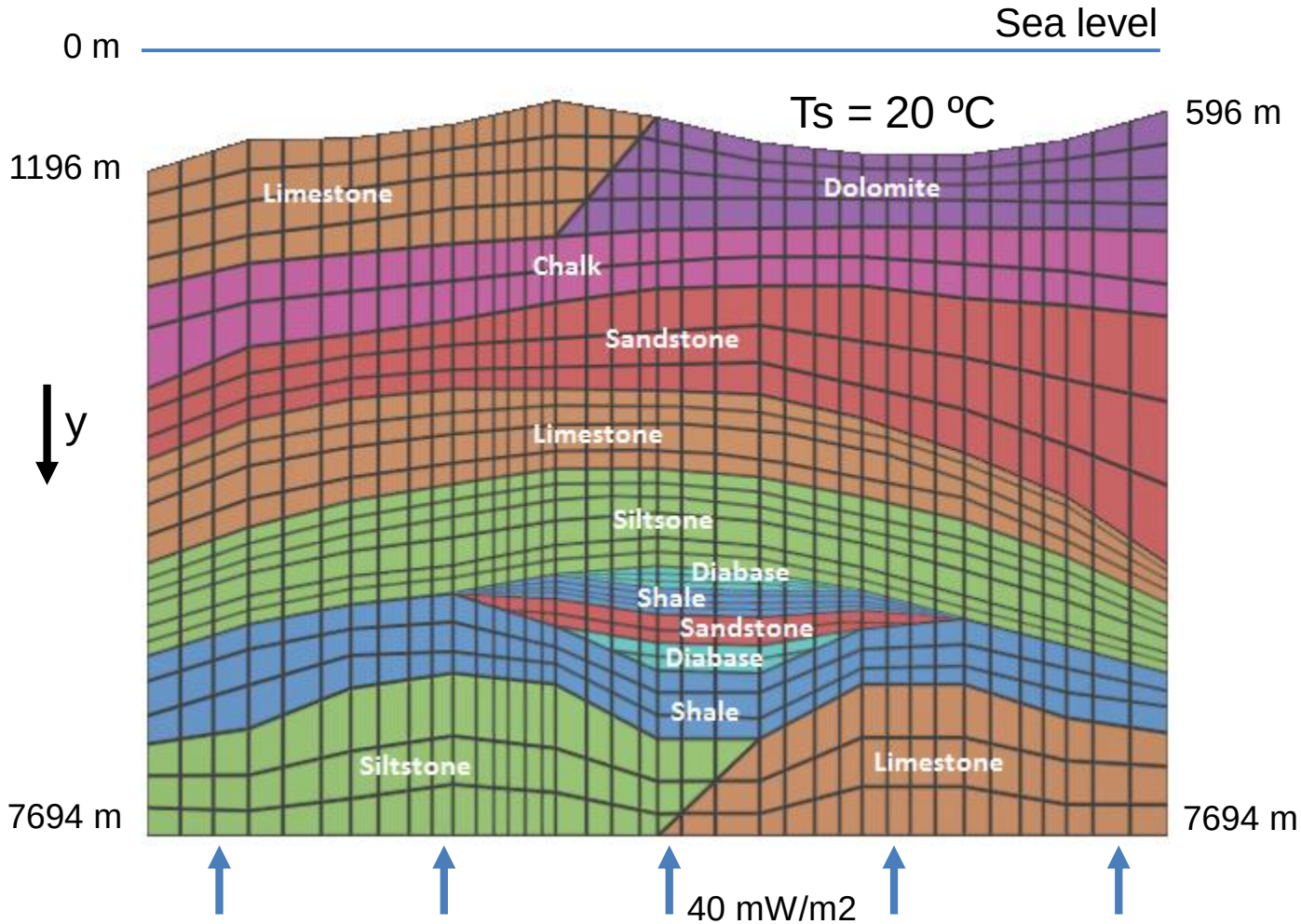
VERSION 1

The Problem

- Analyze the steady state temperature profile for a set of sedimentary layers subjected to a heat flux at their base and a prescribed surface temperature.

Reference: Material properties used on this example are given by “Fundamentals of Basin and Petroleum Systems Modeling”, Hantschel and Kauerauf.

Disclaimer: The geological model used in this example is a dummy test model and does not intend to resemble any real geological situation.



Key Example Points

- The model structure for this simulation is similar to the previous ones. The main features not seen on previous models are:
  - How to create the mesh node and element tables “manually” (without using auxiliary functions) for an heterogeneous mesh composed by quadrilaterals and triangles.
  - How to setup several materials and work with anisotropies.
  - How to enable heat generation on the model.
  - How to parameterize a simulation with used provided values at run-time.
  - How to change the units of saved and printed values.

- This example builds heavily on the previous ones and only shows key difference points. Please refer to the simulation files for the complete source.

Simulation file: BasinLayers1.lua

Model file: Handling user parameters

GeMA models can be parameterized at run time with user provided values passed as command line parameters with the –u switch. On this simulation, we will require the user to provide a true or false value that will be used to enable or disable radiogenic heat generation in our simulation.

Provided parameters are accessed while building the model through the `userParams` Lua variable.

```
-- Turns Radiogenic Heat generation on or off by reading the
-- required user provided parameter
if userParams == nil then
  print()
  print('This simulation must be called with a boolean user parameter to turn')
  print('Radiogenic Heat generation on or off.')
  print('Please run it passing -u true or -u false as parameters.')
  print()
  assert(nil) -- Abort the simulation
end

enableQ = userParams
```

Make sure that the user provided us with a value. If not (`userParams == nil`), prints a message and aborts the simulation

Stores the user choice in the enableQ variable for latter use at the solution file

Model file: Material properties

Material properties are given in the usual way. Since our model features 7 different lithologies, our property set table has 7 value lines. To ease the process of attaching lithologies to mesh cells, materials are labeled with their lithology name. Although this is a steady state problem, our table also includes density and specific heat columns needed by the transient analysis on example 6.

```
PropertySet
{
  id          = 'LithoPSet',
  typeName    = 'GemaPropertySet',
  description = 'Lithological parameters for facies',
  properties  = {
    {id = 'kv',  description = 'Grain vertical conductivity @ 20 degC',  unit = 'W/(m.K)'},
    {id = 'ka',  description = 'Grain conductivity anisotropy'},
    {id = 'k',   description = 'Conductivity',                           unit = 'W/(m.K)', dim = '2x2',
                 functions = true, defVal = 'rotk'},
    {id = 'rho', description = 'Grain density',                          unit = 'kg/m3'},
    {id = 'cp',  description = 'Grain specific heat capacity @ 20 degC', unit = 'J/(kg.K)'},
    {id = 'Q',   description = 'Grain radiogenic heat generation rate',  unit = 'uW/m3'},
    {id = `h',   description = 'Element thickness',                      unit = 'm', defVal = 1.0},
  },
  values = {
    {id = 'sandstone', rho = 2720, kv = 3.95, ka = 1.15, cp = 855, Q = 0.70},
    {id = 'shale',     rho = 2700, kv = 1.64, ka = 1.60, cp = 860, Q = 2.03},
    {id = 'siltstone', rho = 2710, kv = 2.01, ka = 1.71, cp = 940, Q = 0.96},
    {id = 'chalk',     rho = 2680, kv = 2.90, ka = 1.07, cp = 850, Q = 0.60},
    {id = 'limestone', rho = 2680, kv = 2.00, ka = 1.95, cp = 845, Q = 1.40},
    {id = 'dolomite',  rho = 2790, kv = 4.20, ka = 1.06, cp = 860, Q = 0.29},
    {id = 'diabase',   rho = 2800, kv = 2.60, ka = 1.00, cp = 800, Q = 0.18},
  }
}
```

k is a 2x2 matrix instead of a scalar

This example works with an anisotropic conductivity. See the next slide.

Notice that this table does not provides values for the k and t properties. In their absence, the value given by defVal will be used, meaning that every lithology type will have h = 1.0 and k = ‘rotk’.

Optional name associated to each material row. This last material can be referenced either by its row (7) or by its name (diabase).

Model file: Anisotropic conductivity

Layer conductivities are not isotropic. Its value in the direction normal to the layer is much less than the conductivity in the direction parallel to the layer bedding. The GeMA thermo plugin can work with anisotropic scenarios if the conductivity 'k' in the property set is given as a 2x2 tensor (3x3 for 3D problems). One problem though, is that it expects the tensor to be aligned with Cartesian axis. A solution is making the conductivity a user function that rotates the tensor aligning it with the layer dip. In our model, the basic material vertical (normal) conductivity is given by 'kv', while the horizontal (parallel) conductivity is given by multiplying 'kv' by an anisotropy factor 'ka'. Those two properties are not recognized by the physics plugin, they are used instead by the cell function attached to the expected property 'k' to calculate the conductivity tensor. The layer dip is stored per cell in an attribute associated with the mesh and initialized together with mesh data.

```
-- User function to rotate the conductivity tensor according to the layer dip
CellFunction
{
  id = 'rotk',
  parameters = { {src = 'kv'},  -- The grain vertical conductivity
                 {src = 'ka'},  -- The grain anisotropy factor (kh = kv * ka)
                 {src = 'dip', unit = 'rad'}, -- The layer dip in radians
               },
  method = function(kv, ka, dip)
    local ca = math.cos(dip)
    local sa = math.sin(dip)
    local R  = Matrix{{ca, -sa}, {sa, ca}} -- Rotation matrix
    local K  = Matrix{{kv*ka, 0}, {0, kv}} -- Axis aligned conductivity tensor
    local rK = R * K * R:t()
    return rK:toTable()
  end
}
```

The Matrix call creates a matrix object assigned to the local variable R. Matrix objects can be used on arithmetic expressions producing other matrix objects (rK, for example). They export some methods like t() which returns the transposed matrix and toTable() that converts the matrix data to a linearized Lua table in column major format.

