Heat Simulation of a Single Heat Fin CPU Radiator

Pierre-Antoine LAMBRECHT/SENGER

February 15, 2024

UFR de mathématique et d'informatique
Université de Strasbourg

Contents

1 Compilation

Place yourself in the root of the Project.
Move or make a fresh build directory and move into it:

```
mkdir build
cd build
```

Compile an executable :

```
cmake --preset release ..
make
```

2 Usage

2.1 Run a Simulation

From the build directory do :

```
./run <config_file>
```

Example:

```
./run ../simul.cfg
```

You can modify or make your own **configuration files** as long as it's **exactly** in the same **format**:

```
Lx 40 Ly 4 Lz 50
Nx 10000
Phi 0.125
hc 0.0004
Te 20
rho 2700
kappa 164
stationary 0
cycling 0
fan 1
cooling 0
TFinal 300
Nt 600
Mx 100 My 20 Mz 60
doPlots 1 do3D 1
solName test1
```

Note: Writing into the files can take some time (with those parameters there are around $240,000 \times 600 = 144$ million small lines to write).

2.2 Visualization

For 2D solutions, you can choose in the `simul.cfg` if you want to see the plots or not. Additionally, if you want to see more plots and testings, you can do:

```
make test
```

Solutions are saved in `data`.
3D solutions can be visualized in a visualization software like `Paraview`.
Note: To do the plots, you will need the `pandas` library which can be installed using pip :

```
pip install pandas
```

or using conda/mamba:

```
conda install pandas
```

```
mamba install -c conda-forge pandas
```

Note: If the data directory gets too big, you can simply delete it; the save functions know how to create one without crashing.

3 Construction of the Project

The details of the classes are given in a `Doxygen` documentation. You can open it from the root of the project with the browser of your choice:

```
chromium html/index.html
```

3.1 Tridiag

```cpp
template <class T> class Tridiag {
public:
  Tridiag(std::size_t n = 1);
  Tridiag(const Tridiag &t);
  Tridiag(std::size_t n, T l, T d, T u);
  Tridiag &operator=(const Tridiag &t);
  ~Tridiag(){};

  //* Getters & Setters
  std::size_t size() const { return M_n; }
  T upper(int i) const { return M_upper[i]; }
  T diag(int i) const { return M_diag[i]; }
  T lower(int i) const { return M_lower[i]; }
  T &upper(int i) { return M_upper[i]; }
  T &diag(int i) { return M_diag[i]; }
  T &lower(int i) { return M_lower[i]; }
  T operator()(int i, int j) const;
  T &operator()(int i, int j);

  //* Arithmetic Operators & others methods
  Tridiag factorize() const;
  Tridiag &operator+=(const Tridiag &rhs);
  Tridiag &operator-=(const Tridiag &rhs);
  std::vector<T> operator*(const std::vector<T> &rhs) const;
  void clear(); ///< Clear the matrix vectors

  //* The following operators are defined outside the class :
  // Tridiag<T> operator+(const Tridiag<T> &lhs, const Tridiag<T> &rhs)
  // Tridiag<T> operator-(const Tridiag<T> &lhs, const Tridiag<T> &rhs
  // Tridiag<T> operator*(const double lhs, const Tridiag<T> &rhs)
  // Tridiag<T> operator*(const Tridiag<T> &lhs, const double rhs)
  // Tridiag<T> operator/(const Tridiag<T> &lhs, const double rhs)

  //* A method for solving LUx=b is defined outside the class:
  // void solveLU(std::vector<T> &x, const Tridiag<T> &LU, const std::vector<T>
  // &b)

  //* Ostream outside of the class
  // template <class U>
  // friend std::ostream &operator<<(std::ostream &os, const Tridiag<U> &t);

private:
  std::size_t M_n;        ///< Dimention of the square matrix
  std::vector<T> M_upper; ///< Vector of the upper diagonal part
  std::vector<T> M_diag;  ///< Vector of the main diagonal part
  std::vector<T> M_lower; ///< Vector of the lower diagonal part
};
```

The problem is modeled using a **banded matrix**, so we started by making a **template class** `Tridiag` representing a tridiagonal matrix using 3 `std::vector<class T>` representing the **upper**, **lower**, and **main diagonal** of the matrix.

The `Tridiag` class has well-defined accessors in writing/reading as well as an ostream operator. It also has some arithmetic operators defined left and right to make the code more readable. It could be improved by adding a matrix-vector multiplication, which can be useful in solving dynamic systems (I'll do it if I have the time and don't forget).

To efficiently solve a linear system, the class can factorize a `Tridiag` object A using an LU factorization: $A = LU$ where,

L is bidiagonal **lower** (with ones on the main diagonal like in every LU factorization),
U is bidiagonal **superior**.

