Steady State and Transient Thermal Simulation of a 2D Heatpipe-based Cooler for Mobile Processors

Michael Laufer

January 29, 2018

Abstract

This is the abstract.

Introduction

Electronics cooling technologies have Versteeg and Malalasekera [1995]
one more

Project Motivation

blah blha

Numerical Method

Steady State

The governing equation in our steady state heat conduction problem is the 2d heat equation, essentially the Poisson equation with a diffusion constant (in this case the thermal conductivity).

$$\nabla \cdot (k \nabla T) = S_T$$

As per the Finite Volume Method, the equation is integrated over the volume of each cell, whereby with the use of the Gauss divergence theorem, the resulting equation is seen in equation 7.13 and 7.14 in the class textbook Mazumder [2015].

$$\int_{V_0} \nabla \cdot (k \nabla T) dV = \int_S (k \nabla T) \cdot \hat{n} dA$$

$$\int_S (k\nabla T)\cdot\hat{n}dA = \sum_{f=1} {}_{N_{f,O}} \left[(k\nabla T)_f \cdot \hat{n}\right] A_f$$

[We note here that all equations have been modified to relate to heat conduction, i.e temperature, T as the dependant variable, as opposed to the generic ϕ).]
In 2D cartisian coordinates the value of the flux at the cell faces is straightforward. The book goes into great detail showing how the flux can be evaluated using a coordinate system with basis vectors in the face normal, and face tangential direction. From equation 7.50 in the book:

$$(\nabla T)\cdot\hat{nf} = \frac{(\nabla T)_f \cdot I_f}{\delta_f} - \frac{\left[(\nabla T)_f \cdot \hat{t}_f\right]\hat{t}_f \cdot I_f}{\delta_f}$$

The face tangent directional component of the flux, is evaluated using the values of T at the vertex locations, but as only cell center values are known, an expression for the vertex values as a function of cell center values must be used. This leads to the expression given in equation 7.58:

$$(\nabla T)\cdot\hat{nf} = \sum_{f=1}^{N_{f,O}} k_f \left(\frac{T_{N_f} - T_O}{\delta_f} - \left[\frac{T_{a,f} - T_{b,f}}{\delta_f A_f}\right]\hat{t_f}\cdot I_f\right) A_f$$

In general two variations of the scheme can be formulated. One, an implicit approach, whereby substituting expressions for the vertex values given in equation 7.36 in the book, leads to an equation containing only cell center values. A second approach, one used in this work, is to treat the vertex values explicitly, leading to an iterative approach, where the vertex values from the previous iteration are used. The explicit approach is significantly easier to implement, as the only unknowns are adjacent cell center values. On the other hand, in the implicit approach, a less sparse coefficient matrix will be formed, owing to the fact that the stencil now extends beyond the immediate neighbors. The explicit approach is more commonly used than the implicit approach, and it is the one used in this work.

The implementation of the scheme for the steady state case is as follows:

1. Interpolate vertex values from cell center values.

2. Loop over every cell face, add respective link value contribution to the coefficient matrix and RHS vector for every internal face (not boundary node)

3. Loop over each face requiring special boundary treatment, and add the contribution to the coefficient matrix and RHS vector.

4. Solve the sparse system

5. Compute norm of solution, and check for convergence. If not converged, return to step 1.

This is implemented in the "main" of the python code in the following easy-to-follow code snippet (full code available in Appendix B):

```
if __name__ == '__main__':
    converged = False
    iteration = 0
    norm = -1
    max_iter = 50
    t0 = time.time()
    while not converged:
        iteration += 1
        A = lil_matrix((nelements, nelements))
        b[:]   = 0.0
        # interpolate to vertices
        phi_vert = vertices_interpolate(phi_vert, phi)
        # assemble the coefficient and rhs matrix for allfaces
        A, b = compute_interior_faces(A, b)
        b = compute_neumann_faces(b)
        b = compute_adiabatic_faces(b)
        A, b = compute_robin_faces(A, b)
        phi_new = spsolve(A.tocsc(), b)
        norm = np.linalg.norm(phi-phi_new)
        if (norm < tolerance or iteration >= max_iter):
            print('Converged in %d iterations'%iteration)
            converged = True
            break
        print('Iteration %d, Error = %g'%(iteration, norm))
        phi[:] = phi_new[:]
```