Model file: Mesh definition

To separate the mesh geometry from the control code specifying its characteristics, the geometry is read from an external file (shared by this example and the next two) through a Lua `dofile()` call. The heterogeneous mesh used on this example, combining quad4 and tri3 elements, was created by a pre-processor and converted to the expected GeMA syntax with a home-made script.

The additional node attribute “burialDepth” is created to store each node’s coordinate, discounting the bathymetry from its depth. This information will be needed on the next example to calculate porosities. Its value is initialized through the nodes table.

```
-- Loads the mesh definition from an external file. The loaded mesh uses a coordinate system
-- with 0.0 at the surface and positive values for deeper layers
local nodes, elements, borders = dofile('$SIMULATIONDIR/BasinLayers_meshData.lua')

Mesh
{
  ...
  -- The mesh also stores a node attribute with the node's burial depth (actual depth - bathymetry)
  -- Its value is initialized from the mesh file.
  nodeAttributes = {
    {id = 'burialDepth', description = 'Burial depth coordinate', dim = 2, unit = 'km', format = '7.4f'},
  },
  ...
}
```

Dimensions for this model are given in km.

Model file: Mesh definition

The additional cell attribute “dip” is created to store the layer dip at each cell. Its value is initialized through the elements table and is used to align the conductivity tensor to the layer dip, as seen before. The “phi” Gauss attribute will be used on the next example to store layer porosities at each element Gauss points. Those values will be initialized from the orchestration script.

```
Mesh
{
  ...
  -- The mesh stores a cell attribute with the layer dip on that cell.
  -- Its value is initialized from the mesh file.
  cellAttributes = {
    {id = 'dip', description = 'The layer dip (angle between the layer and the horizontal reference).',
                 unit = 'rad'},
  },

  -- Attribute used for storing calculated porosity at Gauss points.
  -- Its value will be initialized from the orchestration.
  gaussAttributes = {
    {id = 'phi', description = 'The porosity calculated at Gauss points', unit = 'V/V'},
  },
  ...
}
```

A practical way to specify that phi stores a value between 0.0 and 1.0 and not a percentage value.

Model file: Mesh data - nodes

The node data from the mesh geometry file is given by a table where each entry is another table storing node coordinates, followed by the initial values for the burial depth.

```
local nodeData = {
  --    x,         y,         burialDepth
  {  0.0000,  7.6939, { 0.0000, 6.4976} },
  {  0.3333,  7.6939, { 0.3333, 6.6022} },
  {  0.6553,  7.6939, { 0.6553, 6.7033} },
  ...
}


...


return nodeData, cellData, borderData
```

The third value for each node line is used to initialize the burialDepth attribute. Since the burialDepth is a vector value, its components are included in a sub-table. Values are given in the attribute unit (km).

The nodeData table is the first table returned by the dofile() call used to load the mesh geometry file.

The node data table should store node coordinates, optionally followed by node attribute initial values (in their definition order), followed by state variable initial values (also in their definition order). Missing values are filled with defaults. For an attribute / state variable value to be provided, all previous values must be given (a nil value can be used to represent the default value).

Model file: Mesh data - elements

Mesh elements are grouped by layer and by element type (quad4 elements should be provided in separate tables from tri3 elements). Each element definition includes the ordered set of element nodes + the dip value to initialize cell attributes in the same way as node attributes are initialized.

```
local quad4CellTable_4 = {
  -- node list,          dip
  { 47, 48, 83, 85,    0.0299 },
  { 85, 86, 46, 47,    0.0299 },
  { 86, 87, 44, 46,    0.0299 },
  { 50, 44, 87, 89,   -0.1190 },
  ...
}
local tri3CellTable_5 = {
  -- node list,     dip
  { 81, 43, 80,    0.0031 },
  { 43, 81, 42,    0.0031, LithoPSet = 5 },
}
...
local cellData = {
  ...
  {cellType = 'quad4', cellGroup = 'layer35', cellList = quad4CellTable_4, LithoPSet = 3},
  {cellType = 'tri3',  cellGroup = 'layer35', cellList = tri3CellTable_5,  LithoPSet = 3},
  ...
}
```

Initial values for the dip attribute given in the attribute unit (radians)

Element definition

quad4 node numbers in CCW order

Specific material value for this element. When missing, the element uses the default provided by the cell group

tri3node numbers in CCW order

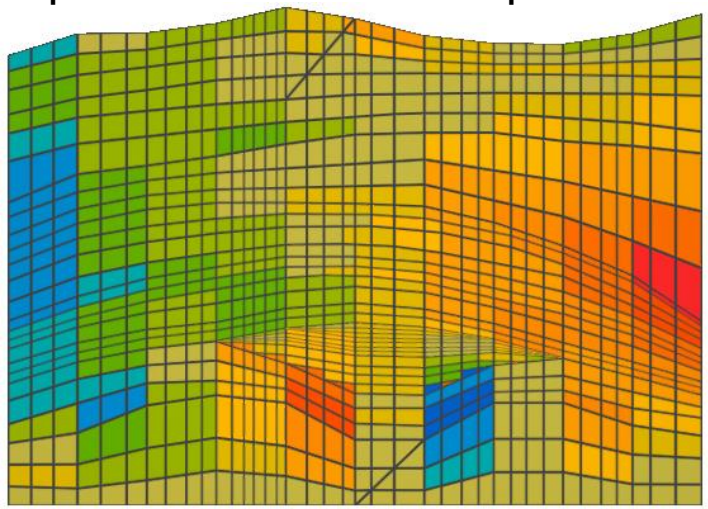
The cell group name. If distinct lines of cellData share the same cellGroup name, they will be merged into a single group

The default LithoPSet value for this group of elements

The element type for a group of elements

Table with element definitions for a group of elements

Dip values for the example mesh



Model file: Mesh data - borders

Border names are used to apply boundary conditions. In 2D, each named border is given by a list of mesh edges, each specified by a cell number + an edge number.

```
local edgesTable_1 = {
  {1091, 3}, {1092, 1}, {1093, 1}, {1094, 3}, {1095, 1}, {1096, 1}, {1097, 3}, {1098, 1}, {1099, 1}, {1100, 1},
  {1101, 2}, {1102, 2}, {1103, 2}, {1104, 2}, {1105, 2}, {1106, 2}, {1107, 2}, {1108, 2}, {1109, 2}, {1110, 1},
  {1112, 3}, {1113, 3}, {1114, 2}, {1115, 3}, {1116, 1}, {1117, 1}, {1118, 1}, {1119, 3}, {1120, 3}, {1121, 3},
  {1122, 1}, {1123, 3}, {1124, 3}, {1125, 3}, {1126, 1}, {1127, 3}, {1128, 3}, {1129, 1},
}
```

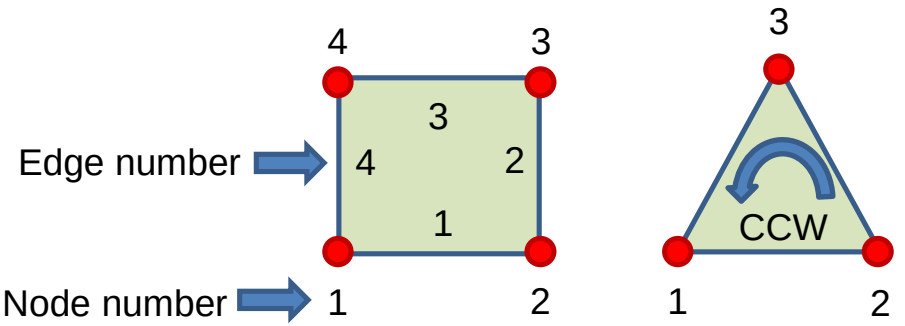
The cell number — The edge number

```
local edgesTable_2 = {
  ...
}

local borderData = {
  {id = 'basinTop',    cellList = edgesTable_1},
  {id = 'basinBottom', cellList = edgesTable_2},
}
```

The border name

```
return nodeData, cellData, borderData
```

The borderData table is the third table returned by the dofile() call used to load the mesh geometry file



Edge numbering for linear 2D elements follows a simple rule: The edge between the first and second vertices is edge 1, the edge between the second and third vertices is edge 2, and so on. Check the documentation for the complete edge numbering for other elements.

