Notes on Finite Volume Method MME 9710

Max Le

January 27, 2022

Contents

1 INTRODUCTION

1.1 Generic Conservation Equation

Consider the following generic conservation equation:

$$\frac{\partial \phi}{\partial t} + \nabla \cdot (\mathbf{u}\phi) + \nabla \cdot \mathbf{J}_\phi = S_\phi \tag{1}$$

where the variables are defined as:

Variable	Description
ϕ	Generic variable
t	Time
$\mathbf{u}$	Velocity vector
$\mathbf{J}_\phi$	Diffusive flux of ϕ
S_ϕ	Volumetric source/sink of ϕ

1.2 Mass Conservation Equation

Setting $\phi = \rho$, where ρ is the density. Also, mass conservation of a `continuous` substance does not have diffusive flux => $\mathbf{J}_\rho = 0$.

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\mathbf{u}\rho) = S_\rho \tag{2}$$

Notes:

- For incompressible, constant density flow:
  - $\frac{\partial \rho}{\partial t} = 0$
  - $\nabla \cdot (\mathbf{u}\rho) = \rho \nabla \cdot \mathbf{u}$
  - Result in: $\nabla \cdot \mathbf{u} = \frac{S_\rho}{\rho}$
  - And if no source/sink => $\nabla \cdot \mathbf{u} = 0$

1.3 Momentum Conservation Equation

Setting $\phi = \rho\mathbf{u}$. Diffusive flux term $\mathbf{J}_\mathbf{u} = -\nabla \cdot \sigma$, where σ is the fluid stress tensor.

$$\frac{\partial (\rho\mathbf{u})}{\partial t} + \nabla \cdot (\rho\mathbf{u}\mathbf{u}) = \nabla \cdot \sigma + S_\mathbf{u} \tag{3}$$

The stress tensor, σ can be expressed in terms of pressure (p) and viscous stress tensor (τ) and identity matrix, I:

$$\sigma = -p\mathbf{I} + \tau \tag{4}$$

After substituting, we get the following form of the momentum conservation equation:

$$\frac{\partial (\rho\mathbf{u})}{\partial t} + \nabla \cdot (\rho\mathbf{u}\mathbf{u}) = -\nabla p + \nabla \cdot \tau + S_\mathbf{u} \tag{5}$$

For incompressible Newtonian fluid, we can rewrite τ in terms of the dynamic viscosity, μ $\tau = \mu(\nabla\mathbf{u} + \nabla\mathbf{u}^T)$. Thus, a momentum conservation equation for incompressible, Newtonian fluid, constant velocity:

$$\frac{\partial (\rho\mathbf{u})}{\partial t} + \nabla \cdot (\rho\mathbf{u}\mathbf{u}) = -\nabla p + \mu\nabla^2\mathbf{u} + S_\mathbf{u} \tag{6}$$

1.4 Energy Conservation Equation

Setting $\phi = \rho h$ with h being the specific enthalpy of a substance at a given state. Thus, the unit for ϕ is energy per unit volume. The diffusive flux is given by Fourier's law: $J = -k\nabla\mathbf{T}$ with k is the thermal conductivity.

$$\frac{\partial(\rho h)}{\partial t} + \nabla \cdot (\rho \mathbf{u} h) = \nabla \cdot (k \nabla \mathbf{T}) + S_h \tag{7}$$

If we assume:

- incompressible flow
- constant specific heat capacity, $h = c_p\mathbf{T}$
- constant thermophysical properties (k and ρ)
- no source term

$$\frac{\partial(\mathbf{T})}{\partial t} + \nabla \cdot (\mathbf{u}\ \mathbf{T}) = \alpha \nabla^2 \mathbf{T} \tag{8}$$

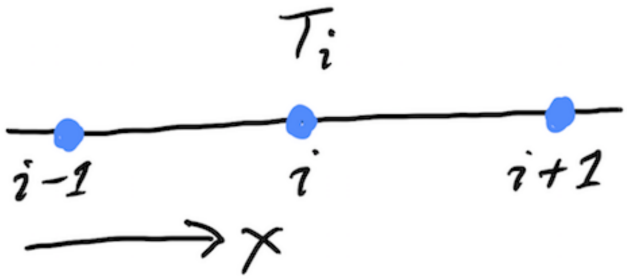
where $\alpha = \frac{k}{\rho c_p}$ is the thermal diffusivity.

1.5 Discretization of the Generic Conservation Equation

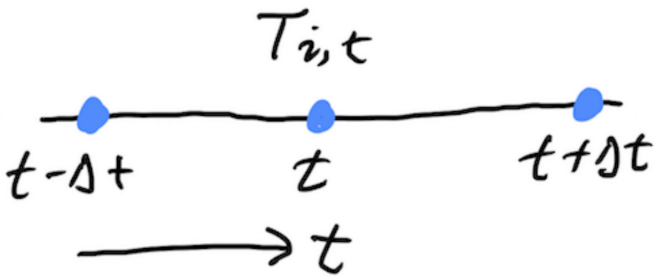
Our generic variable, ϕ is a function of of spatial and time: $\phi = \phi(\mathbf{x}, t)$, where $\mathbf{x} = (x, y, z)$. Note that spatial variable can be influenced "$_{\text{one}}$ way_" or "$_{\text{two}}$ way_", i.e.

- One way: changes in ϕ only occur due to change on one side of that location
- Two way: changes in ϕ occur due to changes on both side of that location.

For example, heat conduction in the image below at cell i is influenced by cell $i-1$ and $i+1$. Here, **x** is a two way coordinate for heat conduction



Now consider transient heat convection/conduction. The temperature at any given time is influenced by existing conditions before that point **in time**. Here, **t** is a one way coordinate for transient heat conduction/convection



Recall our generic conservation equation:

$$\frac{\partial \phi}{\partial t} + \nabla \cdot (\mathbf{u}\phi) + \nabla \cdot \mathbf{J}_\phi = S_\phi$$

We consider the diffusion term or **elliptic PDE** : $\boxed{\nabla \cdot \mathbf{J}_\phi}$ to be two-way in space

Likewise, the convection term or **parabolic PDE** : $\boxed{\nabla \cdot (\mathbf{u}\phi)}$ to be one-way in space

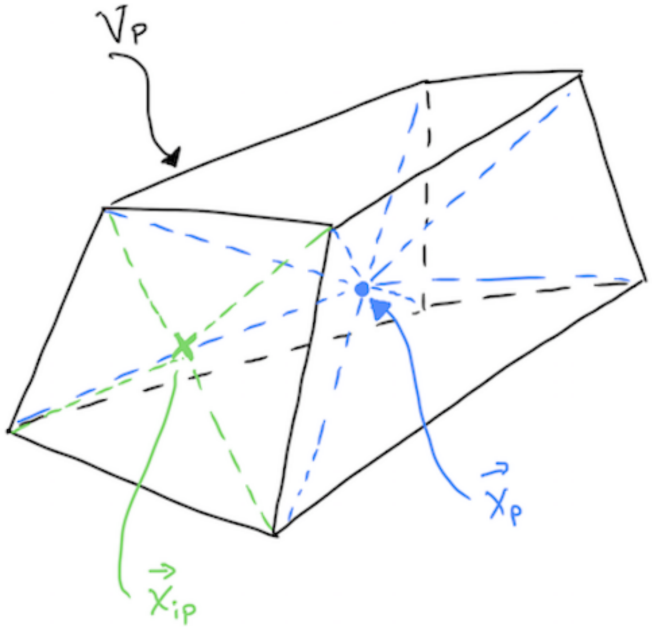
1.6 Main idea behind Discretization

Our goal is to:

- replace the PDEs' continuous solution with discrete solution, at specific location that approximates the continuous solution suitably.

For finite volume:

1. domain is split into non overlapping finite regions that fill the domain
2. the discrete point is at the centroid of each control volume with volume V_p, at position $\mathbf{x}_p$
3. surround these cells, we have the "faces". At the center of these "faces", we have the integration point at position $\mathbf{x}_{ip}$
4. the governing equations are then integrated over a control volume, where surface flux terms and volume source terms are balanced.



1.7 Determine Cell Centre + Face Integration Points

Cell centre => location of solution variables.
Points on face => fluxes are evaluated.
Consider a volume integral of a quantity ϕ, we may express this integral in discrete form as follow:

$$\int_V \phi dV \approx \phi_P V_P \tag{9}$$

where ϕ_P is the value of ϕ at some internal within V and V_P is total volume of the cell:

$$V_P = \int_V dV \tag{10}$$

To prove the above result, we expand ϕ in a Taylor series about the point P.

$$\phi \approx \phi_P + \nabla\phi_P(\mathbf{x} - \mathbf{x}_P) + \nabla^2\phi_P(\mathbf{x} - \mathbf{x}_P)(\mathbf{x} - \mathbf{x}_P) + ...O(\delta^3) \tag{11}$$

with δ being the characteristic grid spacing. Substitute this into our assumed expression for V_P:

$$\int_V \phi dV \approx \int_V [\phi_P + \nabla\phi_P(\mathbf{x} - \mathbf{x}_P) + \nabla^2\phi_P(\mathbf{x} - \mathbf{x}_P)(\mathbf{x} - \mathbf{x}_P) + ...O(\delta^3)]dV \tag{12}$$

We note that ϕ_P and its derivatives are constants:

$$\int_V \phi dV \approx \phi_P dV + \nabla\phi_P \int_V (\mathbf{x} - \mathbf{x}_P)dV + \nabla^2\phi_P \int_V (\mathbf{x} - \mathbf{x}_P)(\mathbf{x} - \mathbf{x}_P)dV + ...O(\delta^3) \tag{13}$$

Because our $\mathbf{x}_P$ point is at centroid, so $\int_V (\mathbf{x} - \mathbf{x}_P)dV = 0$. Likewise, the last term is also neglected, resulting in:

$$\int_V \phi dV \approx [\phi_V + O(\delta^2)]V_P \tag{14}$$

This means that there is a second order error when approximating the cell volume in this way. This is OK because the accuracy of the method is also second order.
Note: If our $\mathbf{x}_P$} does not lie at the centroid of the cell. The second term,$\int_V (\mathbf{x} - \mathbf{x}_P)dV$ does not go to zero, making our approximation to be 1st order, which is worse.