Since there is no point storing the ones of L, the LU factorization is stored in a **single Tridiag** object where,
`Tridiag.lower` represent the lower diagonal of L,
`Tridiag.diag` represent the main diagonal of U,
`Tridiag.upper` represent the upper diagonal of U.

$$L = \begin{pmatrix} 1 & 0 & \cdots & \cdots & 0 \\ l_{21} & 1 & \ddots & & \vdots \\ 0 & l_{32} & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & l_{n,n-1} & 1 \end{pmatrix} \quad U = \begin{pmatrix} u_{11} & u_{12} & \cdots & \cdots & 0 \\ 0 & u_{22} & u_{23} & & \vdots \\ \vdots & \ddots & u_{33} & \ddots & \vdots \\ \vdots & & \ddots & \ddots & u_{n-1,n} \\ 0 & \cdots & \cdots & 0 & u_{nn} \end{pmatrix}$$

$$LU = \begin{pmatrix} u_{11} & u_{12} & 0 & \cdots & 0 \\ l_{21} & u_{22} & u_{23} & \ddots & \vdots \\ 0 & l_{32} & u_{33} & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & u_{n-1,n} \\ 0 & \cdots & 0 & l_{n,n-1} & u_{nn} \end{pmatrix}$$

The file `tridiag.hpp` also contains a function `solveLU` which take a LU matrix as described above and solve a **LUx=b** system with a time complexity in $O(n)$.

3.2 Model

```cpp
using dvector = std::vector<double>;

class Model {
public:
  Model(std::size_t Nx, double x0, double xend);
  Model(const Model &m);
  Model &operator=(const Model &m);
  virtual ~Model(){};

  virtual void setLU() = 0;
  virtual void setB(dvector, double, std::size_t, double) = 0;
  virtual void setU0(double u0);
  virtual void setXend(double d); ///< End of discretization
  virtual void setMesh(Mesh3D &mesh) { M_mesh = &mesh; }
  void setDt(double d); ///< Time step value

  virtual double b(std::size_t i) const;
  virtual double solveExact(double);
  std::size_t Nx() const;         ///< Numbers of space steps
  double x0() const;              ///< Start of discretization
  double xend() const;            ///< End of discretization
  double x(std::size_t i) const;  ///< Value in discretization
  double u(std::size_t i) const;  ///< Value in vector solution
  double dx() const;              ///< Space step value
  Tridiag<double> S();            ///< Matrix of the model
  dvector &u();                   ///< Returns whole solution
  dvector &b();                   ///< Returns whole RHS vector

protected:
  std::size_t M_Nx;    ///< Number of space steps
  double M_x0;         ///< Start of the discretization
  double M_xend;       ///< End of the discretization
  dvector M_x;         ///< Vector of the discretization
  dvector M_u;         ///< Vector of initial contition
  Tridiag<double> M_S; ///< Matrix of the model
  dvector M_b;         ///< RHS of the model
  double M_dt;          ///< Time step
  Mesh3D *M_mesh;       ///< Pointer on a mesh for 3d visualization
};
```

We then created a **abstract base class** `Model` to modelize the problem to solve.

This base class is supposed to put the foundation of what any model should have and should implement. It's an **abstract class** so if we want to solve other problems with similar methods we can just add new models and have a solver with a pointer to a base `Model`.

The most important member attributes are **Nx**, the number of space steps we're going to use in our discretization of the problem and will translate into the size of the matrix of the method **S** and the right-hand side vector **b** of the linear system modeling the problem.

This is why `Model` has 2 pure virtual methods `setLU` and `setB` to ensure all derived classes implement correctly the **tridiagonal matrix of the method** (and factorize it in its LU form right away) as well as **b**, the **RHS** of the system $Ax = b$.
Note that in order to be able to solve dynamic systems `setB` takes in argument the vector of current solution, the time (in case we are in a cycling model), the number of space steps `Nt` as well as the final time `tf`.

In order to make a 3D visualization of the solutions a model can also have a pointer to a **class** `Mesh3D` (I don't think we need more than 3D representation so there is no base class for it).

3.3 HeatFin

```
class HeatFin : public Model {
public:
  HeatFin(double nx, double rho, double c, double k, double Te, double phi,
          double hc, double Lx, double Ly, double Lz);
  HeatFin(const HeatFin &f);
  HeatFin &operator=(const HeatFin &f);
  ~HeatFin() {}

  void setLU() override;
  void setB(dvector u, double t, std::size_t nt, double tf) override;
  void fanON();
  void fanOFF();
  void fluxON();
  void fluxOFF();
  void CoolingON();
  void CoolingOFF();
  void setHc(double power);     ///< Fan power
  void setCooler(double temp); ///< Cooling power
  void setPhi(double d);       ///< Heat flux power
  void setLx(double d);        ///< Same as setXend() from base class
  void setCycling(bool b);     ///< Cycling status ON or OFF
  void setStationnary(bool b); ///< Dynamic or stationnary model

  void interpolate(dvector &U);
  void interpolate(std::vector<dvector> &T, std::size_t Nt,
                   const std::string solName);

  void saveStaticInterpolation(const std::string filename);

  double solveExact(double x) override;
  bool flux() const;     ///< Current status of the flux
  bool cooling() const; ///< Current status of the cooling
  double p() const;     ///< Perimeter
  double s() const;     ///< Surface
  double hc() const;    ///< Fan power

protected:
  double M_rho; ///< Density
  double M_C;   ///< Heat at constant pressure
  double M_k;   ///< Thermal conductivity
  double M_Te;  ///< External Temperature
  double M_phi; ///< Heat flux power
  double M_hc;  ///< Heat surfacic transfer coefficient
  double M_Ly;  ///< Size of the fin on the y-axis
  double M_Lz;  ///< Size of the fin on the z-axis
  ///< M_Lx = Model::M_xend
  double M_cooler;       ///< Cooling power effect (so <0)
  bool M_flux;           ///< Heat flux status
  bool M_cooling;        ///< Cooling status, why did I do this?
  bool M_cyclingStatus; ///< Status of the heat flux
  bool M_stationnary;   ///< Stationnary or Dynamic model
  dvector M_staticInterpolation;
  // Cooling need to be changed to accuratly describe what happens IRL
};
```