Model file: Heat flow boundary condition

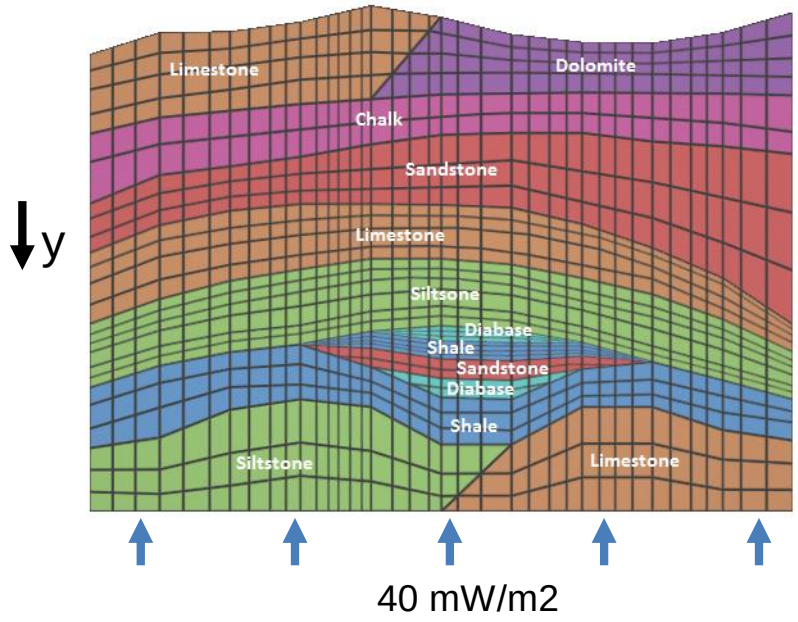
For this model, the heat flux direction is being specified by a direction vector. Since our base border is horizontal and the flow normal to the base, we could have skipped the direction and used a -40 value to specify inflow. In order to make the model more general and applicable to other basins with irregular geometry, the flow direction is given. In that case, the applied flow will be the projection of the given flow onto the element normal.

```
-- Basal heat flow boundary condition
BoundaryCondition {
  id   = 'Basal heat flow',
  type = 'surface flux',
  mesh = 'mesh',

  properties  = {
    {id = 'q',    description = 'Heat flow', unit = 'mW/m2'},
    {id = 'qDir', description = 'Heat flow direction', dim = 2}
  },


  edgeValues = {
    {'basinBottom', 40, {0, -1}}, -- Direction is along the negative y
  }                               -- since our depth axis is inverted
}
```

Boundary condition type for prescribed heat flux

qDir is a vector with 2 components
Its value MUST be normalized

Bottom border name created on the mesh definition



Solution file: Physics

```
PhysicalMethod {
  id         = 'HeatPhysics',
  typeName = 'ThermoFemPhysics',
  type       = 'fem',

  mesh                 = 'mesh',
  boundaryConditions = {'Surface-Water border temperature', 'Basal heat flow'},


  enableGeneration = enableQ,

  properties = {G = 'Q'}, -- Translate propertySet 'Q' to physics expected 'G'
}
```

Radiogenic heat generation is enabled or disabled according to the value of the enableQ variable, initialized with the user given parameter at the beginning of the model file.

When heat generation is turned on, the ThermoFemPhysics plugin expects the volumetric generation rate to be given by the ‘G’ property. In our property set, as an example, that value was named ‘Q’. The properties table supplies name translations for property sets, telling the physics that the expected ‘G’ property is in fact named ‘Q’.

Solution file: Orchestration

```
function ProcessScript()

  -- Solves the model using the FEM method for linear problems
  fem.solve({'HeatPhysics'}, 'NumSolver')

  -- Print results (dip values will be converted from radians to degrees)
  print('\n\nCalculated results:')
  io.printMeshNodeData('mesh', {'coordinate', 'T', 'burialDepth'})
  io.printMeshCellData('mesh', {'dip(degree)'})

  -- Save results on the out sub-directory of the same directory hosting the simulation model
  -- Saved file will have the same name as the simulation file
  -- The saveMeshFile() call expects the 'out' sub-directory to exist
  -- The scaleFactorY parameter is used to invert the depth values providing a better viewing
  -- experience (Remember that our mesh y axis is inverted)
  io.saveMeshFile('mesh', '$SIMULATIONDIR/out/$SIMULATIONNAME.nf', 'nf', {'T', 'burialDepth'}, {'dip(degree)'},
                  {scaleFactorY = -1.0, materialPropertySet = 'LithoPSet'})

  -- The mesh can also be saved using the burial depth as the Y coordinate:
  io.saveMeshFile('mesh', '$SIMULATIONDIR/out/$SIMULATIONNAME_burial.nf', 'nf', {'T'}, {'dip(degree)'},
                  {scaleFactorY = -1.0, materialPropertySet = 'LithoPSet', coordAttribute = 'burialDepth'})
end
```

Although the dip attribute is stored in radians, we want it to be printed / saved in degrees

Depth values are multiplied by -1 before being saved

Specifies which property set table should be saved on the file

Replaces saved mesh node coordinates by the values of the given node property.
On this case, will save a mesh where the y origin is the top layer (no bathymetry).

Running the simulation

- As seen before, this model expects the user to provide a parameter stating if it wants to run the simulation with or without considering radiogenic heat generation.
  - User parameters are provided with the –u switch.

- At the command prompt type either

```
C:\> gema BasinLayers1.lua –u true
```

  to enable radiogenic heat generation or

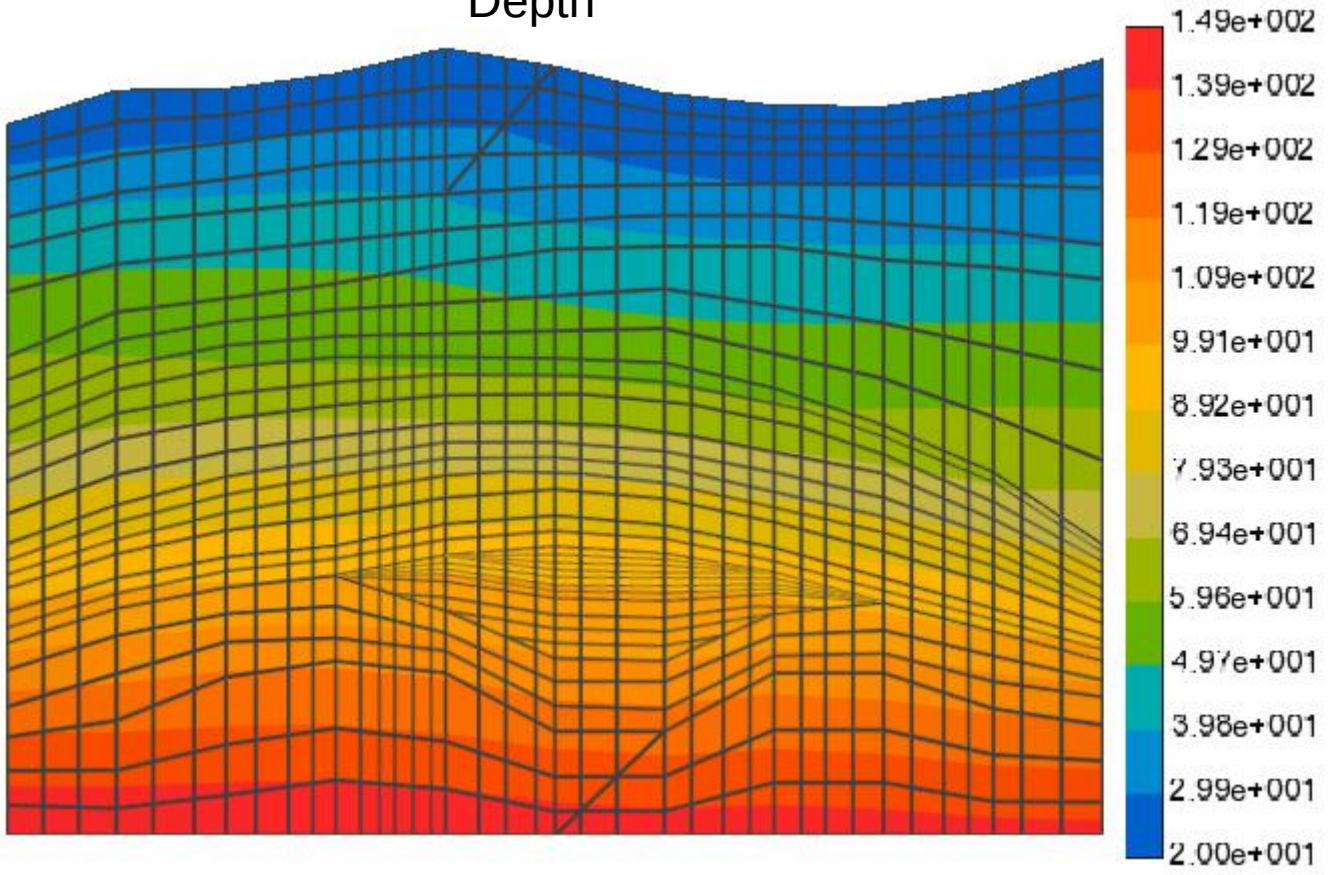
```
C:\> gema BasinLayers1.lua –u true
```

  to disable radiogenic heat generation.