1.8 Transient term

Here we deal with the transient term, $\frac{\partial\phi}{t}$. Discretization of this term relies on:

- order of accuracy
- implicit vs explicit

The idea is to integrate this term over control volume V_P and some time step $\Delta t = t_1 - t_0$ to get the formula for the discretization.

$$\int_{t_0}^{t_1} \int_V \frac{\partial\phi}{\partial t} dV dt \approx (\phi V_P)^{t_1} - (\phi V_P)^{t_0} \tag{15}$$

1.9 Advection term

Here we deal with the advection term, $\nabla \cdot (\mathbf{u}\phi)$. Similar to the transient term, the formula for the discretization can be obtained by integrating over the control volume V_P. We also employ Gauss' theorem to convert volume integral to surface integral:

$$\int_V \nabla \cdot (\mathbf{u}\phi) dV = \int_S (\mathbf{u}\phi) \cdot \mathbf{n} dS \tag{16}$$

For the surface integral, we approximate by summing up over the faces surrounding the cell, each with area A_{ip}.

$$\int_S (\mathbf{u}\phi) \cdot \mathbf{n}_{ip} dS \approx \sum_{i=0}^{N_{ip}-1} \mathbf{u}_{ip} \cdot \mathbf{n}_{ip} \phi_{ip} A_{ip} \tag{17}$$

Note:

- using C program notation, so we sum from 0 till $N_{ip} - 1$
- approximate $\mathbf{u}_{ip}$ by many interpolation methods
- interpolating ϕ_{ip} carefully to obtain stable numerical method.

1.10 Diffusion term

Now, we deal with the diffusion term, $\nabla \cdot \mathbf{J}_\phi$. Similar to the advection term, we integrate over a control volume, then apply Gauss' theorem

$$\int_V \nabla \cdot \mathbf{J}_\phi dV = \int_S \mathbf{J}_\phi \cdot \mathbf{n} dS \tag{18}$$

Again, the surface integral is approximated as discrete sum over the faces surrounding the cell:

$$\int_S \mathbf{J}_\phi \cdot \mathbf{n} dS \approx \sum_{i=0}^{N_{ip}-1} \mathbf{J}_{\phi,ip} \cdot \mathbf{n}_{ip} \mathbf{A}_{ip} \tag{19}$$

where the flux, $\mathbf{J}_{\phi,ip}$ is interpolated from neighboring cell values.

1.11 Source term

Recall our source term: S_ϕ, we assume that the source term is piecewise continuous, with one specific value, S_ϕ, being represented by each cell. We can then write:

$$\int_V S_\phi dV \approx S_\phi V_P \tag{20}$$

Generally, the source term may depend on ϕ so linearization is needed to obtain stable numerical method.

1.12 Linearization

With regard to our last point about J_ϕ, the discretized terms depend non linearly on the solution. This non-linearity is caused by:

- source term depend non linearly on primitive variable, e.g. J_ϕ.
- non linearities in the governing equation, e.g. advection term $\nabla \cdot (\mathbf{u}\phi)$
- on non-orthogonal grid, gradient correction terms are needed <= these are non linear.

To linearize, we assume the governing PDE is represented by the following general differential operator

$$L(\phi^*) = 0 \tag{21}$$

where:

- ϕ^* = the continuous solution to the PDE
- Note that to solve a PDE using finite volume, the continuous solution ϕ^* is approximated by the discrete solution vector $\phi \in \mathbb{R}$ on N number of control volume. Our PDE is then integrated over each control volume and each term in the governing equation is approximated using the discrete solution ϕ
- Of course, the numerical solution will not satisfy the discretized equation exactly; rather we will have a residual, $\mathbf{r} \in \mathbb{R}^N$.

We expand the residual about the solution ϕ_i at iteration i, and find the solution where $r = 0$:

$$\mathbf{r}(\phi_i) + \left.\frac{\partial \mathbf{r}}{\partial \phi}\right|_{\phi_i} (\phi - \phi_i) = 0 \tag{22}$$

We define the **Jacobian of the residual vector** as:

$$\mathbf{J}(\phi) = \frac{\partial \mathbf{r}}{\partial \phi} \tag{23}$$

We use this to update according to fix point iteration:

$$\phi = \phi_i + \Delta\phi_i \tag{24}$$

where:

$$\Delta\phi = (\phi - \phi_i) \tag{25}$$

and:

$$\mathbf{J}(\phi_i)\Delta\phi = -\mathbf{r}(\phi_i) \tag{26}$$

The remaining unknowns are: the residual vector **r** and Jacobian matrix $\mathbf{J}(\phi_i)$.
Note: we can express the linear system for a control volume P as:

$$a_P \delta\phi_P + \sum_{nb} a_{nb}\delta\phi_{nb} = -r_P \tag{27}$$

where nb is sum over all neighboring cells. The coefficients are defined as:

$$a_P = \frac{\partial r_P}{\partial \phi_P} \tag{28}$$

$$a_{nb} = \frac{\partial r_P}{\partial \phi_{nb}} \tag{29}$$

1.13 Four Basic Rules

Outlined by Patankar(1980), these 4 rules are:

- **Rule 1: Consistency at control volume faces**
  For common faces between cells, the flux through those common faces must be the same when evaluated at each cell. If this is not the case, then it means there is an artificial source of the energy at the face.

- **Rule 2: $\alpha_p > 0$ and $\alpha_{nb} < 0$**
  Consider situations involving only convection and diffusion and all other conditions unchanged: if ϕ in 1 cell increases, then we can expect ϕ in the neighboring cells to increase as well. The only way that this could happen is in this equation

  $$a_P \delta\phi_P + \sum_{nb} a_{nb} \delta\phi_{nb} = -r_P$$

  a_p must have opposite sign from each of its a_{nb} coefficients, just so that $\delta\phi_p$ and $\delta\phi_{nb}$ have the same signs and r_p is unchanged.

- **Rule 3: Negative slope linearization of source terms**
  Suppose we have a source term in the form: $S_\phi = a + b\phi_P$. If this is moved to the LHS, the coefficient a_P can be negative if b is positive. So we require that $b < 0$, or negative slope linearization. The idea is that a positive slope linearization would be unstable because the source would cause the variables to increase, which would then increase the source term. This would continue indefinitely and without bounds. In terms of heat transfer, we can have a heat source that grows with temperature and also a heat sink for removal of temperature. This is done to avoid an uncontrolled increase in temperature.

- **Rule 4: Sum of neighboring coefficients**
  Our governing equations contain derivatives of dependent variables, i.e. both ϕ and $\phi + c$ will satisfy the same governing equations. Thinking practically, this means that temperature field in both Kelvin and Celcius would both satisfy the same discretized equations, because Celcius and Kelvin scale are related via a constant. For this to be true, we require:

  $$a_p = -\sum_{nb} a_{nb}$$

  Note that in the case of the linearization of the source term above, it indicates that same equation cannot be used for both ϕ and $\phi + c$. So in this case, make sure to modify the source term coefficients appropriately.

2 STEADY DIFFUSION EQUATION

2.1 Problem Definition

We consider the solution of a steady, 1D heat diffusion equation

$$-k\nabla^2 T - S = 0 \tag{30}$$

2.2 Discretization

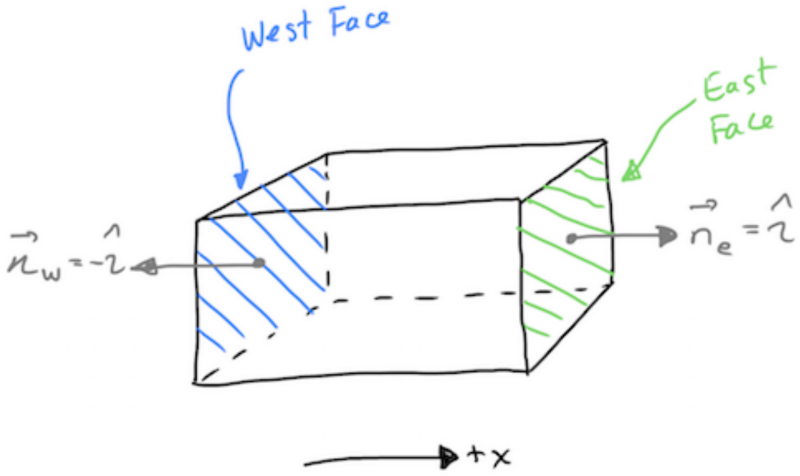
Recall our diffusion term can be discretized as:

$$\int_S \mathbf{J}\cdot\mathbf{n}dS \approx \sum_{i=0}^{N_{ip}-1} \mathbf{J}_{ip}\cdot\mathbf{n}_{ip}A_{ip} \tag{31}$$

Our flux **J** here is the diffusive flux, so: $\mathbf{J} = -k\nabla T$. Thus:

$$\int_S \mathbf{J}\cdot\mathbf{n}dS \approx -\sum_{i=0}^{N_{ip}-1} k_{ip}\nabla T_{ip}\cdot\mathbf{n}_{ip}A_{ip} \tag{32}$$

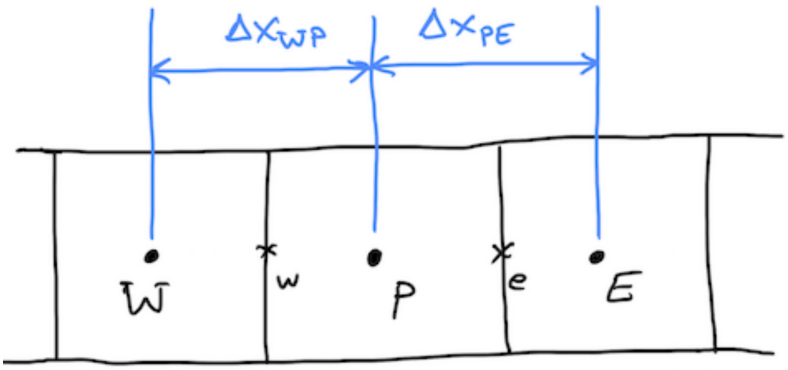
We assume constant thermal conductivity, $k_{ip} = k$. A 1D control volume, with West/East faces and unit vectors drawn, is shown below:



Since we are in 1D, our unit vector is in the **i** only.
Thus, $\nabla T\cdot\mathbf{n} = \nabla T\cdot\mathbf{i}$.
But, $\nabla T\cdot\mathbf{i} = \left\langle \frac{\partial T}{\partial x}\mathbf{i} + \frac{\partial T}{\partial y}\mathbf{j} + \frac{\partial T}{\partial z}\mathbf{k}\right\rangle \cdot \langle 1\mathbf{i} + 0\mathbf{j} + 0\mathbf{k}\rangle = \frac{\partial T}{\partial x}$.
With these points in mind, the discretization for the diffusion term is simplified to:

$$\int_S \mathbf{J}\cdot\mathbf{n}dS \approx k\left.\frac{\partial T}{\partial x}\right|_w A_w - k\left.\frac{\partial T}{\partial x}\right|_e A_e \tag{33}$$

The diagram below shows the cell locations and the nomenclature for the distance between them, note how Δx is center-center



We apply finite differences to the derivatives in the diffusion term, i.e.:

$$k \left.\frac{\partial T}{\partial x}\right|_w A_w - k \left.\frac{\partial T}{\partial x}\right|_e A_e = k\frac{T_P - T_W}{\Delta x_{WP}} A_w - k\frac{T_E - T_P}{\Delta x_{PE}} A_e \tag{34}$$