For our particular problem, we created a **derived class** of `Model` called `HeatFin` which fully describes the setups with the dimensions of the fin, the physical conditions, if we solve a stationary model or dynamic, with constant heat flux or cycling etc.

The `setLU` and `setB` methods are correctly overridden using the equations given in the subject. For the dynamic system,

$$\rho C_p \frac{T_i^{n+1} - T_i^n}{\Delta t} - \kappa \frac{T_{i-1}^{n+1} - 2T_i^{n+1} + T_{i+1}^{n+1}}{\Delta x^2} + \frac{h_c p}{S}(T_i^{n+1} - T_e) = 0 \tag{1}$$

gives us,

$$\left(\frac{\rho C_p}{\Delta t} + \frac{h_c p}{S}\right) I + \frac{\kappa}{\Delta x^2} A = BT^n + B' \tag{2}$$

with,

$$I = \begin{pmatrix} 0 & 0 & \cdots & \cdots & 0 \\ 0 & 1 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & 1 & 0 \\ 0 & \cdots & \cdots & 0 & 0 \end{pmatrix} \quad A = \begin{pmatrix} \Delta x & -\Delta x & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & \cdots & 0 & -\Delta x & \Delta x \end{pmatrix}$$

$$B = \begin{pmatrix} \Phi_p \\ \frac{\rho C_p}{\Delta t} \\ \vdots \\ 0 \end{pmatrix} \quad B' = \begin{pmatrix} 0 \\ \frac{T_e h_c p}{S} \\ \vdots \\ 0 \end{pmatrix}$$

- ρ : density
- C_p : specific heat at constant pressure
- κ : thermal conductivity
- h_c : surface heat transfer coefficient
- $S = L_y L_z$: cross-sectional area
- $p = 2(L_y + L_y)$: cross-sectional perimeter
- Φ_p : heat flux power

This class also has the `interpolate` (static and dynamic) methods which in the case of the static model take the vector of **solution** in argument to calculate the values for the 3D visualization and can be saved with another method `saveStaticInterpolation`.

In the case of the dynamic model, `interpolate` takes the vector of vectors containing all the **solutions in time** (one for each time step `Nt`) as well as the time step `Nt`, the **name** of the solution used in the saving of the file (the dynamic interpolation re-uses the `saveStaticInterpolation` for each time step). It would save disk space to directly write the interpolation data into the files instead of storing them in a vector but it's also less readable and also takes more time somehow (maybe because the file is open and adds charges on the CPU?)

3.4 SolverTime

```cpp
using dvector = std::vector<double>;

class SolverTime {
public:
  SolverTime(std::size_t Nt = 0, double tf = 1);
  SolverTime(const SolverTime &s);
  SolverTime &operator=(const SolverTime &s);
  ~SolverTime();

  double dt() const; ///< Get the time step value

  void setTfinal(double t);   ///< Sets the final time value
  void setNt(std::size_t nt); ///< Sets the number of time steps
  void setModel(Model &m);    ///< Sets a pointer to a Model
  void solveStatic();         ///< Solves a stationary Model
  void solve();               ///< Solves a dynamic Model
  void solve(double tFinal);

  std::vector<dvector> &T() { return M_T; }
  dvector &U() { return M_U; }
  void saveStatic(const std::string filename, const bool do_plot = false);
  void saveAtTimes(const std::string filename, dvector times,
                 const bool do_plot = false);
  template <typename... Args>
  void saveAtTimes(const std::string &filename, bool do_plot, Args... args);
  void saveAtPoints(const std::string filename,
                  const std::vector<double> points,
                  const bool do_plot = false);
  template <typename... Args>
  void saveAtPoints(const std::string &filename, bool do_plot, Args... args);

private:
  std::size_t M_Nt;          ///< Number of time steps
  double M_Tfinal;           ///< Final time
  std::vector<dvector> M_T;  ///< Solutions in time
  dvector M_xt;              ///< Time discretization
  Model *M_model;            ///< Pointer to a model to solve
};
```

Finally, to solve the problem, we created a **class** `SolverTime` which is defined by the number of time steps `Nt`, the final time `Tfinal`, the vector of time discretization `xt`, the vector of all solutions in time `T`, and a pointer to a base class `Model` to solve.