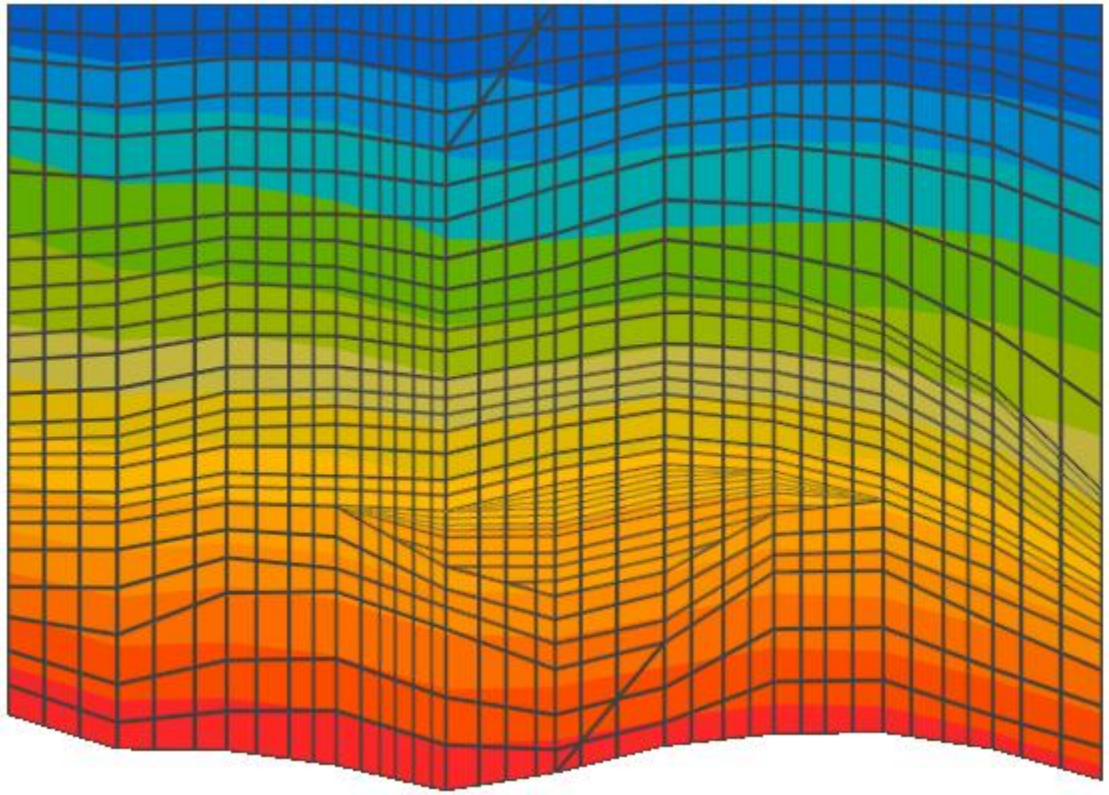
- Failure to provide the parameter will generate an error message.

Results: Temperature distribution (ºC)

Depth



Burial depth

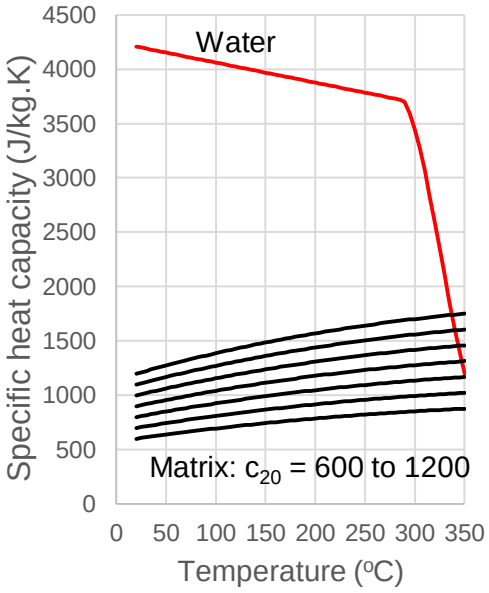


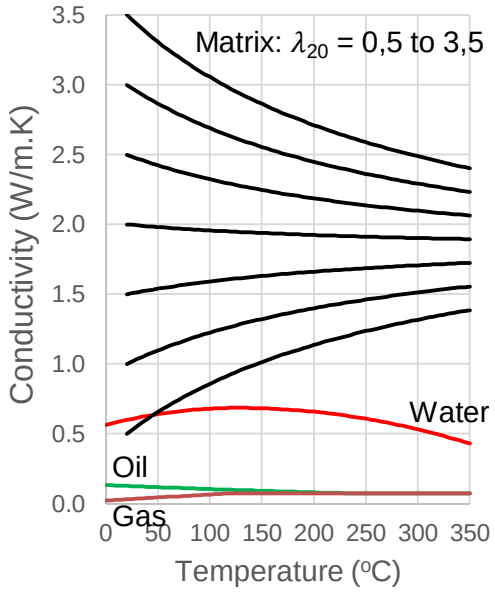
5 – STEADY STATE TEMPERATURE ANALYSIS OF SEDIMENTARY LAYERS

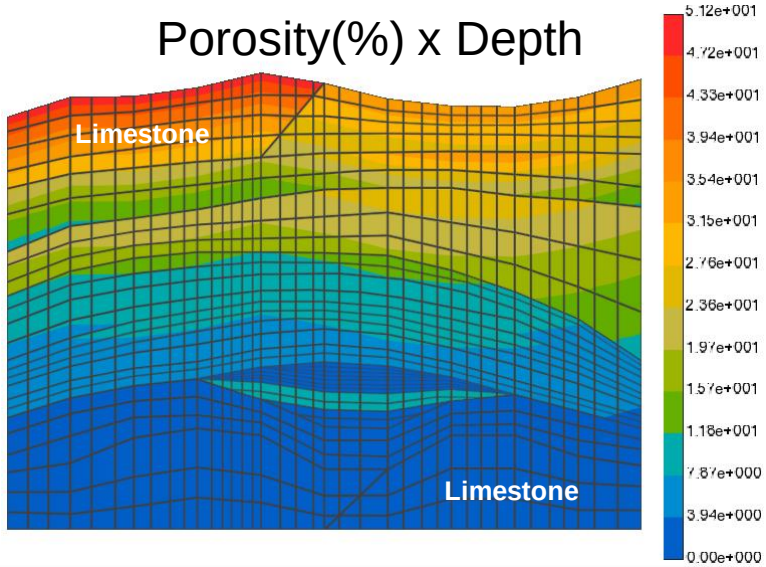
VERSION 2

The Problem

- This simulation revisits the previous example by taking into account layer porosities and non-linearities arising from the conductivity and the specific heat capacity being functions of the temperature.
- Layer porosities are a function of the layer depth, so simply using sets of equivalent properties is not a good solution.







Reference: Mixing laws and temperature dependency models for conductivity and heat capacity used in this example where taken from “Fundamentals of Basin and Petroleum Systems Modeling”, Hantschel and Kauerauf.

Key Example Points

- The model structure for this simulation is similar to the previous ones. The main features not seen on previous models are:
  - How to change the material properties table to account for layer porosity and temperature dependencies.
  - How to initialize values at element Gauss points at the orchestration (for storing layer porosities).
  - How to setup a non-linear analysis using the linear FEM process.

- This example builds heavily on the previous ones and only shows key difference points. Please refer to the simulation files for the complete source.

Simulation file: BasinLayers2.lua

Model file: Material properties

The set of expected material properties (k, cp, rho and Q) are now given by mixing functions that take grain properties and the porosity as input parameters. Those functions are also responsible for temperature corrections and tensor rotations.

```
PropertySet {
  id          = 'LithoPSet',
  typeName    = 'GemaPropertySet',
  description = 'Lithological parameters for facies',
  properties  = {
    {id = 'phic',  description = 'Compaction coefficient for Athy law',          unit = '1/km'    },
    {id = 'phi0',  description = 'Initial porosity for Athy law',                unit = '%'       },
    {id = 'g_rho', description = 'Grain density',                               unit = 'kg/m3'   },
    {id = 'g_kv',  description = 'Grain vertical conductivity @ 20 degC',       unit = 'W/(m.K)' },
    {id = 'g_ka',  description = 'Grain conductivity anisotropy'                                 },
    {id = 'g_cp',  description = 'Grain specific heat capacity @ 20 degC',      unit = 'J/(kg.K)'},
    {id = 'g_Q',   description = 'Grain radiogenic heat',                        unit = 'uW/m3'   },
    {id = 'rho',   description = 'Mixed fluid-grain density',                    unit = 'kg/m3',    defVal = 'rho_phi',
                   functions = true},
    {id = 'k',     description = 'Mixed fluid-grain conductivity', dim = '2x2', unit = 'W/(m.K)',  defVal = 'rotk_phi_T',
                   functions = true},
    {id = 'cp',    description = 'Mixed fluid-grain specific heat capacity',    unit = 'J/(kg.K)', defVal = 'cp_phi_T',
                   functions = true},
    {id = 'Q',     description = 'Mixed fluid-grain radiogenic heat',           unit = 'uW/m3',    defVal = 'Q_phi',
                   functions = true},
    {id = 'h',     description = 'Element thickness', unit = 'm', defVal = 1.0},
  },
  values = {
    {id = 'sandstone', phic = 0.31,  phi0 = 41.0, g_rho = 2720, g_kv = 3.95, g_ka = 1.15, g_cp = 855, g_Q = 0.70},
    ...
  }
}
```

Additional properties used by mixing functions and to calculate layer porosities.