Our discretized source term is simply:

$$\int_V S dV \approx S_P V_P \tag{35}$$

where S_P = value of source term **within** the cell, and V_P = cell volume.
Put everything on one side, we can form the residual equation for the cell **P** as:

$$r_P = -k\frac{T_E - T_P}{\Delta x_{PE}} A_e + k\frac{T_P - T_W}{\Delta x_{WP}} A_w - S_P V_P \tag{36}$$

or expressing in terms of the diffusive fluxes, $\mathbf{F}^d$, through each face:

$$r_P = F_e^d - F_w^d - S_P V_P \tag{37}$$

where:

$$F_e^d = -k\frac{T_E - T_P}{\Delta x_{PE}} A_e \ = -D_e(T_E - T_P) \tag{38}$$

$$F_w^d = -k\frac{T_P - T_W}{\Delta x_{WP}} A_w = -D_w(T_P - T_W) \tag{39}$$

$$D_e = \frac{kA_e}{\Delta x_{PE}} \tag{40}$$

$$D_w = \frac{kA_w}{\Delta x_{WP}} \tag{41}$$

Our cell residual equation is then:

$$r_P = D_w(T_P - T_W) - D_e(T_E - T_P) - S_P V_P \tag{42}$$

The linearized coefficients are then calculated as:

$$a_P = \frac{\partial r_P}{\partial T_P} = D_w + D_e - \frac{\partial S_P}{\partial T_P} V_P \tag{43}$$

$$a_W = \frac{\partial r_P}{\partial T_W} = -D_w \tag{44}$$

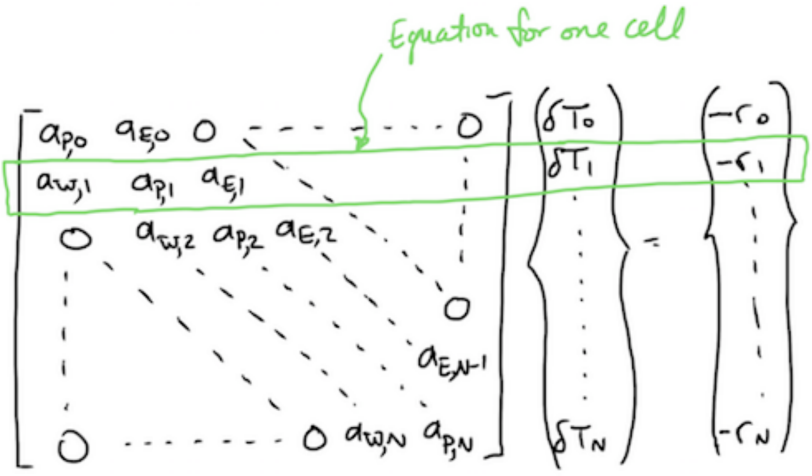
$$a_E = \frac{\partial r_P}{\partial T_E} = -D_e \tag{45}$$

Recall that we can form an algebraic system of equation for each control volume like this:

$$a_P \delta\phi_P + \sum_{nb} a_{nb} \delta\phi_{nb} = -r_P \tag{46}$$

$$a_P \delta T_P + a_W \delta T_W + a_E \delta T_E = -r_P \tag{47}$$

The above linear system of equations can be written as as tridiagonal matrix, like this:



Note: The first and last row only has 2 non zero elements each. This is because these are the left most/right most side and they are adjacent to the domain boundary. Therefore, special boundary conditions are needed to be set.
In matrix notation, we are solving:

$$\mathbf{Ax} = \mathbf{b} \tag{48}$$

where **A** is the Jacobian matrix, **b** = **-r** is the residual vector, **x** = δ**T** is the solution correction. At each current iteration i, the solution is updated according to:

$$\mathbf{T} = \mathbf{T}_i + \delta\mathbf{T}i \tag{49}$$

2.3 Source Terms

Our source term can have many forms, depending on the type of heat source. We will assume *external convection* and *radiation exchange*:

- For external convection:

$$\frac{S_{conv,P}}{V_P} = -hA_0(T_P - T_{\infty,c}) \tag{50}$$

  where:

  - h is the convective coefficient.
  - A_0 is external surface area of the cell P.
  - T_P is temperature at the centroid of cell P.
  - $T_{\infty,c}$ is the ambient temperature for the convection process.

- For radiation exchange:

$$\frac{S_{rad}}{V_P} = -\epsilon\sigma A_0(T_P^4 - T_{\infty,r}^4) \tag{51}$$

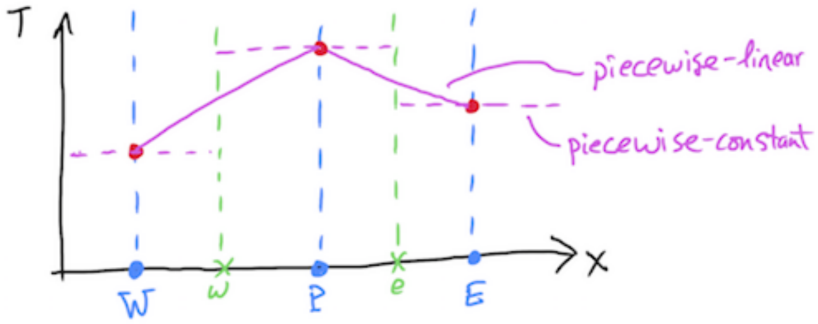
  where:

  - ϵ is the surface emissivity.
  - σ is the Stefan-Boltzmann constant.
  - $T_{\infty,r}$ is the surrounding temperature for radiation exchange.

2.4 Discussion of Discretization Procedure

2.4.1 Temperature Profile Assumptions

When computing the diffusive fluxes through the faces, we assumed a **piecewise-linear profile** for the temperature. This ensures that the derivatives are defined at the integration points and provides consistency for flux at control-volume faces. For the source term, **piece-wise constant profile** is used, implying a single value of the source term in each cell. Note that for piece-wise constant profile, the derivatives are not defined at integration points, due to jump discontinuity. So if fluxes will be inconsistent if piecewise-constant profile is used for temperature.



2.4.2 Implementation of Linearization

In Patakar's method, the solution of the linear system **is** the solution for the variables at the control volume center.
In our method, the solution of the linear system is the **correction** to apply to the previous iteration of the solution.
The correction method is preferred because:

- at convergence, the solution for the correction goes to zero $\rightarrow$ zero a good initial guess for the linear solver.
- linear system involves the residual vector. In Patankar's, there are more work to calculate the residual vector.

2.4.3 Properties of the Discrete Algebraic Equations

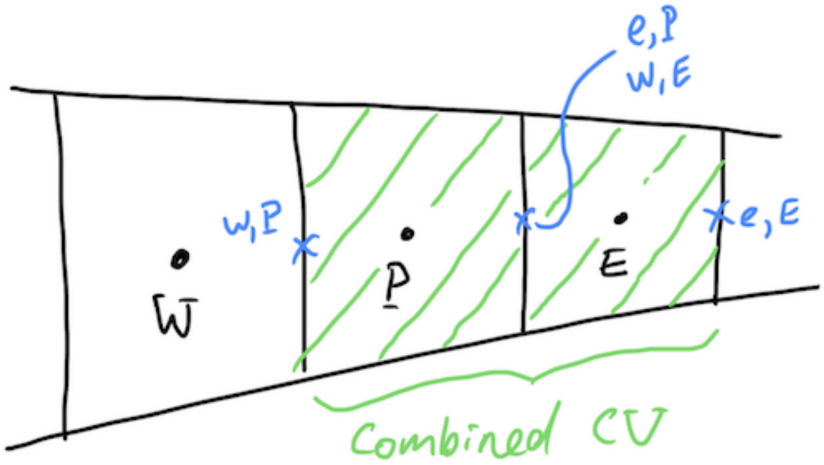
Recall our algebraic equation for the linear system

$$a_P\delta T_P + a_W\delta T_W + a_E\delta T_E = -r_P \tag{52}$$

In Rule 2, we require that $a_P > 0$ and $a_W, a_E < 0$. The reason for this is if we consider the case with no source, and the solution converge, $r_P \to 0$:

$$a_P \delta T_P = -a_W \delta T_W - a_E \delta T_E \tag{53}$$

Now, suppose both T_P and T_E are pertubed. If either of these temperatures were to rise, then T_P would also rise. Similarly, if either temperatures were to drop, T_P should also drop. Therefore, to ensure correct physical effect, if $a_P > 0$ then $a_W, a_E > 0$.
Consider the two cells (P and E) below:



At convergence, $r_P = 0$, the equation for the control volume P is:

$$F^d_{e,P} - F^d_{w,P} - S_P V_P = 0 \tag{54}$$

For the control volume E:

$$F^d_{e,E} - F^d_{w,E} - S_E V_E = 0 \tag{55}$$

Adding these equations together gives:

$$F^d_{e,P} - F^d_{w,P} + F^d_{e,E} - F^d_{w,E} - S_P V_P - S_E V_E = 0 \tag{56}$$

Note that $F^d_{e,P} = F^d_{w,E}$ by continuity, i.e. the flux at cell P going eastward should be the same flux going from westward at cell E. If these are not equal, then it implies that there is a fictuous force at the face, which is not reasonable. Therefore, our algebraic equation for control volume P and E becomes:

$$F^d_{e,E} - F^d_{w,P} - S_P V_P - S_E V_E = 0 \tag{57}$$

The above equation demonstrates integral conservation: a balance of the total source term within the combined control volume with the net diffusive flux from that same control volume. In addition, recall the definition of the diffusive flux:

$$F^d_{e,P} = -k \frac{T_E - T_P}{\Delta x_{PE}} A_{e,P} \tag{58}$$

$$F^d_{w,E} = -k \frac{T_E - T_P}{\Delta x_{PE}} A_{w,E} \tag{59}$$

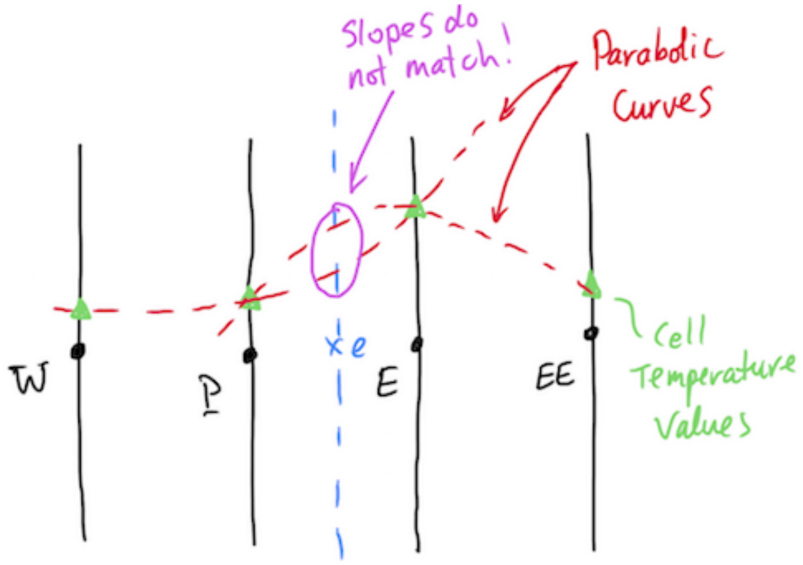
From the gemeotry of the grid, $A_{e,P} = A_{w,E}$; therefore, it is in fact the two-point finite difference estimation of the derivative that cause the fluxes to be equal. This is also due to the piecewise-linear

profile that we assume. If we assume a **parabolic profile** instead, there is no guarantee that the fluxes would be equal. Instead, we would have:

$$F^d_{e,P} = f(T_W, T_P, T_E) \tag{60}$$

$$F^d_{w,E} = f(T_P, T_E, T_{EE}) \tag{61}$$

This means that the flux through the common face depends on different temperature, so we cannot be sure that the derivative from either side is consistent.