As most of the information is stored in the model, `SolverTime` doesn't require a lot of methods. The main ones are, of course, the solve methods with `solveStatic` which solve a **stationary** model and `solve` which solves a **dynamic** model and will use the final time set in the `SolverTime` object or can take a final time as an argument.

The class also has multiple save methods to **save and plot** the 2D results (which took me a lot of time to implement properly...)
Each saves methods has a boolean argument to do the plot or only the data saving (which are stored in `data/2d`).

`saveStatic` saves the stationary model with the **exact solution** next to it for comparison.

`saveAtTimes` saves at a particular time given in **seconds** by a vector, or an arbitrary number of times (variadic function) which is converted to the (approximated) index in the time discretization. Then plots all of them in one figure.

`saveAtPoints` works similarly but with **point.s** on the physical model instead of **time.s**.

All the save methods "simply" look for the correct solutions to save; they don't require redoing the computation.
Each save method has a dedicated python plotting file stored in the `plotting` directory which

contains all the necessary instructions to realize the plots using `pandas` and `matplotlib`.

This class could be improved by specifying the particular times we are interested in before solving the model so we could only store those instead of all the solutions in times thus saving a lot of spaces on the disk.

3.5 Mesh

```cpp
class Mesh3D {
public:
  Mesh3D(double Lx, double Ly, double Lz, int x, int y, int z)
      : M_Lxyz(3), M_Mxyz(3), M_x(x + 1), M_y(y + 1), M_z(z + 1) {
    setLxyz(Lx, Ly, Lz);
    setMxyz(x, y, z);

    for (std::size_t i = 0; i <= Mx(); ++i)
      M_x[i] = i * dx();
    for (std::size_t i = 0; i <= My(); ++i)
      M_y[i] = i * dy();
    for (std::size_t i = 0; i <= Mz(); ++i)
      M_z[i] = i * dz();
  }
  double Lxyz(int i) const { return M_Lxyz[i]; }
  double &Lxyz(int i) { return M_Lxyz[i]; }
  std::size_t Mxyz(int i) const { return M_Mxyz[i]; }
  std::size_t &Mxyz(int i) { return M_Mxyz[i]; }
  double Lx() const { return M_Lxyz[0]; }
  double Ly() const { return M_Lxyz[1]; }
  double Lz() const { return M_Lxyz[2]; }
  double x(int i) const { return M_x[i]; }
  double y(int i) const { return M_y[i]; }
  double z(int i) const { return M_z[i]; }
  double &x(int i) { return M_x[i]; }
  double &y(int i) { return M_y[i]; }
  double &z(int i) { return M_z[i]; }
  std::size_t Mx() const { return M_Mxyz[0]; }
  std::size_t My() const { return M_Mxyz[1]; }
  std::size_t Mz() const { return M_Mxyz[2]; }
  std::size_t &Mx() { return M_Mxyz[0]; }
  std::size_t &My() { return M_Mxyz[1]; }
  std::size_t &Mz() { return M_Mxyz[2]; }
  void setLxyz(double x, double y, double z) {
    M_Lxyz[0] = x;
    M_Lxyz[1] = y;
    M_Lxyz[z] = z;
  }
  void setMxyz(int x, int y, int z) {
    M_Mxyz[0] = x;
    M_Mxyz[1] = y;
    M_Mxyz[2] = z;
  }
  double dx() const { return Lx() / Mx(); }
  double dy() const { return Ly() / My(); }
  double dz() const { return Lz() / Mz(); }
  dvector x_ijk(int i, int j, int k) const {
    dvector res(3);
    res[0] = x(i);
    res[1] = y(j);
    res[2] = z(k);
    return res;
  }

private:
  dvector M_Lxyz;     ///< Dimention of the model to mesh
  sizeVector M_Mxyz; ///< Number of steps in each dimention
  dvector M_x;        ///< Discretization in the x-axis
  dvector M_y;        ///< Discretization in the y-axis
  dvector M_z;        ///< Discretization in the z-axis
};
```

I made the **class** `Mesh` mechanically while reading the subject and by doing so implementing a lot of member attributes, and methods which are in the end not needed for our problem. I chose to keep them nonetheless in case we need it for future models. If not they can be removed to optimize memory usage.