Required properties by the Thermo Fem Physics.

Mixing functions

Athy law parameters for calculating phi(z).

Matrix grain material properties used for calculating mixed fluid-grain properties with temperature corrections.

Model file: Mixing functions

The mixing function for conductivities applies the required temperature corrections to the grain conductivity and to the water conductivity. Those values are then mixed by a geometric law and used to form the conductivity tensor that will then be rotated, as seen on the previous example. Other mixing rules are simpler and not shown here. Refer to the simulation file.

```
-- User function to rotate the conductivity tensor according to the layer dip
CellFunction
{
  id = 'rotk_phi_T',
  parameters = { {src = 'g_kv'},  -- The grain vertical conductivity
                 {src = 'g_ka'},  -- The grain anisotropy factor (kh = kv * ka)
                 {src = 'dip', unit = 'rad'},  -- The layer dip in radians
                 {src = 'phi', unit = 'V/V'},  -- Porosity
                 {src = 'T',   unit = 'degC'}, -- Interpolated Temperature
               },
  method = function(kvG, ka, dip, phi, T)
    local khG = kvG * ka

    -- Calc Water conductivity following Deming model
    local fluid
    if T < 137 then
      fluid = 0.565 + 1.88e-3 * T - 7.23e-6 * T * T
    else
      fluid = 0.602 + 1.31e-3 * T - 5.14e-6 * T * T
    end

    -- Calc grain conductivity following Sekiguchi model
    local TK = T + 273.15 -- TK = T in Kelvin
    local grain_v = 358 * (1.0227*kvG - 1.882) * (1/TK - 0.00068) + 1.84
    local grain_h = 358 * (1.0227*khG - 1.882) * (1/TK - 0.00068) + 1.84

    -- Fluid and grain are mixed with a geometric law
    local mixed_v = (fluid ^ phi) * (grain_v ^ (1-phi))
    local mixed_h = (fluid ^ phi) * (grain_h ^ (1-phi))

    -- Axis aligned conductivity tensor
    local K = Matrix{{mixed_h, 0}, {0, mixed_v}}

    -- Rotate tensor aligning with the layer direction
    local ca = math.cos(dip)
    local sa = math.sin(dip)
    local R  = Matrix{{ca, -sa}, {sa, ca}}
    local rK = R * K * R:t()
    return rK:toTable()
  end
}
```

Solution file: Orchestration

The orchestration starts by calling a user defined function to initialize porosity values on the element Gauss points. The non-linearity due to k(T) is solved by repeated linear iterations. It could also be solved by the non-linear solver.

```
function ProcessScript()
  -- Before running the simulation, we must init mesh porosity values
  initPorosity()

  -- Solves the model using the FEM method
  -- Since the conductivity is a function of T, the model is non-linear.
  -- It could be solved by the non-linear solver, but to show the orchestration
  -- flexibility, we will solve it by repeated linear iterations.
  local solver  = fem.initLinearSolver({'HeatPhysics'}, 'NumSolver')
  local r       = 1.0
  local iter    = 1
  local maxIter = 10    -- The maximum number of allowed nonlinear iterations
  local tol     = 1e-5  -- The convergence tolerance

  while r > tol and iter <= maxIter do
    print('Executing linear step '..iter)
    fem.linearStep(solver)

    -- Evaluate error
    r = fem.linearResidual(solver)
    print(string.format('  r = %.6f', r))

    iter = iter + 1
  end
  assert(r <= tol, 'Failed to achieve convergence...')
  ...
end
```

Convergence loop

Executes one linear step

The solver object returned by fem.initLinearSolver()

Calculates the L2 norm of the solution error

Abort if the maximum number of iterations was reached

A FEM problem is ultimately transformed into a linear system K.x = f. For non-linear problems, both K and f can be functions of x. After iteration i, we have the values for x_i. The error measure is done by building a new matrix K and a new vector f based on the current x_i estimate and then calculating the L2 norm of the vector K.x_i – f.

Solution file: Initializing porosities

Porosity values are initialized on the element Gauss points by traversing all mesh elements. For each element, its integration points are traversed and the porosity calculated using the Athy law, based on the Cartesian coordinate of each point.

```
--
-- Initializes element porosity on Gauss points using the Athy porosity model:
--    phi(z) = phi0 * exp(-c*z)
--
-- Where:
--   z    : the BURIAL depth (without bathymetry), in m
--   phi0 : initial porosity in v/v (0 to 1)
--   c    : compaction coefficient in 1/m
--
function initPorosity()
  -- Get the needed accessors.  Notice that the accessors will retrieve the data in the
  -- needed units (which are different from the specified material units for phi0 and phic)
  local m = modelData:mesh('mesh')

  local coordAc = assert(m:nodeValueAccessor('burialDepth', 'm'))
  local phiAc   = assert(m:gaussAttributeAccessor('phi', 'V/V'))
  local phi0Ac  = assert(m:cellPropertyAccessor('phi0', 'V/V'))
  local phicAc  = assert(m:cellPropertyAccessor('phic', '1/m'))

  local ir, shape, etype -- Cache for integration rules and shape functions
```

Get the mesh object for the named mesh

Get accessor objects for the needed attributes and properties

Accessors are the objects used for querying and changing values associated to mesh nodes, cells and Gauss points. To get an accessor, all that is needed is the attribute/property/state variable name. Accessors can automatically convert units if the desired unit is different from the data unit.

… continues on the next slide

Solution file: Initializing porosities

```
for i=1, m:numCells() do
  local e = m:cell(i)

  -- Get the elements integration rule and shape functions
  if e:type() ~= etype then
    etype = e:type()
    ir    = assert(m:elementIntegrationRule(etype, 1))
    shape = assert(e:shape())
  end

  -- Get a matrix with cell node coordinates
  local X = e:nodeMatrix(coordAc, true)

  -- Get the values of phi0 and phic for this element
  local phi0 = phi0Ac:value(e)
  local phic = phicAc:value(e)

  -- Loop over the element integration points
  for j=1, ir:numPoints() do
    local ip    = ir:integrationPoint(j)          -- Get the integration point natural coordinates...
    local coord = shape:naturalToCartesian(ip, X) -- .. and converts them to cartesian coordinates

    -- Calc phi at the integration point depth using Athy law
    local phi = phi0 * math.exp(-phic*coord(2))

    -- Save calculated porosity
    phiAc:setValue(e, j, phi)
  end
 end
end
```

Loop over all the mesh elements

Get the element object representing element ‘i’

Get the integration rule and the shape function associated with the current element type. Since they change per element type, they don’t need to be recovered again if the current type is equal to the previous one.

Get a matrix with the current element node coordinates using the burial depth instead of the mesh coordinates as its base.

Cell properties are indexed by the element object e

Loop over all integration points

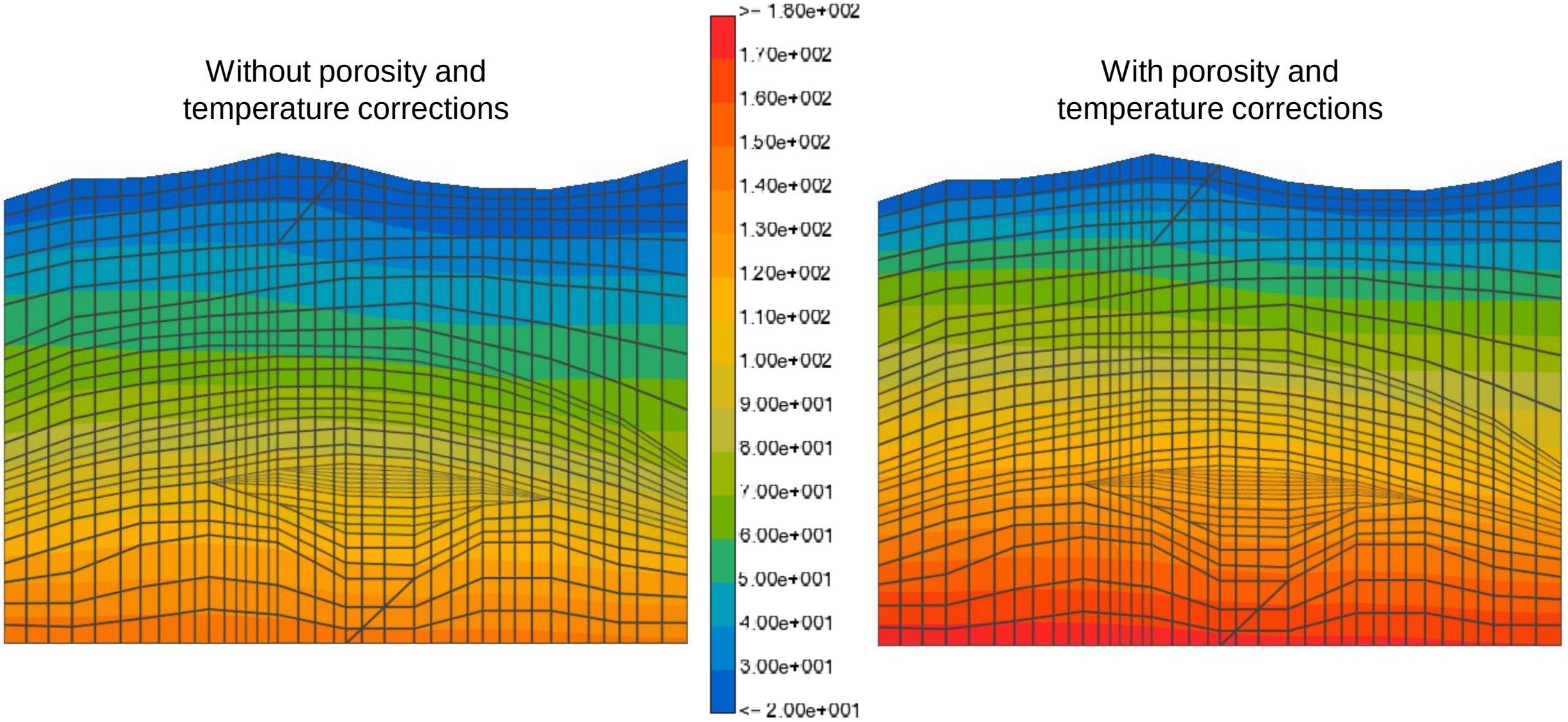
coord is a vector with dimension 2. The first dimension (x) is accessed through coord(1) while the second (y) through coord(2).