2.5 Implemenation: Python Code

```
# Solving the 1D Diffusion Equation
import numpy as np
from enum import Enum
from scipy.sparse.linalg import spsolve
from scipy.sparse import csr_matrix
from numpy.linalg import norm
import sys
import matplotlib.pyplot as plt


# class Grid: defining a 1D cartesian grid
class Grid:

    def __init__(self,lx,ly,lz,ncv):
        # # constructor
        # lx = total length of domain in x direction [m]
        # ly = total length of domain in y direction [m]
        # lz = total length of domain in z direction [m]

        # store the number of control volumes
        self._ncv = ncv

        # calculate the control volume length
```

```
        dx = lx/float(ncv)

        # calculate the face locations
        # generate array of control volume element with length dx each
        self._xf = np.array([i*dx for i in range(ncv+1)])

        # calculate the cell centroid locations
        # Note: figure out why add a xf[-1] last item
        self._xP = np.array([self._xf[0]] +
                            [0.5*(self._xf[i]+self._xf[i+1]) for i in range(ncv)]
    +
                            [self._xf[-1]])
        # calculate face areas
        self._Af = ly*lz*np.ones(ncv+1)

        # calculate the outer surface area of each cell
        self._Ao = (2.0*dx*ly + 2.0*dx*lz)*np.ones(ncv)

        # calculate cell volumes
        self._vol = dx*ly*lz*np.ones(ncv)

    @property
    def ncv(self):
        # number of control volumes in domain
        return self._ncv

    @property
    def xf(self):
        # face location array
        return self._xf

    @property
    def xP(self):
        # cell centroid array
        return self._xP

    @property
    def dx_WP(self):
        return self.xP[1:-1]-self.xP[0:-2]

    @property
    def dx_PE(self):
        return self.xP[2:]-self.xP[1:-1]

    @property
    def Af(self):
        # Face area array
        return self._Af

    @property
    def Aw(self):
        # West face area array
        return self._Af[0:-1]
```

```
    @property
    def Ae(self):
        # East face area array
        return self._Af[1:]

    @property
    def Ao(self):
        # Outer face area array
        return self._Ao

    @property
    def vol(self):
        # Cell volume array
        return self._vol



# class ScalarCoeffs defining coeficients for the linear system
class ScalarCoeffs:
    def __init__(self,ncv):
        # # constructor:
        # ncv = number of control volume in domain

        self._ncv = ncv
        self._aP = np.zeros(ncv)
        self._aW = np.zeros(ncv)
        self._aE = np.zeros(ncv)
        self._rP = np.zeros(ncv)

    def zero(self):
        # zero out the coefficient arrays
        self._aP.fill(0.0)
        self._aW.fill(0.0)
        self._aE.fill(0.0)
        self._rP.fill(0.0)

    def accumulate_aP(self,aP):
        # accumulate values onto aP
        self._aP += aP

    def accumulate_aW(self,aW):
        # accumulate values onto aW
        self._aW += aW

    def accumulate_aE(self,aE):
        # accumulate values onto aE
        self._aE += aE

    def accumulate_rP(self,rP):
        # accumulate values onto rP
        self._rP += rP

    @property
    def ncv(self):
```

```
        # number of control volume in domain
        return self._ncv

    @property
    def aP(self):
        # cell coefficient
        return self._aP

    @property
    def aW(self):
        # West cell coefficient
        return self._aW

    @property
    def aE(self):
        # East cell coefficient
        return self._aE

    @property
    def rP(self):
        # cell coefficient
        return self._rP


# class BoundaryLocation defining boundary condition locations
class BoundaryLocation(Enum):
    WEST = 1
    EAST = 2


# Implementing Boundary Conditions
class DirichletBC:
    def __init__(self,phi,grid,value,loc):
        # # constructor
        # phi = field variable array
        # grid = grid
        # value = boundary value
        # loc = boundary location

        self._phi = phi
        self._grid = grid
        self._value = value
        self._loc = loc

    def value(self):
        # return the boundary condition value
        return self._value

    def coeff(self):
        # return the linearization coefficient
        return 0

    def apply(self):
        # applies the boundary condition in the referenced field variable array
```

```
        if self._loc is BoundaryLocation.WEST:
            self._phi[0] = self._value
        elif self._loc is BoundaryLocation.EAST:
            self._phi[-1] = self._value
        else:
            raise ValueError("Unknown boundary location")


class NeumannBC:
    def __init__(self,phi,grid,gradient,loc):
        # # constructor
        # phi = field variable array
        # grid = grid
        # gradient = gradient at cell adjacent to boundary
        # loc = boundary location

        self._phi = phi
        self._grid = grid
        self._gradient = gradient
        self._loc = loc

    def value(self):
        # return the boundary condition value
        if self._loc is BoundaryLocation.WEST:
            return self._phi[1]-self._gradient*self._grid.dx_WP[0] # T(0) = T(1) -
    gb*dx
        elif self._loc is BoundaryLocation.EAST:
            return self._phi[-2]-self._gradient*self._grid.dx_PE[-1] #T(-1) = T
   (-2) -gb*dx
        else:
            raise ValueError("Unknown boundary location")


    def coeff(self):
        # return the linearization coefficient
        return 1

    def apply(self):
        # applies the boundary condition in the referenced field variable array
        if self._loc is BoundaryLocation.WEST:
            self._phi[0] = self._phi[1] - self._gradient*self._grid.dx_WP[0]
        elif self._loc is BoundaryLocation.EAST:
            self._phi[-1] = self._phi[-2] + self._gradient*self._grid.dx_PE[-1]
        else:
            raise ValueError("Unknown boundary location")

class RobinBC:
    def __init__(self,phi,grid,h,k,tinfty, loc):
        # # constructor
        # phi = field variable array
        # grid = grid
        # h = convective coefficient
        # k = conductive coefficient
        # tinfty = freestream temperature
```

```
        # loc = boundary location

        self._phi = phi
        self._grid = grid
        self._h = h
        self._k = k
        self._tinfty = tinfty
        self._loc = loc

    def value(self):
        # return the boundary condition value
        if self._loc is BoundaryLocation.WEST:
            return (self._phi[1] + self._grid.dx_WP[0]*(self._h/self._k)*(self.
    _tinfty)) \
                /(1 + self._grid.dx_WP[0]*(self._h/self._k))
        elif self._loc is BoundaryLocation.EAST:
            return (self._phi[-2] - self._grid.dx_PE[-1]*(self._h/self._k)*(self.
    _tinfty))\
                /(1 - self._grid.dx_PE[-1]*(self._h/self._k))
        else:
            raise ValueError("Unknown boundary location")


    def coeff(self):
        # return the linearization coefficient
        return 1

    def apply(self):
        # applies the boundary condition in the referenced field variable array
        if self._loc is BoundaryLocation.WEST:
            self._phi[0]  =  (self._phi[1] + self._grid.dx_WP[0]*(self._h/self._k)
    *(self._tinfty))\
                /(1 + self._grid.dx_WP[0]*(self._h/self._k))
        elif self._loc is BoundaryLocation.EAST:
            self._phi[-1] = (self._phi[-2] - self._grid.dx_PE[-1]*(self._h/self._k
    )*(self._tinfty))\
                /(1 - self._grid.dx_PE[-1]*(self._h/self._k))
        else:
            raise ValueError("Unknown boundary location")



# class DiffusionModel defining a defusion model
class DiffusionModel:
    def __init__(self,grid,phi,gamma,west_bc,east_bc):
        # constructor
        self._grid = grid
        self._phi = phi
        self._gamma = gamma     # gamma here is "k" in the heat model
        self._west_bc = west_bc
        self._east_bc = east_bc
```

```
    # add diffusion terms to coefficient arrays
    def add(self,coeffs):

        # calculate west/east diffusion flux for each face
        flux_w = -self._gamma*self._grid.Aw*(self._phi[1:-1]-self._phi[0:-2])\
            /self._grid.dx_WP # Fw = k*(Tp-Tw)*Aw/ dx_WP
        flux_e = -self._gamma*self._grid.Ae*(self._phi[2:]-self._phi[1:-1])\
            /self._grid.dx_PE # Fe = k*(Te-Tp)*Ae/dx_PE

        # calculate the linearized coefficient [aW, aP, aE]
        coeffW = -self._gamma*self._grid.Aw/self._grid.dx_WP
        coeffE = -self._gamma*self._grid.Ae/self._grid.dx_PE
        coeffP = -coeffW - coeffE # should be +?

        # modified the linearized coefficient on the boundaries
        coeffP[0] += coeffW[0]*self._west_bc.coeff()
        coeffP[-1] += coeffE[-1]*self._east_bc.coeff()

        # zero the coefficients that are not used (on the boundary)
        coeffW[0] = 0.0
        coeffE[-1] = 0.0

        # calculate the net flux from each cell (out -in)
        flux = flux_e - flux_w

        # add to coefficient arrays
        coeffs.accumulate_aP(coeffP)
        coeffs.accumulate_aW(coeffW)
        coeffs.accumulate_aE(coeffE)
        coeffs.accumulate_rP(flux)

        # return the coefficient arrays
        return coeffs


# function to return a sparse matrix representation of a set of scalar
    coefficients
def get_sparse_matrix(coeffs):
    ncv = coeffs.ncv
    data = np.zeros(3*ncv-2)
    rows = np.zeros(3*ncv-2, dtype = int)
    cols = np.zeros(3*ncv-2, dtype = int)
    data[0] = coeffs.aP[0]
    rows[0] = 0
    cols[0] = 0

    if ncv > 1:
        data[1] = coeffs.aE[0]
        rows[1] = 0
        cols[1] = 1

    for i in range(ncv-2):
        data[3*i+2] = coeffs.aW[i+1]
        data[3*i+3] = coeffs.aP[i+1]
```

```
        data[3*i+4] = coeffs.aE[i+1]

        rows[3*i+2:3*i+5] = i+1

        cols[3*i+2] = i
        cols[3*i+3] = i+1
        cols[3*i+4] = i+2

    if ncv > 1:
        data[3*ncv-4] = coeffs.aW[-1]
        data[3*ncv-3] = coeffs.aP[-1]

        rows[3*ncv-4:3*ncv-2] = ncv-1

        cols[3*ncv-4] = ncv-2
        cols[3*ncv-3] = ncv-1

    return csr_matrix((data, (rows,cols)))

# solve the linear system and return the field variables
def solve(coeffs):
    # get the sparse matrix
    A = get_sparse_matrix(coeffs)
    # solve the linear system
    return spsolve(A, -coeffs.rP)




# Solving the 1D steady conduction with Dirichet BC
# Initially, east = 300K, west = 300K
# define the grid
lx = 1.0
ly = 0.1
lz = 0.1
ncv = 10
mygrid = Grid(lx,ly,lz,ncv)

#set the max iterations and convergence criterion
maxIter = 100
converged=1e-6

# thermal properties
k = 0.1

# coefficients
coeffs = ScalarCoeffs(mygrid.ncv)

# # initial condition
T0 = 300

# # initialize field variable arrays
```

```
T = T0*np.ones(mygrid.ncv+2)

### boundary condition
west_bc = DirichletBC(T, mygrid, 400, BoundaryLocation.WEST)
east_bc = DirichletBC(T, mygrid, 0, BoundaryLocation.EAST)

### apply boundary conditions
west_bc.apply()
east_bc.apply()


### list to store the solution at each iteration
T_solns = [np.copy(T)]

### define the diffusion model
diffusion = DiffusionModel(mygrid, T, k, west_bc, east_bc)

### iterate until the solution is converged
for i in range(maxIter):
    # zero the coeff and add
    coeffs.zero()
    coeffs = diffusion.add(coeffs)

    # compute residual and check for convergence
    maxResid = norm(coeffs.rP, np.inf)
    avgResid = np.mean(np.absolute(coeffs.rP))
    print("Iteration = {} ; Resid = {} ; Avg. Resid = {}".format(i,maxResid,
    avgResid) )
    if maxResid < converged:
        break

    # solve the sparse matrix system
    dT = solve(coeffs)

    # update the solution and boundary conditions
    T[1:-1] = T[1:-1] + dT
    west_bc.apply()
    east_bc.apply()

    # store the solution
    T_solns.append(np.copy(T))



# plotting
for Ti in T_solns:
    plt.plot(mygrid.xP, Ti, label = str(i))
    i += 1

plt.title('Explicit')
plt.xlabel('X')
plt.ylabel('T')
plt.savefig('pic/1d_transient_explicit.png')
```