3.6 SimulParam

```cpp
struct SimulParam {
  // This contructor exist solely to silence -Weffc++ ...
  SimulParam()
      : Lx(), Ly(), Lz(), Phi(), hc(), Te(), tFinal(), rho(), kappa(), Nx(),
        Nt(), Mx(), My(), Mz(), stationary(), cycling(), doPlots(), fan(),
        cooling(), do3D(), solName() {}

  double Lx, Ly, Lz, Phi, hc, Te, tFinal, rho, kappa;
  std::size_t Nx, Nt, Mx, My, Mz;
  bool stationary, cycling, doPlots, fan, cooling, do3D;
  std::string solName;

  friend std::ostream &operator<<(std::ostream &os, const SimulParam &params);
  friend std::istream &operator>>(std::istream &is, SimulParam &params);
};
```

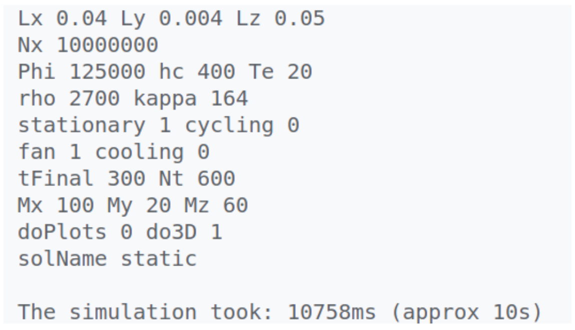
In order to avoid recompiling for each new simulation, a `SimulParam` struct is used to read and store the **configuration** of the simulation in an object given by the user in the configuration file. (This could maybe be done using a `std::map` instead)

4 Results

There is an infinity possible simulations, but it will also take an infinity amount of time and resources so let's look at only some of them.

4.1 Convergence of the dynamic model

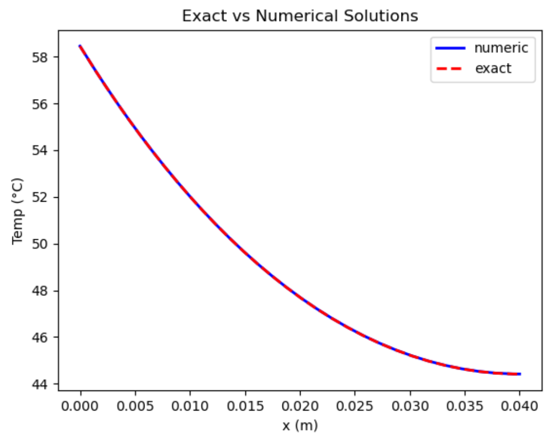
We want to confirm that the dynamic model converges to the stationary model. We're going to compare the results of both simulations using same physical and geometrical parameters as in the subject :



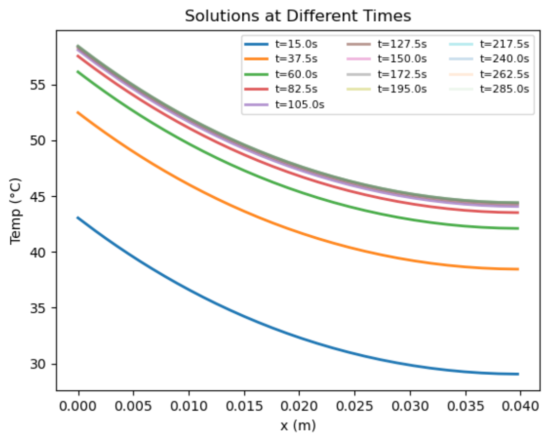
```
Lx 0.04 Ly 0.004 Lz 0.05
Nx 10000000
Phi 125000 hc 400 Te 20
rho 2700 kappa 164
stationary 1 cycling 0
fan 1 cooling 0
tFinal 300 Nt 600
Mx 100 My 20 Mz 60
doPlots 0 do3D 1
solName static

The simulation took: 10758ms (approx 10s)
```

Figure 1: Parameters



(a) Stationary

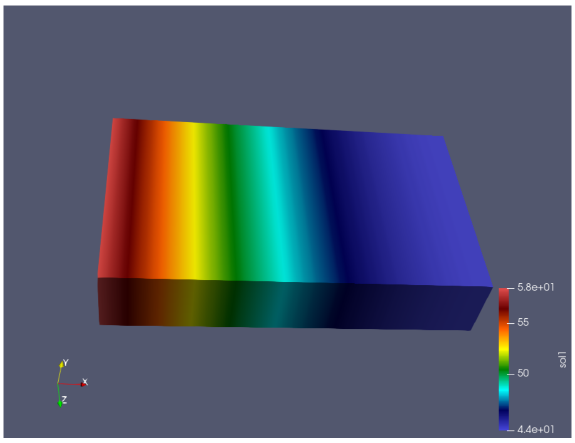


(b) Dynamic

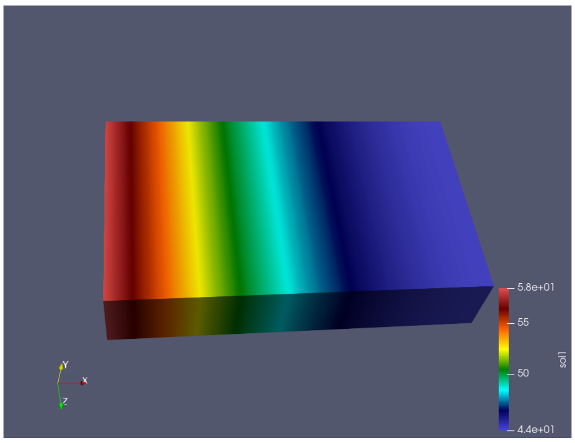
Figure 2: Stationary vs Dynamic

As we can see the dynamic model indeed converges to the stationary one in about 200s . (The stationary solution is confirmed by the exact solution available for this model).