Boundary Conditions

The heatpipe based cooler that we are trying to model has 3 boundary conditions:

- Heat flux from the electronic component on the bottom surface of the cooler
- Adiabatic surfaces on the sides of the cooler
- Convective heat transfer from the fin surfaces

1. Heat Flux Boundary Condition The heat flux on the bottom of the cooler is essentially a Neumann boundary condition. The contribution of the corresponding face to the RHS vector is just the heat flux value itself, $J_B.

$$(\nabla T_B) \cdot \hat{n}_B = J_B$$

   This is implemented in the following function that returns the new RHS vector:

```
def compute_neumann_faces(b):
    # assemble coefficient & rhs for all Neumann faces
    for tmp_face_index, face_index in enumerate(neumann_faces):
        elem_index = f2e[face_index, 0]
        ds         = face_areas[face_index]
        b[elem_index] += heat_flux*ds
    return(b)
```

2. Adiabatic Boundary Condition The adiabatic boundary condition can be thought of as a Neumann boundary condition where the spatial derivative is zero. Or:

$$(\nabla T_B) \cdot \hat{n}_B = J_B = 0$$

   The implementation is thus rudimentary, as the contribution of the adiabatic face to the RHS vector is simply zero for each respective adiabatic face.

3. Convective Heat Transfer Boundary Condition A convective heat transfer boundary condition relates the heat flux though a surface to the temperature difference of the surface and the ambient fluid temperature.

$$\dot{Q} = hA_f(T_S - T_\infty)$$

   We can recognize that this is a form of a Robin boundary condition. The text book gives a generic formula for the implementation of Robin boundary conditions in Equations 7.90:

$$\alpha(\nabla T_B) \cdot \hat{n_B} + \beta T_B = \gamma$$

and the flux in equation 7.92:

$$(\nabla T) \cdot \hat{nf} = \left(\frac{\gamma}{\beta} - T_O - [(\nabla T)_B \cdot \hat{t}_B] \hat{t}_B \cdot I_B \right) / \left(\delta_B + \frac{\alpha}{\beta} \right)$$

Using the above equation we ascertain:

$$\alpha = k$$

$$\beta = h$$

$$\gamma = -hT_\infty$$

The implementation of this boundary condition is seen in the following code snippet:

```
def compute_robin_faces(A, b):
    # assemble coefficient & rhs for all Robin faces
    for tmp_face_index, face_index in enumerate(robin_faces):
        elem_index      = f2e[face_index, 0]
        elem_face_index = -1
        for tmp_index in range(elem_nfaces):
            if e2f[elem_index, tmp_index] == face_index:
                elem_face_index = tmp_index
                break
        deltaf = elinks[elem_index, elem_face_index, 0]
        tdotI  = elinks[elem_index, elem_face_index, 1]
        ds     = face_areas[face_index]
        node1, node2 = xfaces[face_index]
        b[elem_index] -= ds*k_faces[face_index]*(-gamma_div_beta/(deltaf + alpha
        A[elem_index, elem_index] += k_faces[face_index]*ds/(deltaf + alpha_div_
    return(A, b)
```