3 TRANSIENT 1D HEAT DIFFUSION

3.1 Problem Definition

In contrast to the steady case, here we are solving the following transient 1D heat diffusion equation

$$\frac{\partial(\rho c_p T)}{\partial t} = k\nabla^2 T + S \tag{62}$$

assuming constant ρ and c_p

3.2 Discretization

Integrating the governing equation through space and time yields:

$$\int_{t_0}^{t_1}\int_V \frac{(\partial \rho c_p T)}{\partial t} dt dV = \int_{t_0}^{t_1}\int_V k\nabla^2 T dt dV + \int_{t_0}^{t_1}\int_V S dt dV \tag{63}$$

By assuming a timestep $\Delta t = t_1 - t_0$, we also assume that the solution is stored at time levels t and later at $t + \Delta t$. Thus, we can assume various profiles for the integrands as functions of time. Here, we will examine the following:

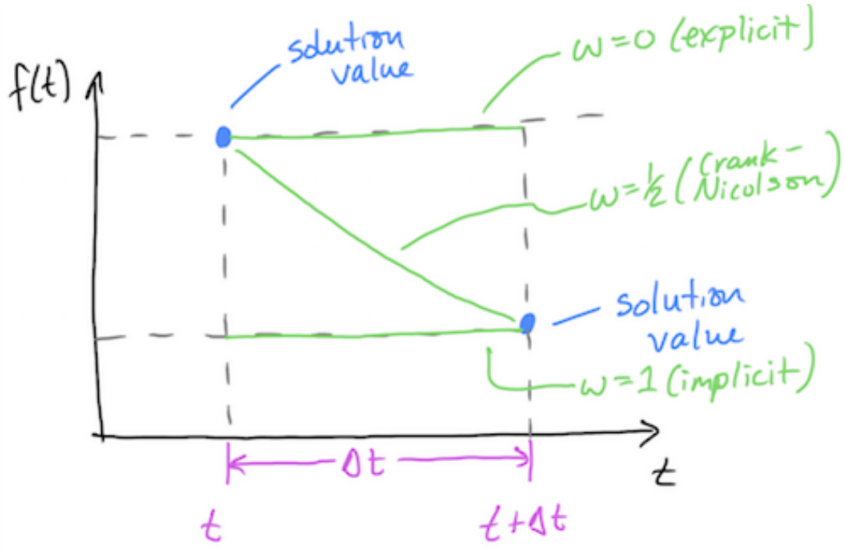
- **Fully explicit**: evaluate the integrands on RHS at initial time level, $t_0 = t$.
- **Fully implicit**: evaluate the integrands on RHS at final time level, $t_1 = t_0 + \Delta t$
- **Crank Nicolson**: assume a linear profile of the RHS over the interval Δt.

For the LHS, we interchange the order of integration, which results in this, for a control volume **P**:

$$\int_{t_0}^{t_1}\int_V \frac{(\partial \rho c_p T)}{\partial t} dt dV = (\rho c_p T_p)^{t+\Delta t} - (\rho c_p T_p)^t \tag{64}$$

Recall that for the steady case, the diffusive term is the difference in flux. Here, we add a weighting function w to control the assumed variation of the integrand over the timestep.

$$\int_{t_0}^{t_1}\int_V k\nabla^2 T dt dV = -\left[\omega (F_e^d)^{t+\Delta t} + (1-\omega)(F_e^d)^t\right]\Delta t + \left[\omega (F_w^d)^{t+\Delta t} + (1-\omega)(F_w^d)^t\right]\Delta t \tag{65}$$



Grouping the time levels together:

$$\int_{t_0}^{t_1}\int_V k\nabla^2 T dt dV = \left[\omega[F_w^d - F_e^d\right]^{t+\Delta t}\Delta t + (1-\omega)\left[F_w^d - F_e^d\right]^t \Delta t \tag{66}$$

Assuming no source term, $S = 0$, constant properties and divide by Δt, our discretized equation becomes

$$\rho c_p \frac{T_P^{t+\Delta t} - T_P^t}{\Delta t} V_P = \omega\left[F_w^d - F_e^d\right]^{t+\Delta t} + (1-\omega)\left[F_w^d - F_e^d\right]^t \tag{67}$$

Note

- fully implicit and fully explicit are *1st order* accurate in time.
- Crank-Nicolson are *2nd order* accurate in time.
  - but Crank-Nicolson are *less stable*, cause oscillations.
- Generally, explicit solutions do not require the solution of a system of equations. All diffusive fluxes are calculated using solution values from previous timestep
- Implicit requires solution of a linear system at current time step. The same is true for Crank-Nicolson, or any scheme where $0 < \omega < 1$

3.3 Analysis of Explicit Scheme

Recall explicit scheme means $\omega = 0$. Our discretized equation becomes:

$$\rho c_p \frac{T_P^{t+\Delta t}}{\Delta t} V_P = [F_w^d - F_e^d]^t + \rho c_p \frac{T_P^t}{\Delta t} V_P \tag{68}$$

Recall that the fluxes can be defined as:

$$F_e^d = -k\frac{T_E - T_P}{\Delta x_{PE}} A_e \ = -D_e(T_E - T_P) \tag{69}$$

$$F_w^d = -k\frac{T_P - T_W}{\Delta x_{WP}} A_w = -D_w(T_P - T_W) \tag{70}$$

$$D_e = \frac{kA_e}{\Delta x_{PE}} \tag{71}$$

$$D_w = \frac{kA_w}{\Delta x_{WP}} \tag{72}$$

Thus, our explicit formulation becomes:

$$\rho c_p \frac{T_P^{t+\Delta t}}{\Delta t} V_P = D_e T_E^t + D_w T_W^t + \left(\frac{\rho c_p V_P}{\Delta t} - D_e - D_w\right) T_P^t \tag{73}$$

To get correct physical influence, the coefficient of T_P^t must be positive, to ensure $\uparrow T_P^t$ leads to $\uparrow T_P^{t+\Delta t}$. Therefore, our timestep must be selected such that:

$$\frac{\rho c_p V_P}{\Delta t} \geq D_e + D_w$$

or

$$\Delta t \leq \frac{\rho c_p V_P}{D_e + D_w} = \frac{1}{\frac{D_e}{\rho c_p V_P} + \frac{D_w}{\rho c_p V_P}}$$

Simplifying further, we assume $V_P = A\Delta x$ where A is the cross-sectional area of the domain at P, and Δx is the grid spacing. Also assuming $A_e, A_w \approx A$

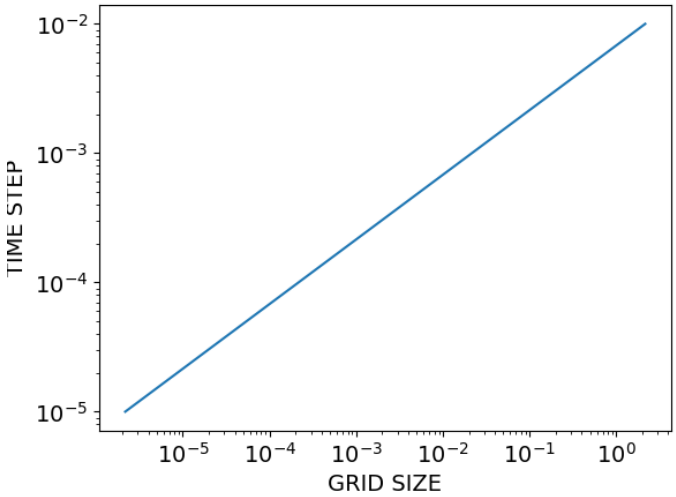
$$\frac{D_e}{\rho c_p V_P} \approx \frac{\frac{kA_e}{\Delta x}}{\rho c_p A \Delta x} = \frac{k}{\rho c_p}\frac{1}{\Delta x^2} = \frac{\alpha}{\Delta x^2}$$

The quantity $\frac{\Delta x^2}{\alpha}$ may be interpreted as the timescale associated with conduction through the face. For uniform grid, $A_e = A_w = A$, the timestep restriction is:

$$\Delta t \leq \frac{1}{\frac{\alpha}{\Delta x^2} + \frac{\alpha}{\Delta x^2}} = \frac{\Delta x^2}{2\alpha} \tag{74}$$

Note how our timestep is related to the square of the grid size, so the a fine grid will have a very small "Δx^2". To study this, we consider an iron bar with $\alpha = 23.1 \times 10^{-6}$ $[m^2/s]$ at various grid sizes:

GRID SIZE [m]	TIME STEP [sec]
0.01	2.1645022
0.001	0.021645022
0.0001	2.1645022e-4
0.00001	2.1645022e-6



We can quickly see how the timestep restriction gets worse with increasing grid refinement. As a result, implicit methods are more commonly used in practice. Exception would be calculation of turbulent flow using direct numerical simulation (DNS). In this case, explicit method are a good choice because they are less expensive per timestep, since no linear system must be solved.