We can take a look at the 3d visualization for both cases :



(a) Stationary

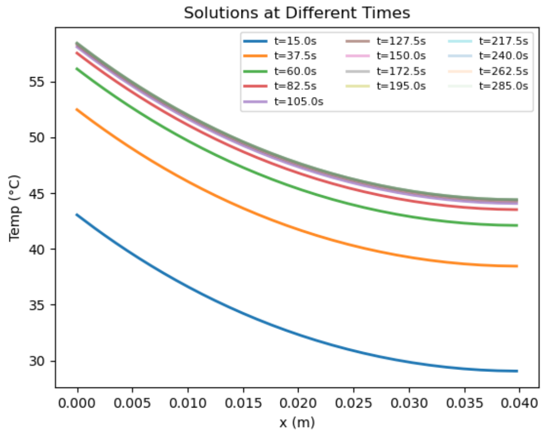


(b) Dynamic after 200s

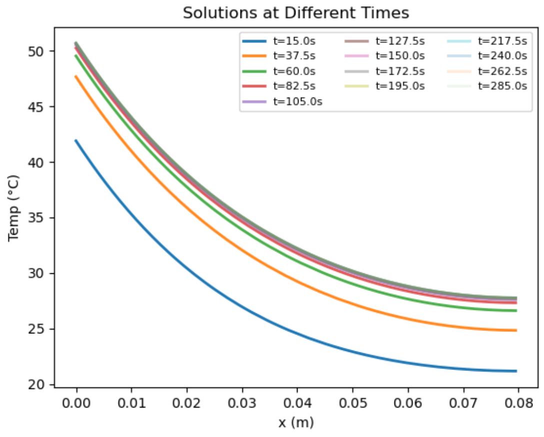
Figure 3: Stationary vs Dynamic

4.2 Effect of Fin Length

We're going to look at the difference between a 40mm length fin versus a 80mm one, the other physical parameters stays the same.