Gauss attributes are indexed by the element object + the integration point index

A full description of every available method to interact with GeMA objects is available at the orchestration reference manual at the GeMA project page.

Results: Temperature distribution (ºC)

Without porosity and temperature corrections

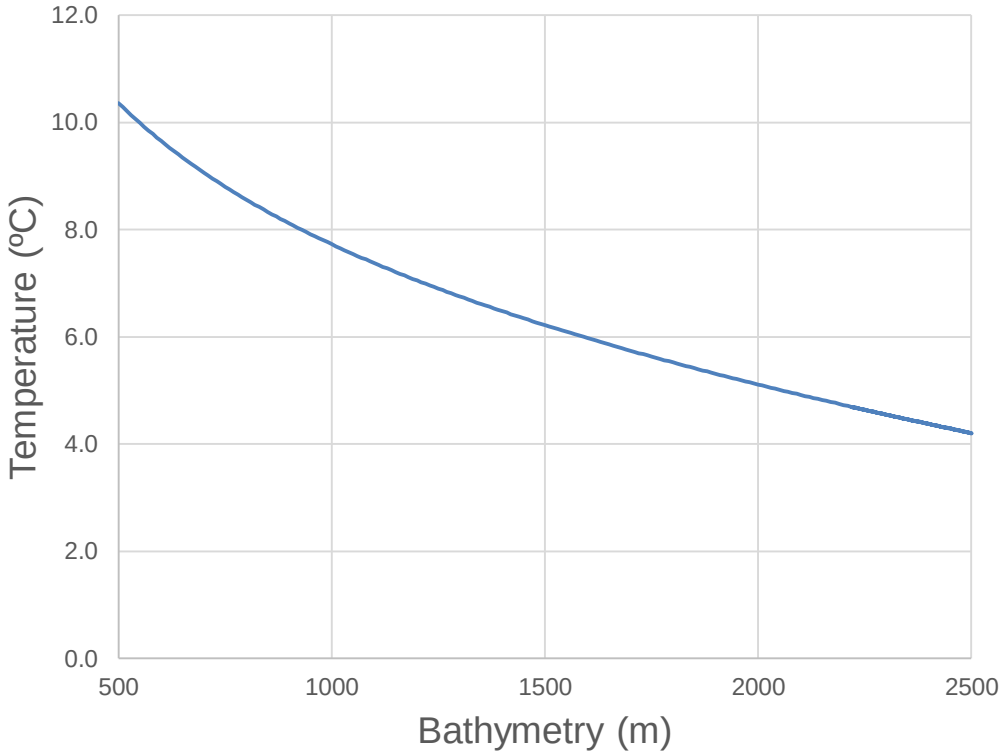


With porosity and temperature corrections

6 – TRANSIENT TEMPERATURE ANALYSIS OF SEDIMENTARY LAYERS WITH A TIME DEPENDENT SURFACE TEMPERATURE

The Problem

- This simulation revisits the previous example by now considering that the surface temperature is a function of the basin bathymetry and that the local bathymetry is rising with a constant rate of 200 m / Myr (a rather large value used for example purposes only).



Reference: The temperature as a function of the bathymetry was taken from **“**Crustal Heat Flow”, Beardsmore and Cull.

Key Example Points

- The model structure for this simulation is similar to the previous ones. The main features not seen on previous models are:
  - How to specify a boundary condition that changes over time.
  - How to setup a non-linear transient analysis using the linear FEM process.

- This example builds heavily on the previous ones and only shows key difference points. Please refer to the simulation files for the complete source.

Simulation file: BasinLayers3.lua

Model file: Surface temperature boundary condition

```
-- Returns the Surface-Water temperature following the model given by
-- Beardsmore & Cull in "Crustal Heat Flow", equations 3.19 to 3.21.
-- Parameters are the bathymetry (in m) and the Latitude (in degrees).
-- This models predicts temperatures that are too high for shallow waters,
-- so if the calculated value is greater than a known surface temperature
-- Ts, that value is used instead.
local function SWT(z, L, Ts)
  if z < 20.0 then
    return Ts
  end

  local A =  4.63 + 8.84e-4*L - 7.24e-4*L*L
  local B = -0.32 + 1.04e-4*L + 7.08e-5*L*L
  local lnTsf = A + B * math.log(z)
  local Tf = -1.90 - 7.64e-4*z
  local BWT = math.exp(lnTsf) + Tf

  return math.min(Ts, BWT)
end

local Latitude = 22
local SurfaceTemperature = 20
```

Constants with the basin latitude and surface temperature

… continues on the next slide

Model file: Surface temperature boundary condition

```
-- Returns the Surface-Water border temperature as a function of the bathymetry
-- and time. The bathymetry is increasing at a constant rate of 200 m/Myr.
NodeFunction {
  id = 'swt_bat_t',
  parameters = { {src = 'coordinate', dim = 2, unit = 'm'}, -- The depth coordinate
                 {src = 'time', unit = 'Myr'},              -- The simulation time
               },
  method = function(z, t)
      z = z + 200 * t
      return SWT(z, Latitude, SurfaceTemperature)
  end
}

-- Surface temperature boundary condition
BoundaryCondition {
  id    = 'Surface-Water border temperature',
  type = 'node temperature',
  mesh = 'mesh',

  properties  = {
    {id = 'T',  description = 'External temperature applied on the node', unit = 'degC', functions = true},
  },
  nodeValues = {
    {'basinTop',  'swt_bat_t'},
  }
}
```

The bathymetry is increasing at a rate of 200 m / Myr

Solution file: Orchestration

The orchestration starts by executing a steady state analysis to initialize the basin temperature at t = 0.0. It then enters the simulation time loop. Like the previous example, at each time step, the non-linearity due to k(T) is solved by repeated linear iterations.

```
function ProcessScript()
  -- Transient simulation parameters
  local time      = 0.0
  local dt        = 0.05 -- In Myr. Our time step will be of 50.000 years
  local finalTime = 20.0 -- The simulation will run for 20 million years

  -- Non-linear loop parameters
  local maxIter = 10    -- The maximum number of allowed nonlinear iterations
  local tol     = 1e-5  -- The convergence tolerance
  local r, iter

  setCurrentTimeUnit('Myr')

  -- Before running the simulation, we must init mesh porosity values
  initPorosity()
```

Tells GeMA that values passed to setCurrentTime() and time intervals passed to the transient step function are given in million years

… continues on the next slide

Solution file: Orchestration

```
-- First, lets run a steady state simulation to initialize the basin temperature at t = 0
print(string.format('Calculating initial solution at t = %.2f Myr', 0.0))

local steadySolver = fem.initLinearSolver({'HeatPhysics'}, 'NumSolver')
r    = 1.0
iter = 1
while r > tol and iter <= maxIter do
  print('  Executing linear step '..iter)

  -- Execute a linear step
  fem.linearStep(steadySolver)

  -- Evaluate error
  r = fem.linearResidual(steadySolver)
  print(string.format('    r = %.6f', r))

  iter = iter + 1
end
assert(r <= tol, 'Failed to achieve initial convergence...')

-- Init result file with the solution at t = 0
local file = io.prepareMeshFile('mesh', '$SIMULATIONDIR/out/$SIMULATIONNAME.nf', 'nf',
                                {'T', 'burialDepth'}, {'dip(degree)', 'phi(%)', 'k'},
                                {scaleFactorY = -1.0, materialPropertySet = 'LithoPSet'})
io.addResultToMeshFile(file, 0.0)
```

Same logic as the steady state orchestration used on the previous example.