3.4 Analysis of Fully Implicit Scheme

Setting $\omega = 1$ for implicit scheme. Our discretized equation becomes:

$$\rho c_p \frac{T_P^{t+\Delta t} - T_P^t}{\Delta t} V_P = [-D_w(T_P - T_W) + D_e(T_E - T_P)]^{t+\Delta t} \tag{75}$$

Drop the superscript $t + \Delta t$ and denotes old value as "o", after rearranging, we have:

$$\left(\frac{\rho c_p V_P}{\Delta t} + D_w + D_e\right) T_P = D_w T_W + D_e T_E + \frac{\rho c_p V_P}{\Delta t} T_P^o \tag{76}$$

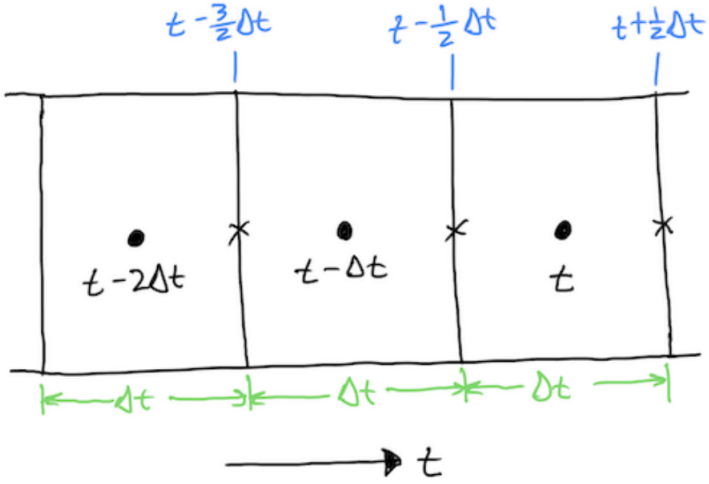
We can see that none of the coefficients can be come negative when written out this way. Still, we must ensure the timestep is small enough to resolve all transient phenomena. In contrast to this fully implicit scheme, the Crank-Nicholson scheme has no formal restriction on Δt, but still produces **oscillatory** at large Δt.

3.5 Derivation of Second Order Implicit Scheme

3.5.1 General idea

We consider integration over a special **space-time control volume**, with:

- the time faces being located at $t - \Delta t/2, t + \Delta t/2$
- the solution values are stored at $t, t - \Delta t, t - 2\Delta t$.



By using the space-time control volume, the RHS of the discretized equation evaluated at time t, can be considered as a representative of the entire timesweep. The advantages are:

- we do not need to assume a profile in time, e.g. piecewise constant for fully implicit/explicit, piecewise linear for Crank-Nicholson.
- no need to store old flux value
- interpolation depends on the face values:
  - if piecewise constant $\rightarrow$ 1st order scheme
  - if piecewise linear $\rightarrow$ 2nd order scheme

3.5.2 Derivation

Integrate the governing equation over the space-time control volume

$$\int_{t-\Delta t/2}^{t+\Delta t/2}\int_V \frac{(\partial \rho c_p T)}{\partial t} dt dV = \int_{t-\Delta t/2}^{t+\Delta t/2}\int_V k\nabla^2 T dt dV + \int_{t-\Delta t/2}^{t+\Delta t/2}\int_V S dt dV \tag{77}$$

Resulting in:

$$(\rho c_p T_p V_p)^{t+\Delta t/2} - (\rho c_p T_p V_p)^{t-\Delta t/2} = [F_w^d - F_e^d]^t \Delta t + S_P^t \Delta t V_P \tag{78}$$

Divide by Δt, express the diffusive fluxes in terms of D_w and D_e, dropping superscripts t for current time:

$$\frac{(\rho c_p T_P V_P)^{t+\Delta t/2} - (\rho c_p T_P V_P)^{t-\Delta t/2}}{\Delta t} = -D_w(T_P - T_W) + D_e(T_E - T_P) + S_P V_P \tag{79}$$

LHS is known, for RHS $\rightarrow$ need to specify values for times \$t-Δ t/2 and $t + \Delta t/2$

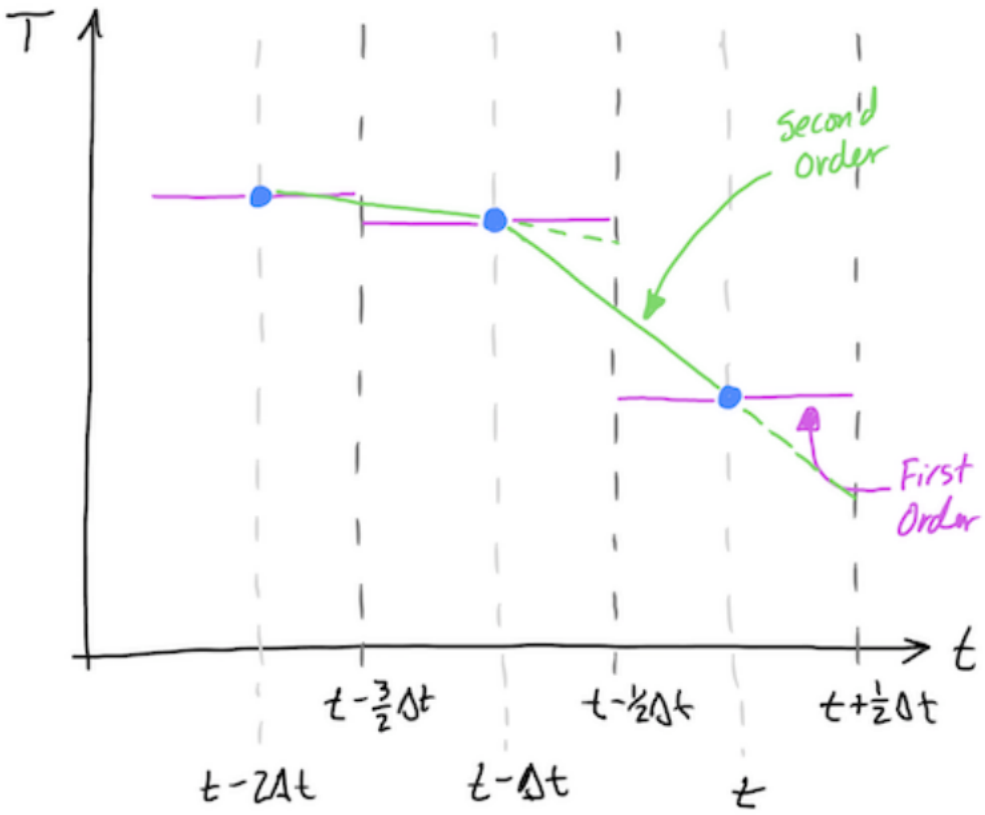
- 1st order time integration scheme We assume a **piecewise constant** distribution over each timestep between the face values, resulting in:

$$T_P^{t-\Delta t/2} = T_P^{t-\Delta t}$$
$$T_P^{t+\Delta t/2} = T_P^{t}$$

- 2nd order time integration scheme We assume a **piecewise linear** distribution over each timestep between the face values, resulting in:

$$T_P^{t-\Delta t/2} = T_P^{t-\Delta t} + \frac{1}{2}(T_P^{t-\Delta t} - T_P^{t-2\Delta t})$$
$$T_P^{t+\Delta t/2} = T_P^{t} + \frac{1}{2}(T_P^{t} - T_P^{t-\Delta t})$$

  This is achieved by doing backward interpolation and then forward interpolation on the face values. The schematic below shows these interpolations:



Substituting these relations to the integrated governing equation's LHS:

- For 1st order scheme:

$$\frac{(\rho c_p T_P V_P)^{t+\Delta t/2} - (\rho c_p T_P V_P)^{t-\Delta t/2}}{\Delta t} = \rho c_p V_P \frac{T_P - T_P^o}{\Delta t} \tag{80}$$

  Note:

  - the superscript for current time is dropped, and superscript for $t - \Delta t$ is replaced by o for "old value".
  - also that this is exactly the same as the result for the fully implicit scheme.

- For 2nd order scheme:

$$\frac{(\rho c_p T_P V_P)^{t+\Delta t/2} - (\rho c_p T_P V_P)^{t-\Delta t/2}}{\Delta t} = \rho c_p V_P \frac{T_P + \frac{1}{2}(T_P - T_P^o) - T_P^o - \frac{1}{2}(T_P^o - T_P^{oo})}{\Delta t}$$

  or a more simplified version. . . .

$$\frac{(\rho c_p T_P V_P)^{t+\Delta t/2} - (\rho c_p T_P V_P)^{t-\Delta t/2}}{\Delta t} = \rho c_p V_P \frac{\frac{3}{2}T_P - 2T_P^o + \frac{1}{2}T_P^{oo}}{\Delta t} \tag{81}$$

  Note:

  - superscript oo is used for time value $t - 2\Delta t$.
  - unlike Crank-Nicholson's, flux values at previous timestep **do not need to be solved**. Instead, only temperature values at the **previous two time step** need to be retained.