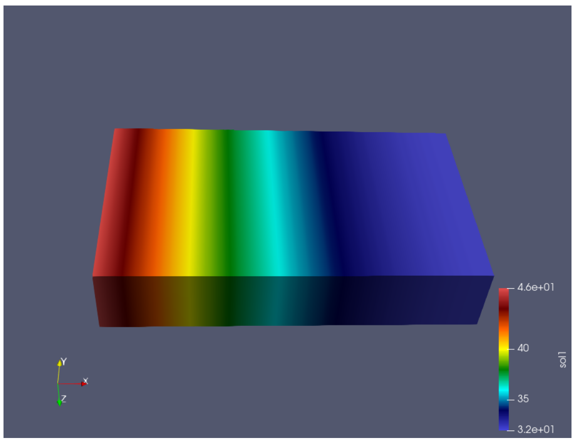


(a) $L_x = 40mm$

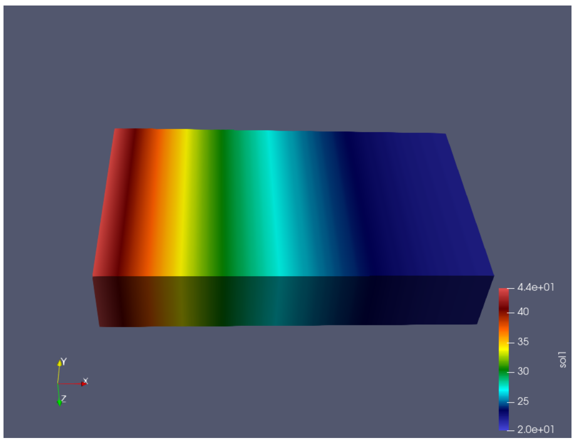


(b) $L_x = 80mm$

Figure 4: $L_x = 40mm$ vs $L_x = 80mm$

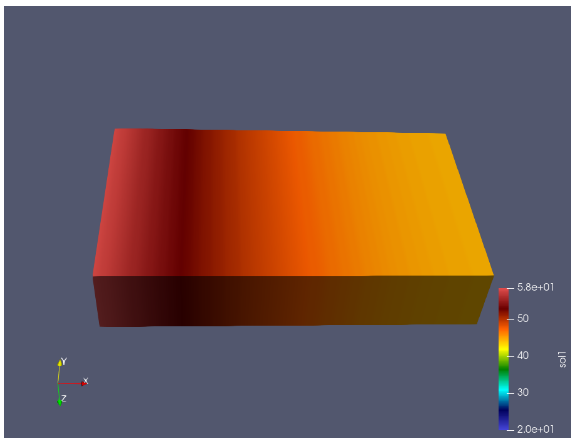


(a) $L_x = 40mm$

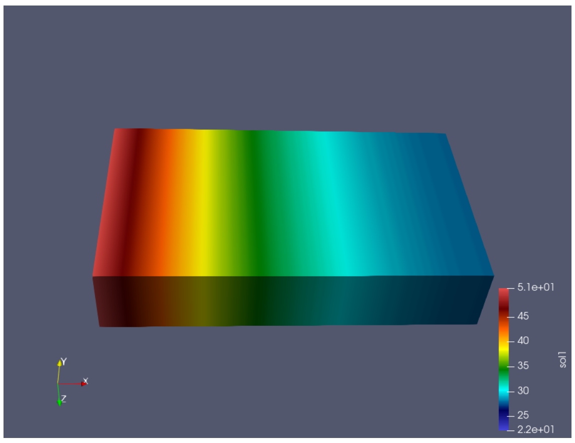


(b) $L_x = 80mm$

Figure 5: $L_x = 40mm$ vs $L_x = 80mm$ after 20s



(a) $L_x = 40mm$



(b) $L_x = 80mm$

Figure 6: $L_x = 40mm$ vs $L_x = 80mm$ after 200s

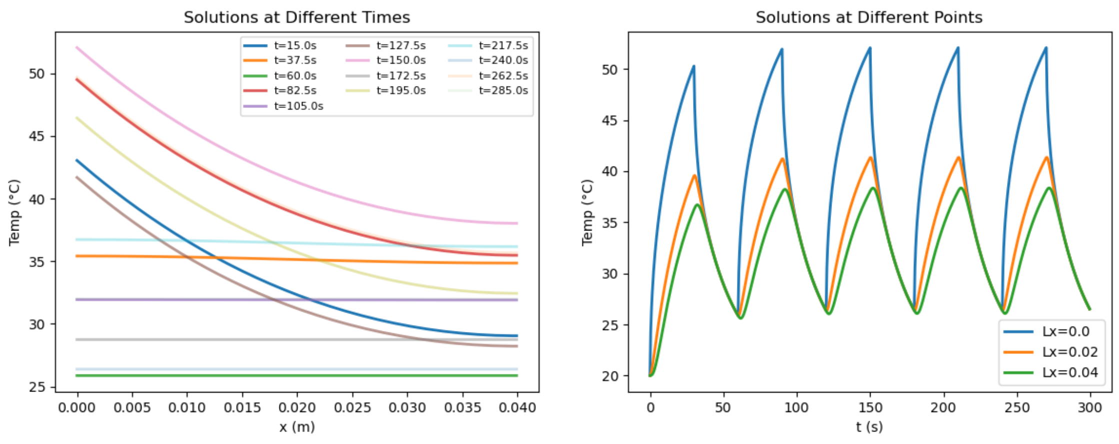
When it comes to radiator heat fin, size does matter. Already after 20 seconds, differences become apparent, especially at both ends.

After approximately 200 seconds, both models appear stable, with a noticeable temperature difference between them. Even at the point of contact, the 80mm fin is approximately 5°C cooler. This suggests good thermal conductivity for the material.

However, it's worth noting that the 40mm model doesn't exceed 60°C, which might be acceptable depending on the specific situation and potential temperature variations with other parameters.

4.3 Cycling Heat Flux Scenarios

Let's now look at a configuration with the fan ON and a cycling heat flux (30s ON 30s OFF) on a 40mm heat fin.



(a) (b)

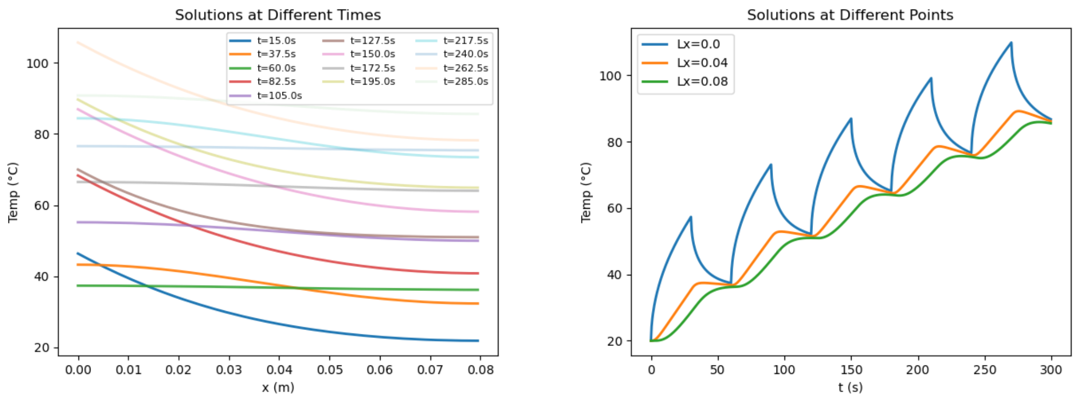
Figure 7: Cycling heat flux

As we can see, the temperature in the fin rapidly decreases to approximately 26°C when the heat flux is turned off, even at the point of contact with the CPU. This reinforces our hypothesis regarding the thermal conductivity of the material. (This is also a very satisfying 3d simulation to watch).