… continues on the next slide

Solution file: Orchestration

```
-- Init transient solver
local transientSolver = fem.initTransientSolver({'HeatPhysics'}, 'NumSolver', nil, true)

-- Simulation loop. Will be run for 25 million years
while time < finalTime do
  time = time + dt
  setCurrentTime(time)
  -- Since the conductivity is a function of T, the model is non-linear.
  -- It could be solved by the non-linear solver, but to show the orchestration
  -- flexibility, we will solve it by repeated linear iterations.
  r    = 1.0
  iter = 1
  while r > tol and iter <= maxIter do
    -- Execute a linear step
    fem.transientLinearStep(transientSolver, dt, iter)

    -- Evaluate error
    r = fem.transientLinearResidual(transientSolver, dt)
    print(string.format('    r = %.6f', r))

    iter = iter + 1
  end
  assert(r <= tol, 'Failed to achieve convergence...')

  -- Save results
  io.addResultToMeshFile(file, time)
end

-- Close result file
io.closeMeshFile(file)
end
```

This will be a non-linear simulation

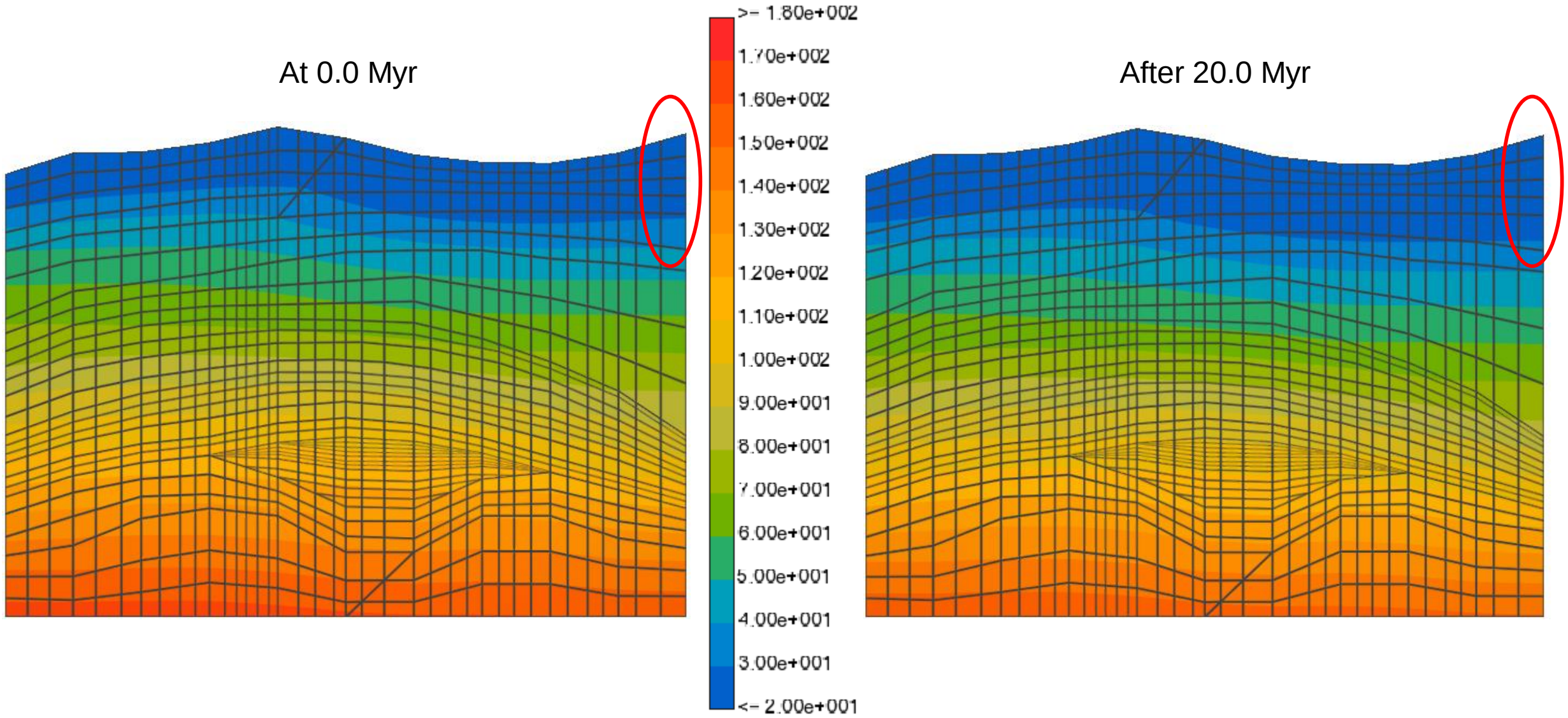
Simulation (time) loop

Set current simulation time in Myr

Non-linear loop

Time step in Myr

Results: Temperature distribution (ºC)



7 – TRANSIENT TEMPERATURE ANALYSIS OF SEDIMENTARY LAYERS WITH A TIME DEPENDENT SURFACE TEMPERATURE WITH PARALLEL POROSITY INITIALIZATION

The Problem / Key Example Points

- This simulation revisits the previous example by initializing the porosity values by traversing mesh cells in parallel

- The model structure for this simulation is similar to the previous ones. The main features not seen on previous models are:
  - How to transform a serial loop traversing mesh cells (or nodes) into a parallel loop

- This example builds heavily on the previous ones and only shows key difference points. Please refer to the simulation files for the complete source.

Simulation file: BasinLayers4.lua

Solution file: Orchestration

The needed changes are minimum. Below we see the initPorosity() code for both the serial and parallel versions, side by side to highlight the differences.

Serial version:

```
function initPorosity()
  ...
  for i=1, m:numCells() do
    local e = m:cell(i)
    ...
  end
end


function ProcessScript()
  ...
  initPorosity()
  ...
end
```

Parallel version:

```
SharedCodeBegin{}

function initPorosity(taskCells)
  ...
  for e in taskCells do
    ...
  end
end

SharedCodeEnd{}

function ProcessScript()
  ...
  cellParallelCall('mesh', true, 'initPorosity')
  ...
end
```

Tells GeMA that functions inside the block defined by the pair SharedCodeBegin / End are meant to be called in parallel environments

Iterator object defining the subset of mesh cells that should be processed by this function call

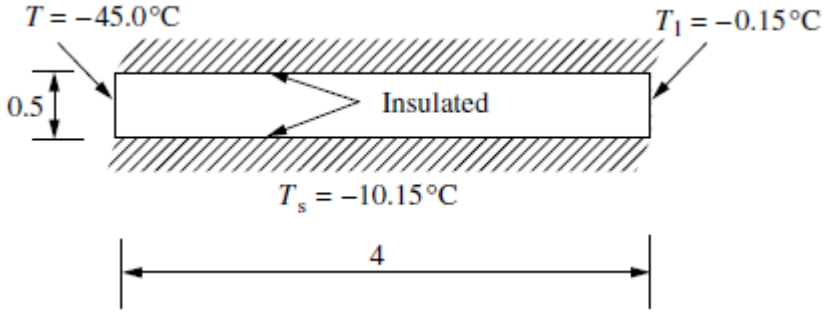
A string with the function name, not the function itself.

Parallelizes a loop over mesh cells by calling the initPorosity() function from multiple threads, each receiving as parameter a subset of the mesh cells.

8 – PHASE CHANGE BY THE EFFECTIVE HEAT CAPACITY METHOD

The Problem

- This example simulates a solidification problem where a material with liquidus temperature of -0.15 ºC and solidus temperature of -10.15ºC, initially at 0.0 ºC, is subjected to a prescribed temperature on its left side of -45 ºC and -0.15 ºC at the right side.



L = 70.26 (Latent heat of solidification)
k = 1.0
rho = 1.0
cp = 1.0

- The phase change is modeled with the effective heat capacity method:

$$c_{eff} = \begin{cases} c_s & \text{if } T \leqslant T_s \\ c_f + \dfrac{L}{T_l - T_s} & \text{if } T_s < T < T_l \\ c_l & \text{if } T \geqslant T_l \end{cases}$$

Reference: “Fundamentals of the Finite Element Method for Heat and Fluid Flow”, Lewis at al.

Example 6.7.1

Key Example Points

- The model structure for this simulation is similar to the previous ones. The main features not seen on previous models are:
  - How to setup a non-linear transient analysis using the non-linear FEM process
  - Track results at a given node during the simulation

- This example builds heavily on the previous ones and only shows key difference points. Please refer to the simulation files for the complete source.