3.5.3 Other Transient Discretization Schemes

Some higher order schemes are also used such as:

- Adams-Bashforth (explicit)
- Adams-Moulton (implicit)
- Runge-Kutta (implicit or explicit)

3.5.4 Linearization

Recall the cell residual for steady conduction:

$$r_P = D_w(T_P - T_W) - D_e(T_E - T_P) - S_P V_P$$

where $D_e = \frac{kA_e}{\Delta x_{PE}}$, and $D_w = \frac{kA_w}{\Delta x_{WP}}$. If we apply 1st order implicit to the transient term:

$$r_P = \rho c_p V_P \frac{T_P - T_P^o}{\Delta t} D_w(T_P - T_W) - D_e(T_E - T_P) - S_P V_P$$

This makes the linearization coefficients to be as follow:

$$\begin{aligned} a_P &= \frac{\partial r_P}{\partial T_P} = \frac{\rho c_p V_P}{\Delta t} + D_w + D_e - \frac{\partial S_P}{\partial T_P} V_P \\ a_W &= \frac{\partial r_P}{\partial T_W} = -D_w \\ a_E &= \frac{\partial r_P}{\partial T_E} = -D_e \end{aligned}$$

Similar to before, we can form an algebraic equation for each control volume like this:

$$a_P \delta T_P + a_W \delta T_W + a_E \delta T_E = -r_P \tag{82}$$

If we apply 2nd order implicit temporal scheme instead, then a_P term would look like this:

$$a_P = \frac{\partial r_P}{\partial T_P} = \frac{3}{2}\frac{\rho c_p V_P}{\Delta t} + D_w + D_e - \frac{\partial S_P}{\partial T_P} V_P$$

3.6 Implementation: Python code

```
# Solving the 1D Diffusion Equation-TRANSIENT
import numpy as np
from enum import Enum
from scipy.sparse.linalg import spsolve
from scipy.sparse import csr_matrix
from numpy.linalg import norm
import sys
import matplotlib.pyplot as plt


# class Grid: defining a 1D cartesian grid
class Grid:
```

```
    def __init__(self,lx,ly,lz,ncv):
        # # constructor
        # lx = total length of domain in x direction [m]
        # ly = total length of domain in y direction [m]
        # lz = total length of domain in z direction [m]

        # store the number of control volumes
        self._ncv = ncv

        # calculate the control volume length
        dx = lx/float(ncv)

        # calculate the face locations
        # generate array of control volume element with length dx each
        self._xf = np.array([i*dx for i in range(ncv+1)])

        # calculate the cell centroid locations
        # Note: figure out why add a xf[-1] last item
        self._xP = np.array([self._xf[0]] +
                            [0.5*(self._xf[i]+self._xf[i+1]) for i in range(ncv)]
    +
                            [self._xf[-1]])
        # calculate face areas
        self._Af = ly*lz*np.ones(ncv+1)

        # calculate the outer surface area of each cell
        self._Ao = (2.0*dx*ly + 2.0*dx*lz)*np.ones(ncv)

        # calculate cell volumes
        self._vol = dx*ly*lz*np.ones(ncv)

    @property
    def ncv(self):
        # number of control volumes in domain
        return self._ncv

    @property
    def xf(self):
        # face location array
        return self._xf

    @property
    def xP(self):
        # cell centroid array
        return self._xP

    @property
    def dx_WP(self):
        return self.xP[1:-1]-self.xP[0:-2]

    @property
    def dx_PE(self):
        return self.xP[2:]-self.xP[1:-1]
```

```
    @property
    def Af(self):
        # Face area array
        return self._Af

    @property
    def Aw(self):
        # West face area array
        return self._Af[0:-1]

    @property
    def Ae(self):
        # East face area array
        return self._Af[1:]

    @property
    def Ao(self):
        # Outer face area array
        return self._Ao

    @property
    def vol(self):
        # Cell volume array
        return self._vol



# class ScalarCoeffs defining coeficients for the linear system
class ScalarCoeffs:
    def __init__(self,ncv):
        # # constructor:
        # ncv = number of control volume in domain

        self._ncv = ncv
        self._aP = np.zeros(ncv)
        self._aW = np.zeros(ncv)
        self._aE = np.zeros(ncv)
        self._rP = np.zeros(ncv)

    def zero(self):
        # zero out the coefficient arrays
        self._aP.fill(0.0)
        self._aW.fill(0.0)
        self._aE.fill(0.0)
        self._rP.fill(0.0)

    def accumulate_aP(self,aP):
        # accumulate values onto aP
        self._aP += aP

    def accumulate_aW(self,aW):
        # accumulate values onto aW
        self._aW += aW
```

```
    def accumulate_aE(self,aE):
        # accumulate values onto aE
        self._aE += aE

    def accumulate_rP(self,rP):
        # accumulate values onto rP
        self._rP += rP

    @property
    def ncv(self):
        # number of control volume in domain
        return self._ncv

    @property
    def aP(self):
        # cell coefficient
        return self._aP

    @property
    def aW(self):
        # West cell coefficient
        return self._aW

    @property
    def aE(self):
        # East cell coefficient
        return self._aE

    @property
    def rP(self):
        # cell coefficient
        return self._rP


# class BoundaryLocation defining boundary condition locations
class BoundaryLocation(Enum):
    WEST = 1
    EAST = 2


# Implementing Boundary Conditions
class DirichletBC:
    def __init__(self,phi,grid,value,loc):
        # # constructor
        # phi = field variable array
        # grid = grid
        # value = boundary value
        # loc = boundary location

        self._phi = phi
        self._grid = grid
        self._value = value
        self._loc = loc
```

```
    def value(self):
        # return the boundary condition value
        return self._value

    def coeff(self):
        # return the linearization coefficient
        return 0

    def apply(self):
        # applies the boundary condition in the referenced field variable array
        if self._loc is BoundaryLocation.WEST:
            self._phi[0] = self._value
        elif self._loc is BoundaryLocation.EAST:
            self._phi[-1] = self._value
        else:
            raise ValueError("Unknown boundary location")


class NeumannBC:
    def __init__(self,phi,grid,gradient,loc):
        # # constructor
        # phi = field variable array
        # grid = grid
        # gradient = gradient at cell adjacent to boundary
        # loc = boundary location

        self._phi = phi
        self._grid = grid
        self._gradient = gradient
        self._loc = loc

    def value(self):
        # return the boundary condition value
        if self._loc is BoundaryLocation.WEST:
            return self._phi[1]-self._gradient*self._grid.dx_WP[0] \
                # T(0) = T(1) - gb*dx
        elif self._loc is BoundaryLocation.EAST:
            return self._phi[-2]-self._gradient*self._grid.dx_PE[-1] \
                #T(-1) = T(-2) -gb*dx
        else:
            raise ValueError("Unknown boundary location")


    def coeff(self):
        # return the linearization coefficient
        return 1

    def apply(self):
        # applies the boundary condition in the referenced field variable array
        if self._loc is BoundaryLocation.WEST:
            self._phi[0] = self._phi[1] - self._gradient*self._grid.dx_WP[0]
        elif self._loc is BoundaryLocation.EAST:
            self._phi[-1] = self._phi[-2] + self._gradient*self._grid.dx_PE[-1]
        else:
```

```
            raise ValueError("Unknown boundary location")

class RobinBC:
    def __init__(self,phi,grid,h,k,tinfty, loc):
        # # constructor
        # phi = field variable array
        # grid = grid
        # h = convective coefficient
        # k = conductive coefficient
        # tinfty = freestream temperature
        # loc = boundary location

        self._phi = phi
        self._grid = grid
        self._h = h
        self._k = k
        self._tinfty = tinfty
        self._loc = loc

    def value(self):
        # return the boundary condition value
        if self._loc is BoundaryLocation.WEST:
            return (self._phi[1] + self._grid.dx_WP[0]*(self._h/self._k)*(self.
    _tinfty))\
                /(1 + self._grid.dx_WP[0]*(self._h/self._k))
        elif self._loc is BoundaryLocation.EAST:
            return (self._phi[-2] - self._grid.dx_PE[-1]*(self._h/self._k)*(self.
    _tinfty))\
                /(1 - self._grid.dx_PE[-1]*(self._h/self._k))
        else:
            raise ValueError("Unknown boundary location")


    def coeff(self):
        # return the linearization coefficient
        return 1

    def apply(self):
        # applies the boundary condition in the referenced field variable array
        if self._loc is BoundaryLocation.WEST:
            self._phi[0] = (self._phi[1] + self._grid.dx_WP[0]*(self._h/self._k)
    *(self._tinfty))\
                /(1 + self._grid.dx_WP[0]*(self._h/self._k))
        elif self._loc is BoundaryLocation.EAST:
            self._phi[-1] = (self._phi[-2] - self._grid.dx_PE[-1]*(self._h/self._k
    )*(self._tinfty))\
                /(1 - self._grid.dx_PE[-1]*(self._h/self._k))
        else:
            raise ValueError("Unknown boundary location")



# class DiffusionModel defining a defusion model
class DiffusionModel:
```

```
    def __init__(self,grid,phi,gamma,west_bc,east_bc):
        # constructor
        self._grid = grid
        self._phi = phi
        self._gamma = gamma     # gamma here is "k" in the heat model
        self._west_bc = west_bc
        self._east_bc = east_bc



    # add diffusion terms to coefficient arrays
    def add(self,coeffs):

        # calculate west/east diffusion flux for each face
        flux_w = -self._gamma*self._grid.Aw*(self._phi[1:-1]-self._phi[0:-2])\
            /self._grid.dx_WP # Fw = k*(Tp-Tw)*Aw/ dx_WP
        flux_e = -self._gamma*self._grid.Ae*(self._phi[2:]-self._phi[1:-1])\
            /self._grid.dx_PE # Fe = k*(Te-Tp)*Ae/dx_PE

        # calculate the linearized coefficient [aW, aP, aE]
        coeffW = -self._gamma*self._grid.Aw/self._grid.dx_WP # Dw
        coeffE = -self._gamma*self._grid.Ae/self._grid.dx_PE # De
        coeffP = -coeffW - coeffE                            # Dw + De in notes,
    but due to minus sign so -Dw-De in code

        # modified the linearized coefficient on the boundaries
        coeffP[0] += coeffW[0]*self._west_bc.coeff()
        coeffP[-1] += coeffE[-1]*self._east_bc.coeff()

        # zero the coefficients that are not used (on the boundary)
        coeffW[0] = 0.0
        coeffE[-1] = 0.0

        # calculate the net flux from each cell (out -in)
        flux = flux_e - flux_w

        # add to coefficient arrays
        coeffs.accumulate_aP(coeffP)
        coeffs.accumulate_aW(coeffW)
        coeffs.accumulate_aE(coeffE)
        coeffs.accumulate_rP(flux)

        # return the coefficient arrays
        return coeffs


# function to return a sparse matrix representation of a set of scalar
    coefficients
def get_sparse_matrix(coeffs):
    ncv = coeffs.ncv
    data = np.zeros(3*ncv-2)
    rows = np.zeros(3*ncv-2, dtype = int)
    cols = np.zeros(3*ncv-2, dtype = int)
    data[0] = coeffs.aP[0]
```

```
    rows[0] = 0
    cols[0] = 0

    if ncv > 1:
        data[1] = coeffs.aE[0]
        rows[1] = 0
        cols[1] = 1

    for i in range(ncv-2):
        data[3*i+2] = coeffs.aW[i+1]
        data[3*i+3] = coeffs.aP[i+1]
        data[3*i+4] = coeffs.aE[i+1]

        rows[3*i+2:3*i+5] = i+1

        cols[3*i+2] = i
        cols[3*i+3] = i+1
        cols[3*i+4] = i+2

    if ncv > 1:
        data[3*ncv-4] = coeffs.aW[-1]
        data[3*ncv-3] = coeffs.aP[-1]

        rows[3*ncv-4:3*ncv-2] = ncv-1

        cols[3*ncv-4] = ncv-2
        cols[3*ncv-3] = ncv-1

    return csr_matrix((data, (rows,cols)))

# solve the linear system and return the field variables
def solve(coeffs):
    # get the sparse matrix
    A = get_sparse_matrix(coeffs)
    # solve the linear system
    return spsolve(A, -coeffs.rP)

# defining a first order implicit transient model
class FirstOrderTransientModel:
    # constructor
    def __init__(self,grid,T,Told,rho,cp,dt):
        self._grid = grid
        self._T = T
        self._Told = Told
        self._rho = rho
        self._cp = cp
        self._dt = dt

    def add(self,coeffs):
        # calculate the transient term

        # calculate the linearization coefficients
        coeffP = self.rho*self._cp*self._vol/self._dt # ap = rho*cp*vp/dt
```

```
        # add to coefficient arrays
        coeffs.accumulate_aP(coeffP)

        return coeffs

# class defining a surface convection model
class SurfaceConvectionModel:
    # constructor
    def __init__(self,grid,T,ho,To):
        self._grid = grid
        self._T = T
        self._ho = ho
        self._To = To

    # add surface convection terms to coefficient arrays
    def add(self,coeffs):
        # calculate the source term q = hA(T-Tinfty)
        source = self._ho * self._grid.Ao*(self._T[1:-1]-self._To)

        # calculat linearization coefficients
        coeffP = self._ho * self._grid.Ao

        # add to coefficient arrays
        coeffs.accumulate_aP(coeffP)
        coeffs.accumulate_rP(source)

        return coeffs




# Solving the 1D steady conduction with Dirichet BC
# Initially, east = 300K, west = 300K
# define the grid
lx = 1.0
ly = 0.1
lz = 0.1
ncv = 10
mygrid = Grid(lx,ly,lz,ncv)

# set the timestep information
nTime = 10
dt = 1
time = 0

#set the max iterations and convergence criterion
maxIter = 100
converged=1e-6

# thermal properties
k = 0.1
rho = 1000
cp = 1000
k = 100
```

```
# convection parameters
ho = 25
To = 200

# coefficients
coeffs = ScalarCoeffs(mygrid.ncv)

# initial condition
T0 = 300

# initialize field variable arrays
T = T0*np.ones(mygrid.ncv+2)

# boundary condition
west_bc = DirichletBC(T, mygrid, 400, BoundaryLocation.WEST)
east_bc = DirichletBC(T, mygrid, 0, BoundaryLocation.EAST)

# apply boundary conditions
west_bc.apply()
east_bc.apply()


# list to store the solution at each iteration
T_solns = [np.copy(T)]

# define the transient model
Told = np.copy(T)
transient = FirstOrderTransientModel(mygrid,T,Told,rho,cp,dt)

# define the diffusion model
diffusion = DiffusionModel(mygrid, T, k, west_bc, east_bc)

# define the surface convection model
surfaceConvection = SurfaceConvectionModel(mygrid,T,ho,To)

# # # iterate until the solution is converged
for tStep in range(nTime):

    # update the time information
    time += dt

    # print the timestep information
    print("Timestep = {}; Time = {}".format(tStep,time))


    # store the old temperature field
    Told[:] = T[:]

    # iterate until converge
    for i in range(maxIter):

        # zero the coeff and add
        coeffs.zero()
```

```
        coeffs = diffusion.add(coeffs)
        coeffs = surfaceConvection.add(coeffs)

        # compute residual and check for convergence
        maxResid = norm(coeffs.rP, np.inf)
        avgResid = np.mean(np.absolute(coeffs.rP))
        print("Iteration = {} ; Resid = {} ; Avg. Resid = {}".format(i,maxResid,
    avgResid) )
        if maxResid < converged:
            break

        # solve the sparse matrix system
        dT = solve(coeffs)

        # update the solution and boundary conditions
        T[1:-1] += dT
        west_bc.apply()
        east_bc.apply()

    # store the solution
    T_solns.append(np.copy(T))



# plotting
i = 0
for Ti in T_solns:
    plt.plot(mygrid.xP, Ti, label = str(i))
    i += 1


plt.title('Implicit')
plt.xlabel('X')
plt.ylabel('T')
plt.savefig('pic/1d_transient_implicit.png')
plt.show()
```