Note that some solution appears almost constant which is not too surprising considering we are in a cycling model.

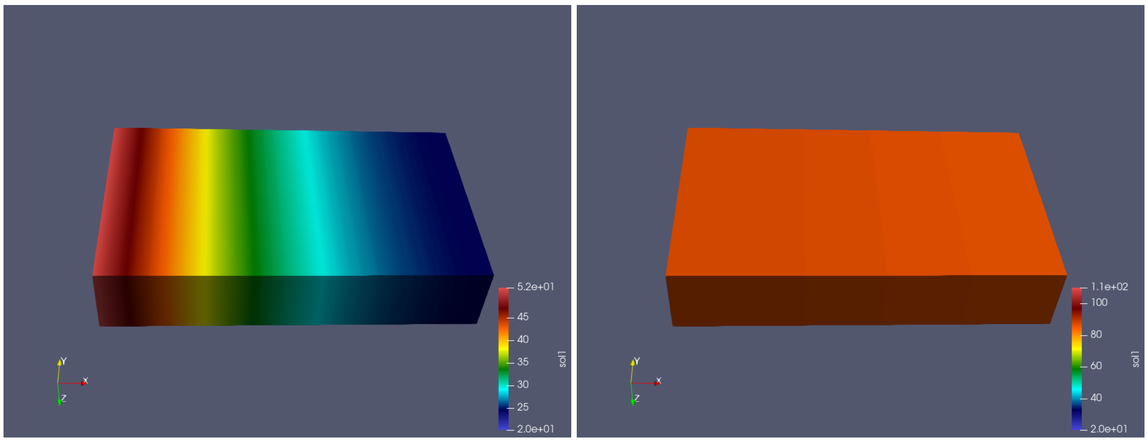
4.4 Impact of Fan Operation

What would happens in the case of a fan malfunction ?



(a) (b)

Figure 8: Cycling heat flux without fan



(a) after 20s (b) after 300s

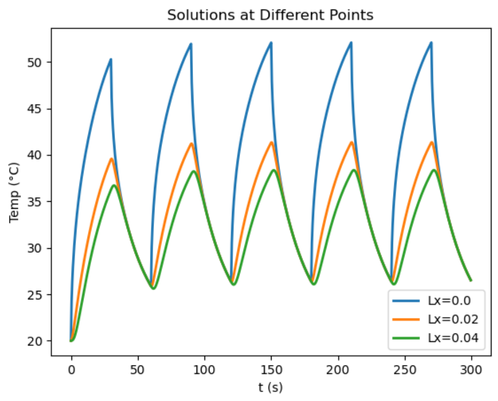
Figure 9: Cycling heat flux without fan

The temperature definitely goes down when the flux is off but not enough and the heat keeps building up in the material reaching more than 100°C at the end of our 300s simulation.

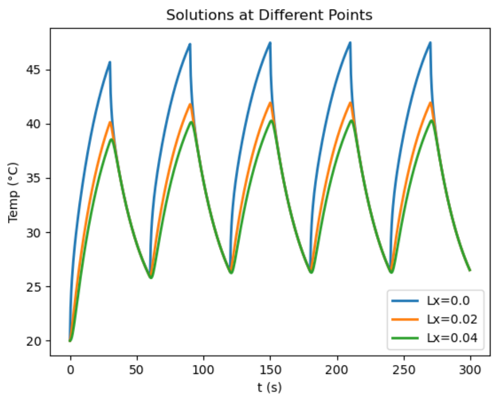
Clearly, even with a cycling heat flux and an 80mm fin, this configuration is not viable. Without an automatic emergency shutdown procedure, the CPU could be irreversibly damaged.

4.5 Effect of Thermal Conductivity

Finally let's see what would happen if we could double the thermal conductivity κ of the heat fin material going from 164 to 328.



(a) $\kappa = 164$



(b) $\kappa = 328$

Figure 10: Different thermal conductivity

We can notice a difference but not that significant, the fin cool down better and faster but also heat up better and faster. The improvement is probably not worth it for this context of use.

5 Conclusion

In this project, we developed a comprehensive simulation framework to analyze the thermal behavior of a single heat fin CPU radiator. The implementation includes a flexible and extensible structure, allowing for the modeling of various heat transfer scenarios.

Our approach involved the creation of a `Tridiag` template class to handle the banded matrix operations efficiently. The `Model` base class served as the foundation for specific heat transfer models, with the `HeatFin` class tailored to the particulars of the problem, including dynamic and stationary simulations.

The `SolverTime` class provided a systematic way to solve both static and dynamic models, offering options for saving and visualizing results at different time points or spatial locations.

The simulation results demonstrated the impact of various parameters on the temperature distribution within the heat fin. The comparison of different fin lengths, cycling heat flux with and without a fan, and variations in thermal conductivity showcased the versatility of our framework.

Our findings highlighted the significance of fin dimensions, cycling heat flux dynamics, and material properties in determining the effectiveness of the heat fin. The 3D visualizations provided valuable insights into temperature variations over time and space.