Material Properties

The final important element in the numerical implementation of a heatpipe based cooler is dealing with the discontinuous material properties that are inherent in the problem. The heatpipe is a closed rod with a fluid that evaporates and turns to a vapor at the hot end, and due to density differences, moves to the cold end, where it condenses back down to a liquid state. This process allows the heatpipe to be a very efficient way of transporting large amounts heat over relatively long distances. This is often modelled as a

material with an extremely high thermal conductivity, reaching values up to 100 times the thermal conductivity of copper. In this work, the thermal conductivity was chosen to be $K_{heatpipe} = 10000$. This leads to a discontinuous thermal conductivity, which if implemented naively leads to inaccuracies during the cell center value to face value interpolation process. One way to deal with this discontinuity is to use a an inverse distance-weighted interpolation for the thermal conductivty. This can be seen in equation 7.32b:

$$k_f = \frac{k_1 k_2}{\frac{k_1\delta_2 + k_2\delta_1}{\delta_1+\delta_2}}$$

To implement this 2 additional pre-proccessing steps are needed. First a new array called $K_{elements}$ is created that contains the values of the thermal conductivity for each cell volume. The values are determined by the location of the centroids of each cell.

```
print('{:<30}'.format('Setting Cell Diffusion Coefficients...'))
k_elements = k_nom*np.ones(nelements)
for elem_index in range(nelements):
    #checks if cell centroid is inside of heatpipe
    if msh.cents[elem_index][1] > 0.005 and msh.cents[elem_index][1] < 0.03:
            k_elements[elem_index] = 10000.0
```

Next using geometrical and mesh generation data, the facial values can be computed using the above formulation

```
print('{:<30}'.format('Computing Face Diffusion Coefficients...\n'))
k_faces = 400*np.ones(msh.nfaces)
for face_index in range(msh.nfaces):
    if face_index not in msh.bfaces:
        element_index_1 = msh.f2e[face_index,0]
        k_element_1 =  k_elements[element_index_1]
        for tmp_index in range(msh.nelements):
            if msh.e2f[element_index_1, tmp_index] == face_index:
                face_index_1 = tmp_index
                break
        dist_element_1 = msh.elinks[element_index_1, face_index_1, 0]
        element_index_2 = msh.f2e[face_index,1]
        k_element_2 =  k_elements[element_index_2]
        for tmp_index in range(msh.nelements):
            if msh.e2f[element_index_2, tmp_index] == face_index:
                 face_index_2 = tmp_index
```

```
            break
        dist_element_2 = msh.elinks[element_index_2, face_index_2, 0]
        k_faces[face_index] = (k_element_1*k_element_2)/((k_element_1*dist_element_2
```

Unsteady

Unsteady heat conduction is governed by the following equation:

$$\rho c_p \frac{\partial T}{\partial t} = \nabla \cdot (k \nabla T)$$

The derivation of the unsteady FVM scheme is very similar to the steady case, except that in addition to integrating over each cell volume, both sides of the equation are integrating over a time period.

$$FILLIN$$

A fully implicit approach was chosen to deal with the numerical time integration. Although it is only 1st order accurate in time, the fully implicit method is unconditionally stable, as any time step period may be chosen without stability issues. Implementing an implicit solver is important as the ultimate goal of this work is for it to be used as an engineering tool, moreover it will be just one function call inside of a larger script. The ability to arbitrarily change the time step to match the time step of a global script is paramount. Additionally, the accuracy penalty of this approach compared to a second order Crank-Nicholson scheme is not important, as the future engineering tool is required to be just a rough estimate of a consumer cooler.

Grid Generation

blah blah Versteeg and Malalasekera [1995]

Steady State Results

blah blah

Transient Results

blah blah

Summary and Conclusions

blah blah

Appendix A - GMSH grid generation script

Appendix B - Steady State Code

blah blah

Appendix C - Unsteady Code

blah blah

References

S. Mazumder. *Numerical Methods for Partial Differential Equations: Finite Difference and Finite Volume Methods*. Elsevier Science, 2015. ISBN 9780128035047. URL https://books.google.co.il/books?id=YVC2BgAAQBAJ.

H. Versteeg and W. Malalasekera. *An Introduction to Computational Fluid Dynamics: The Finite Volume Approach*. Longman Scientific & Technical, 1995. ISBN 9780582218840. URL https://books.google.co.il/books?id=7UynjgEACAAJ.