Simulation file: Solidification.lua

Model file: Effective heat capacity

```
-- User function implementing the concept of effective heat capacity
CellFunction
{
  id = 'ceff',
  parameters = { {src = 'T',  unit = 'degC'},
                 {src = 'Ts', unit = 'degC'},
                 {src = 'Tl', unit = 'degC'},
                 {src = 'cs', unit = 'J/(kg.degC)'},
                 {src = 'cl', unit = 'J/(kg.degC)'},
                 {src = 'cf', unit = 'J/(kg.degC)'},
                 {src = 'L',  unit = 'J/kg'},
               },
  method = function(T, Ts, Tl, cs, cl, cf, L)
      if T <= Ts then
        return cs
      elseif T >= Tl then
        return cl
      else
        return cf + L / (Tl - Ts)
      end
  end
}
```

➡ The interpolated temperature

➡ Material properties with the solidus and liquidus temperatures, latent heat and specific heat for the solid, liquid an “freezing” states

Solution file: Orchestration

```
-- The configuration options for the non-linear solver
local solverOptions = {
  type                = 'transient automatic time step',
  timeMax             = 5.0,    -- The total simulation time
  timeInitIncrement   = 0.05,   -- Time increment
  timeMinIncrement    = 0.01,
  timeMaxIncrement    = 1.0,
  iterationsMax       = 15,
  tolerance           = {temperature = 1.000E-05},
}
```

This set of options controls the behavior of the non-linear FEM solver. See the FEM process documentation for an explanation of each available option.

The group name for the simulation state variable (see the state var definition on the model file). Separating state variables in named groups allows for specifying different tolerances for each kind of value in multi-physics simulations.

```
function ProcessScript()

  -- For comparison with literature results, we want to keep track of the temperature at x = 1.0
  -- This can usually be done by a post-processor but in this example we show how this can also
  -- be done easily in GeMA.
  local resultAt26 = {}              -- A table to store results at node 26 (x = 1.0)
  local m = modelData:mesh('mesh')   -- Get the mesh object
  local T = m:nodeValueAccessor('T') -- And an accessor to query temperature values

  -- Creates the output simulation file
  local file = io.prepareMeshFile('mesh', '$SIMULATIONDIR/out/$SIMULATIONNAME.nf', 'nf', {'T'})
```

… continues on the next slide

Solution file: Orchestration

```
-- Load VtkLib to enable saving results in the Vtk format
dofile('$SCRIPTS/vtkLib.lua')

-- Creates the non-linear FEM solver used for calculations
local solver = fem.init({'HeatPhysics'}, 'NumSolver', solverOptions)

local dt     = solverOptions.timeInitIncrement
local endt   = solverOptions.timeMax
local nsteps = endt / dt

-- Time loop
for i=1, nsteps do
  print('------------------')
  print(string.format('Temperature iteration - time = %.2f s', i*dt))
  print('------------------')

  -- Advances the simulation by dt. The solver returns a suggested next time step 'newt'
  -- For this example it will be ignored to ease the comparison with results from the literature
  local newt = fem.step(solver, dt)

  -- Save result using neutral file and vtk formats
  io.addResultToMeshFile(file, i*dt)
  vtkLib.saveMeshFile('mesh', {'T'}, nil,
                      '$SIMULATIONDIR/out/$SIMULATIONNAME', {state = i, stateTime = i * dt})
```

Solver options from the previous declared table.

Number of simulation steps

… continues on the next slide

Solution file: Orchestration

```
    -- Store current time and the calculated temperature at x = 1.0 to the table resultAt26
    -- Each table entry will be a subtable with the current time and temperature
    resultAt26[i] = {i*dt, T:value(26)}
  end

  -- Closes the output file
  io.closeMeshFile(file)

  -- Print stored results
  print()
  print('Temperature results at x = 1.0')
  print('Time(s)    T(degC)')
  print('-------    --------')
  for i=1, nsteps do
    print(string.format('%5.2f      %8.4f', resultAt26[i][1], resultAt26[i][2]))
  end

end
```

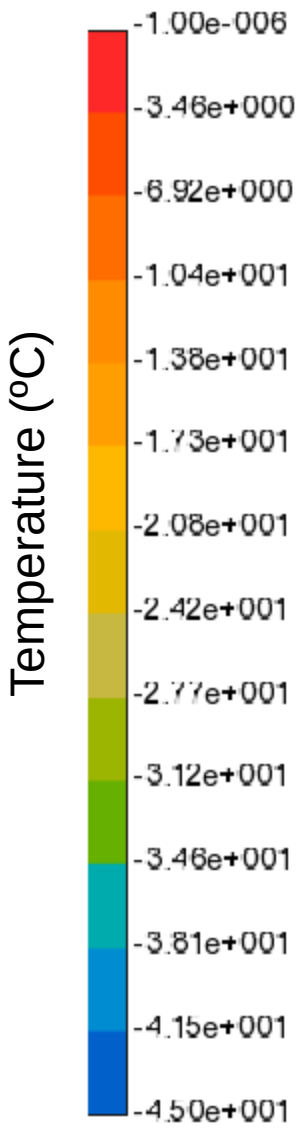
Get the current temperature value from node 26

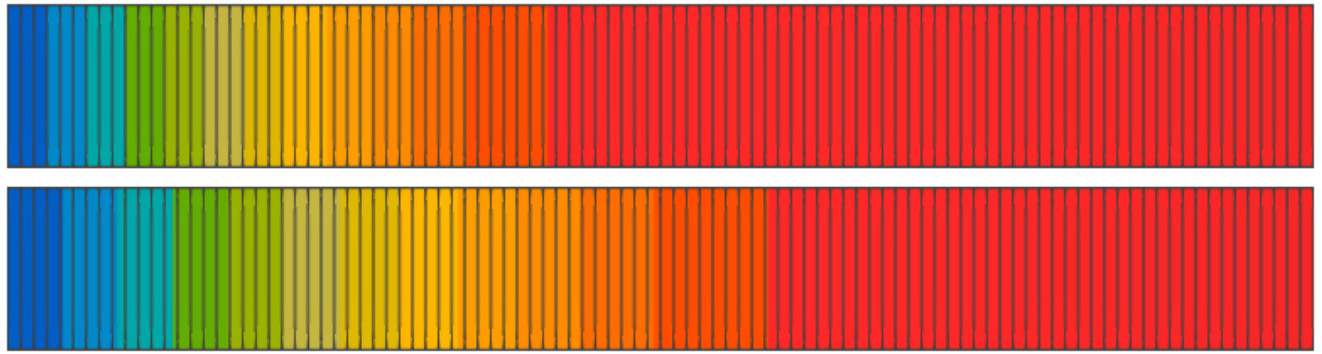
The sub-table storing results saved at the ‘ith’ time step

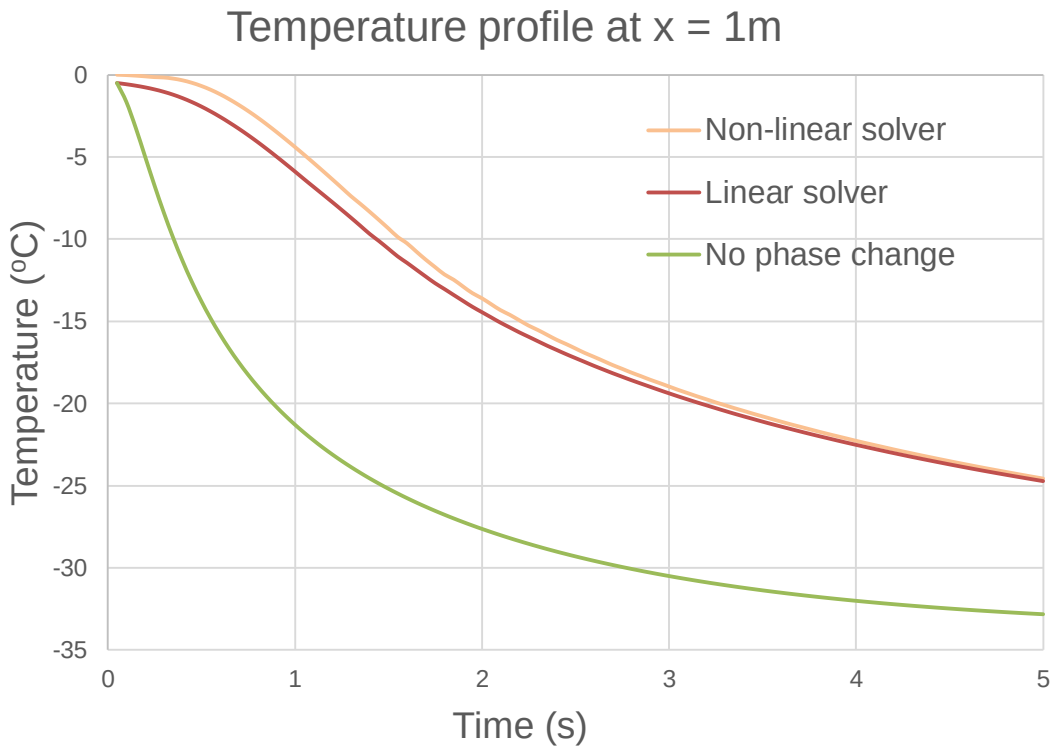
The stored time for the ‘ith’ time step

The stored temperature for the ‘ith’ time step

Results


Temperature (ºC)
-1.00e-006
-3.46e+000
-6.92e+000
-1.04e+001
-1.38e+001
-1.73e+001
-2.08e+001
-2.42e+001
-2.77e+001
-3.12e+001
-3.46e+001
-3.81e+001
-4.15e+001
-4.50e+001


t = 2.5 s
t = 5 s


Temperature profile at x = 1m
Temperature (ºC)
0
-5
-10
-15
-20
-25
-30
-35
0
1
2
3
4
5
Time (s)
Non-linear solver
Linear solver
No phase change