4 ONE-DIMENSIONAL CONVECTION OF A SCALAR

4.1 Problem Definition

For thermal convection, we need the advection-diffusion equation. So far, we only dealt with diffusion, now we add the advection term which results in:

$$\frac{\partial(\rho c_p T)}{\partial t} + \nabla \cdot (\rho c_p \mathbf{u} T) = k\nabla^2 T + S \tag{83}$$

assuming constant ρ and c_p. We also assume that the flow field **u** is known, and we only use it to advect and solve for the temperature field. To preserve continuity across cells, we also define a mass conservation equation without mass source.

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \tag{84}$$

4.2 Discretization

Just like we did before, we now integrate the advection-diffusion equation through space and time:

$$\int_{t_0}^{t_1}\int_V \frac{\partial(\rho c_p T)}{\partial t} dt dV + \int_{t_0}^{t_1}\int_V \nabla \cdot (\rho c_p \mathbf{u} T) dt dV = \int_{t_0}^{t_1}\int_V k\nabla^2 T dV dt + \int_{t_0}^{t_1}\int_V S dV dt \tag{85}$$

Integration of the transient diffusion equation is covered. Here, we deal with the advection term. Using the Gauss' divergence theorem to convert volume integral to surface integral:

$$\int_V \nabla \cdot (\rho c_p \mathbf{u} T) dV = \int_S (\rho c_p \mathbf{u} T) \cdot \mathbf{n} dS \tag{86}$$

The surface integral is then approximated as discrete sum over the integration points

$$\int_S (\rho c_p \mathbf{u} T) \cdot \mathbf{n} dS = \sum_{i=0}^{N_{ip}-1} (\rho c_p \mathbf{u} T) \cdot \mathbf{n}_{ip} \mathbf{A}_{ip} \tag{87}$$

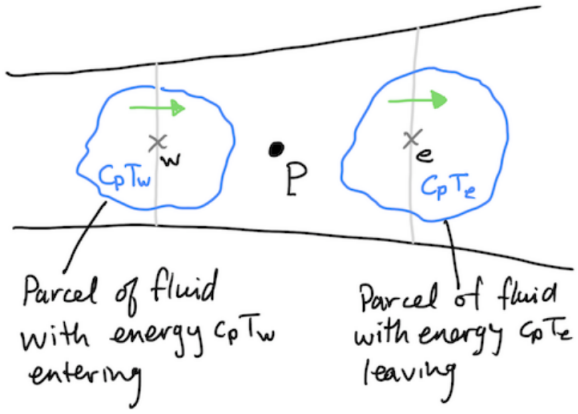
For 1D flow across control volume P, this results in:

$$\sum_{i=0}^{N_{ip}-1} (\rho c_p \mathbf{u} T) \cdot \mathbf{n}_{ip} \mathbf{A}_{ip} = \rho c_p u_e T_e A_e - \rho c_p u_w T_w A_w \tag{88}$$

Or in terms of the mass flux, $\dot{m} = \rho u A$:

$$\sum_{i=0}^{N_{ip}-1} (\rho c_p \mathbf{u} T) \cdot \mathbf{n}_{ip} \mathbf{A}_{ip} = \dot{m}_e c_p T_e - \dot{m}_w c_p T_w \tag{89}$$

Note how the $c_p T$ terms are similar to some forms of internal energy (enthalpy). Thus, we can think off the above equation as the difference in energy between 2 parcels of fluids, one with internal energy $c_p T_e$ and one with internal energy $c_p T_w$



As a result, our discretized energy equation becomes:

$$\frac{(\rho c_p T_P V_P)^{t+\Delta t/2} - (\rho c_p T_P V_P)^{t-\Delta t/2}}{\Delta t} + \dot{m}_e c_p T_e - \dot{m}_w c_p T_w = -D_w(T_P - T_W) + D_e(T_E - T_P) + S_P V_P \tag{90}$$

Note:

- transient term is like before, evaluated at time $t + \Delta t/2$ and $t - \Delta t/2$.
- our discretization scheme is **NOT** completed, because we still do not know how to calculate mass flux and temperature at integration points, namely $\dot{m}_e, \dot{m}_w$ and T_e, T_w.
- we must consider whether the given equation is independent of temperature level according to Rule 4. One can say that it has to be independent because transient, diffusion, advection terms involve only derivative of temperature. This is true, if mass is conserved ($\dot{m}_e = \dot{m}_w$). For 1D, this is easy to ensure. For multidimensional problems, this is difficult. In other words, we cannot assure that the numerical mass fluxes will always be conserved. This can lead to major problem, because if mass is not conserved, one may think that there is an energy source (or sink) within the domain.
- to get around the mass conservation problem, we substract the discretized mass equation from the energy equation. Assuming constant density:

  $$\dot{m}_e - \dot{m}_w = 0$$

  multiply this by T_P and C_P and subtracting from the discretized equation:

  $$\frac{(\rho c_p T_P V_P)^{t+\Delta t/2} - (\rho c_p T_P V_P)^{t-\Delta t/2}}{\Delta t} + \dot{m}_e c_p (T_e - T_P) - \dot{m}_w c_p (T_w - T_P) = -D_w(T_P - T_W) + D_e(T_E - T_P) + S_P V_P \tag{91}$$

  This means that if there is a positive imbalance of mass ($\dot{m}_e > \dot{m}_w$), there will be a negative source in the energy equation to counter balance. If there is a negative imbalance, the opposite is true. This step helps with the stability of the numerical method such that the equations are again independent of the temperature level.

4.3 Advection term with Explicit Time Integration

Assume we can interpolate the integration point values in the advection term using a piecewise linear approximation:

$$T_e = \frac{1}{2}(T_P + T_E)$$
$$T_w = \frac{1}{2}(T_W + T_P)$$

Assuming no source term and use an explicit time integration scheme, and keeping the T_P term arising from subtracting the mass conservation from the energy equation as implicit (i.e. at current timestep). We get the following discretized equation:

$$\frac{\rho c_p V_P (T_P - T_P^o)}{\Delta t} + \dot{m}_e c_p \left[\frac{1}{2}(T_P^o + T_E^o) - T_P\right] - \dot{m}_w c_p \left[\frac{1}{2}(T_W^o + T_P^o) - T_P\right]$$
$$= -D_w(T_P^o - T_W^o) + D_e(T_E^o - T_P^o)$$

where $'o'$ denotes values at previous timestep, and those without superscripts are for current timestep (i.e. those being solved). We can then group the terms according to their temperature: