1 Code development

Explicit Method

According to the problem, we know that:

$$\rho c\frac{\partial u}{\partial t}-\kappa\frac{\partial^2 u}{\partial x^2}=\sin(l\pi x)$$

$$\Rightarrow \frac{\partial u}{\partial t}-\frac{\kappa}{\rho c}\frac{\partial^2 u}{\partial x^2}=\frac{\sin(l\pi x)}{\rho c}$$

Applying the explicit Euler method and central difference method, we obtain:

$$\frac{u_i^{n+1}-u_i^n}{\Delta t}-\frac{\kappa}{\rho c}\frac{u_{i+1}^n-2u_i^n+u_{i-1}^n}{\Delta x^2}=\frac{\sin(l\pi x)}{\rho c}$$

$$\Rightarrow u_i^{n+1}-u_i^n-\frac{\kappa\Delta t}{\rho c\Delta x^2}\left(u_{i+1}^n-2u_i^n+u_{i-1}^n\right)=\frac{\Delta t\bullet\sin(l\pi x)}{\rho c}$$

$$\Rightarrow u_i^{n+1}=\frac{\kappa\Delta t}{\rho c\Delta x^2}u_{i+1}^n+\left(1-\frac{2\kappa\Delta t}{\rho c\Delta x^2}\right)u_i^n+\frac{\kappa\Delta t}{\rho c\Delta x^2}u_{i-1}^n+\frac{\Delta t\bullet\sin(l\pi x)}{\rho c}$$

Let: $$\alpha=\frac{\kappa}{\rho c},\qquad \beta=\frac{\alpha\Delta t}{\Delta x^2}$$

The equation can be rewrite as:

$$u_i^{n+1}=\beta u_{i+1}^n+(1-\beta)u_i^n+\beta u_{i-1}^n+\alpha\bullet\Delta t\sin(l\pi x)$$

Implicit Method

According to the problem, we know that:

$$\rho c\frac{\partial u}{\partial t}-\kappa\frac{\partial^2 u}{\partial x^2}=\sin(l\pi x)$$

$$\Rightarrow \frac{\partial u}{\partial t}-\frac{\kappa}{\rho c}\frac{\partial^2 u}{\partial x^2}=\frac{\sin(l\pi x)}{\rho c}$$

Applying the implicit Euler method and central difference method, we obtain:

$$\frac{u_i^{n+1}-u_i^n}{\Delta t}-\frac{\kappa}{\rho c}\frac{u_{i+1}^{n+1}-2u_i^{n+1}+u_{i-1}^{n+1}}{\Delta x^2}=\frac{\sin(l\pi x)}{\rho c}$$

Let: $$\alpha=\frac{\kappa}{\rho c},\qquad \beta=\frac{\alpha\Delta t}{\Delta x^2}$$

$$u_i^{n+1}-\beta\left(u_{i+1}^{n+1}-2u_i^{n+1}+u_{i-1}^{n+1}\right)=\alpha\bullet\Delta t\bullet\sin(l\pi x)+u_i^n$$

$$-\beta u_{i-1}^{n+1}+(1+2\beta)u_i^{n+1}-\beta u_{i+1}^{n+1}=u_i^n+\alpha\bullet\Delta t\bullet\sin(l\pi x)$$

$$\begin{bmatrix} 1+2\beta & -\beta & \cdots & & & \\ -\beta & 1+2\beta & -\beta & \cdots & & \\ \vdots & -\beta & 1+2\beta & -\beta & \cdots & \\ & \vdots & \ddots & \ddots & \ddots & \\ & & & -\beta & 1+2\beta & -\beta \\ & & & & -\beta & 1+2\beta \end{bmatrix} \begin{bmatrix} u_2^{n+1} \\ u_3^{n+1} \\ \vdots \\ \vdots \\ u_{n-1}^{n+1} \\ u_n^{n+1} \end{bmatrix} = \begin{bmatrix} u_2^n+\alpha\bullet\Delta t\bullet\sin(l\pi(2-1)\Delta x) \\ u_3^n+\alpha\bullet\Delta t\bullet\sin(l\pi(3-1)\Delta x) \\ \vdots \\ \vdots \\ \vdots \\ u_3^n+\alpha\bullet\Delta t\bullet\sin(l\pi(n-1)\Delta x) \end{bmatrix}$$

The explicit Euler folder is using explicit method to solve the equation and the implicit Euler folder is using implicit method to solve the implicit equation. The necessary data such as dx, dt, etc are store in file explicit_heat.h5. The folder also contains Makefile and source code etc.

We use HDF5 to enable a restart facility, we run the code from 0 to 2 second, and then stop it when arrived 0.1 second then restart the program at 0.1second, read the data we have already save and keep running until 2 second. The result as figure 1. Left column shows when it arrived 0.1 second and middle column shows restart from 0.1 second and calculate until 2 second. The right column shows non-restart simulation. So compare the middle and right column, we can see that the restart implementation is correct.

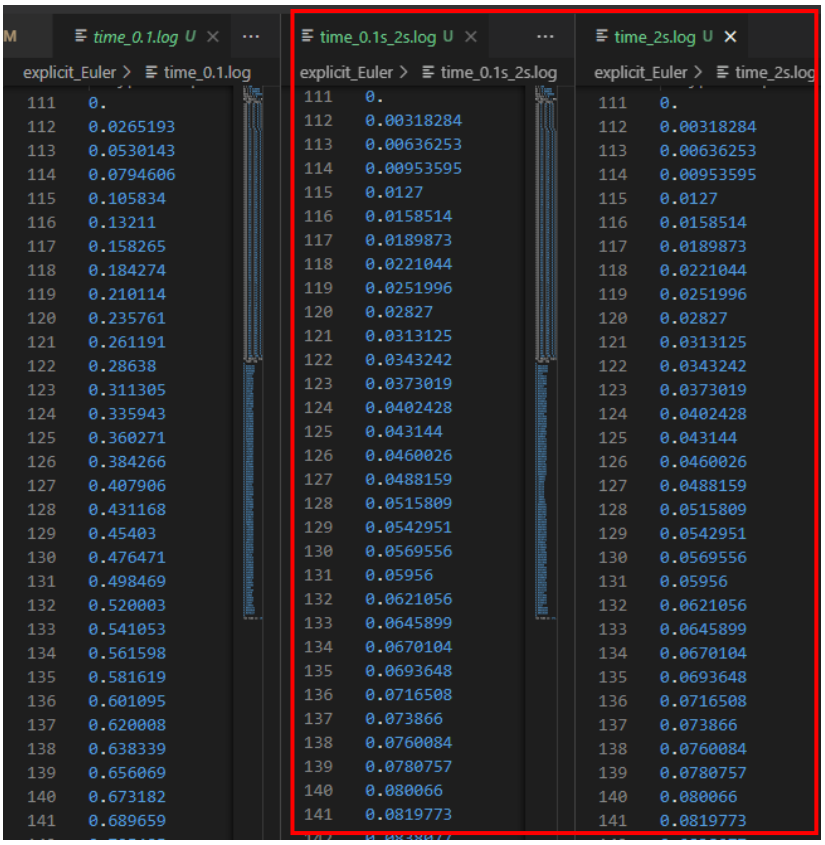


Figure 1. result of restart and non-restart

2 Code testing

2.1 Method stability

When reach the steady state, we know:

$$\frac{\partial u}{\partial t} = 0$$

$$\Rightarrow \frac{\partial^2 u}{\partial x^2} = -\frac{1}{\kappa}\sin(l\pi x)$$

$$\Rightarrow u = \frac{\sin(l\pi x)}{\kappa l^2 \pi^2} + C_1 x + C_2$$

We know the boundary condition:

$$u(0,t) = u(1,t) = 0$$

Substitute into the equation, we can obtain the steady **analytical form**:

$$\Rightarrow u = \frac{\sin(l\pi x)}{l^2 \pi^2} + \frac{\sin(l\pi)}{l^2 \pi^2} x$$

Stability analysis of explicit method:

The error equation is:

$$\delta u_i^{n+1} = \beta \delta u_{i+1}^n + (1-2\beta)\delta u_i^n + \beta \delta u_{i-1}^n$$

For Neumann stability: $\delta u_i^n \sim e^{\sigma n \Delta t} \bullet e^{i(k \bullet j\Delta x)}$

Therefore:

$$\begin{aligned} e^{\sigma \Delta t} &= \beta \bullet e^{ik\Delta x} + (1-2\beta) + \beta e^{-ik\Delta x} \\ &= (1-2\beta) + \beta\left(e^{ik\Delta x} + e^{-ik\Delta x}\right) \\ &= 1 - 2\beta + 2\beta\cos(k\Delta x) \end{aligned}$$

$$\left|e^{\sigma \Delta t}\right| \le 1 \quad \Rightarrow \quad -1 \le 1 - 4\beta \le 1 \quad \Rightarrow \quad 0 \le \beta \le \frac{1}{2}$$

Stability analysis of implicit method:

The error equation is:

$$-\beta \delta u_{i-1}^{n+1} + (1+2\beta)\delta u_i^{n+1} - \beta \delta u_{i+1}^{n+1} = \delta u_i^n$$

$$\Rightarrow -\beta e^{\sigma \Delta t} \bullet e^{-ik\Delta x} + (1+2\beta)\delta e^{\sigma \Delta t} - \beta e^{\sigma \Delta t} \bullet e^{ik\Delta x} = 1$$

$$\Rightarrow e^{\sigma \Delta t}\left[\beta e^{ik\Delta x} + \beta e^{-ik\Delta x} - (1+2\beta)\right] = -1$$

$$\Rightarrow \left|e^{\sigma \Delta t}\right| = \left|-\frac{1}{2\beta\cos(k\Delta x) - (1+2\beta)}\right| = \left|\frac{1}{1+2\beta\left[1-\cos(k\Delta x)\right]}\right| \le 1$$

So, for implicit method, it is unconditional stable.

From the above analysis, we know that for explicit method, the stable condition is CFL no large than 0.5. So we set dx=0.01, let it develop from 0 to 3 second and change the dt, set the CFL as 0.1, 0.49, 0.50, 0.51 to see it is stable or not. The result as figure 2. From the figure we can see that when CFL=0.51, the result is infinite which show it is unstable. So, the numerical example is consistence with the theoretical analysis. And to obtain the stable calculations result, the maximum step size is 0.00005.

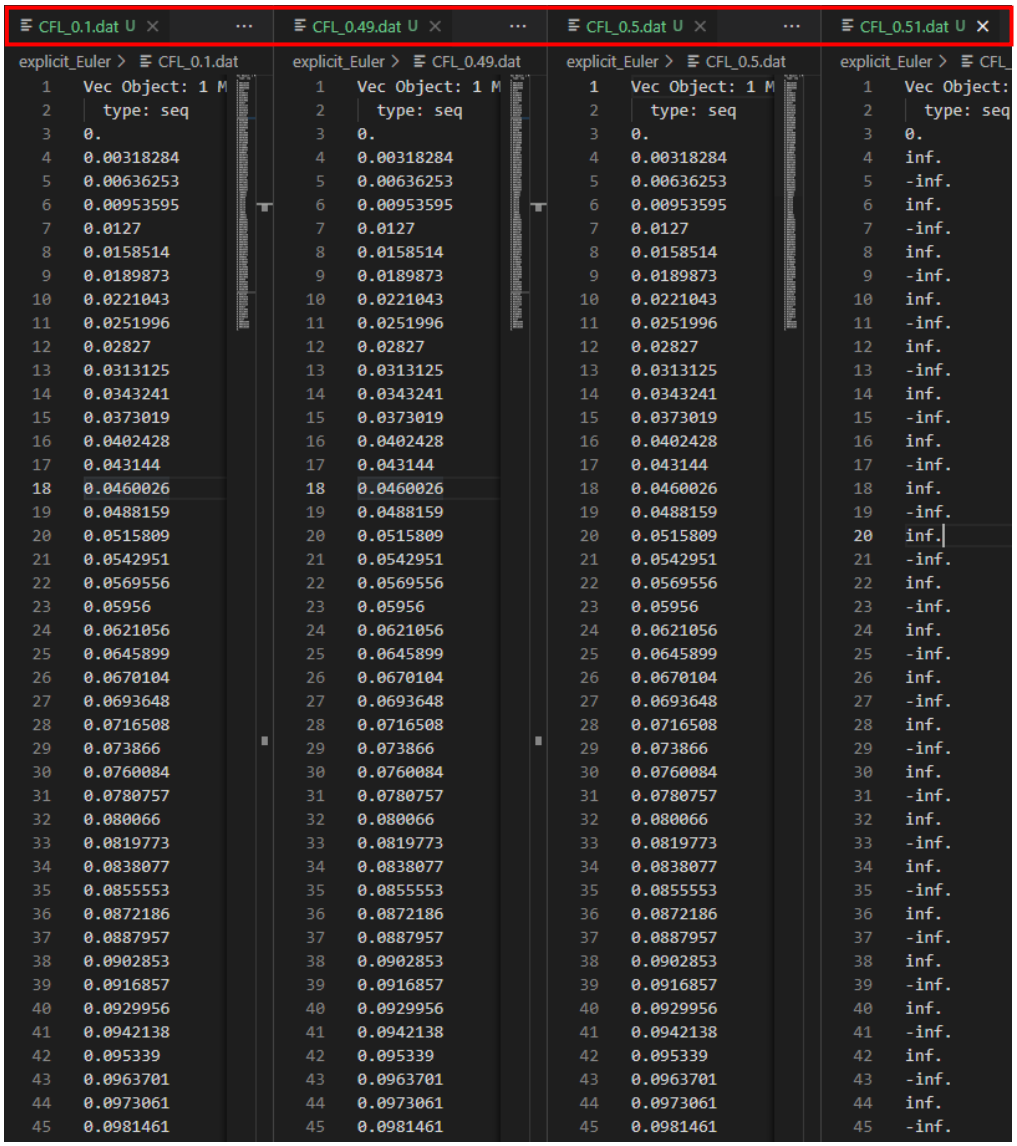
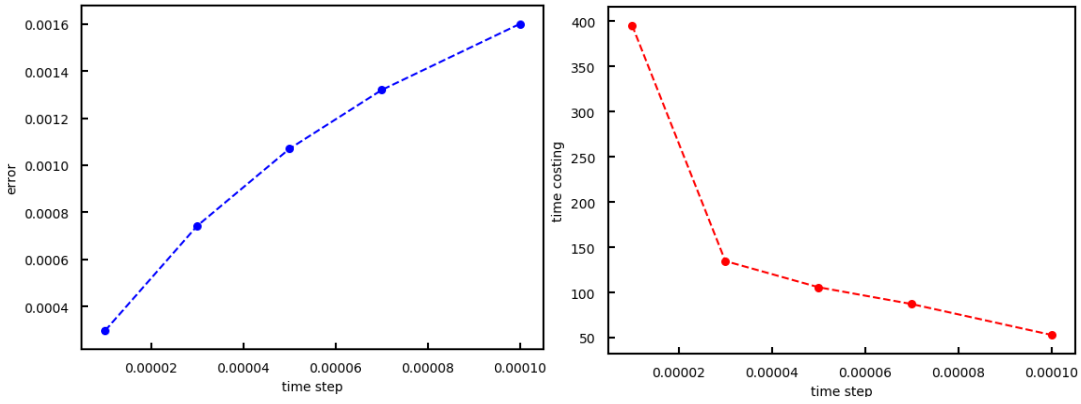


Figure 2. result of different CFL number. First column is 0.1, second is 0.49, third is 0.50, last is 0.51

From the above theoretical analysis, we know the implicit method is unconditional stable. So we set the CFL as 0.1, 0.3, 0.5 ,0.7 and 1.0 to see how will the time step influence the calculation speed. As the dx=0.01 so the corresponding dt are: 1e-5, 3e-5, 5e-5, 7e-5, 1e-4. So we can obtain the result as table 1.

Table 1. variable influenced by time step in implicit method

Time step	CFL	Maximum error(compare with analytical solution)	Running time
1e-5	0.1	0.0002978163576622256	395.6
3e-5	0.3	0.0007428163576622265	134.7
5e-5	0.5	0.0010697505689575282	105.7
7e-5	0.7	0.0013192482611641965	87.08
1e-4	1.0	0.0015999585439685354	52.91



(a) (b)

Figure 3. (a)relation between time step and maximum error (b) relation between time step and time cost

From the figure 3 we can know that for implicit method, the error will increase with the increasement of time step. And the time costing will decrease with the increase of time step which also means the speed will increase with the increasement of the time step.

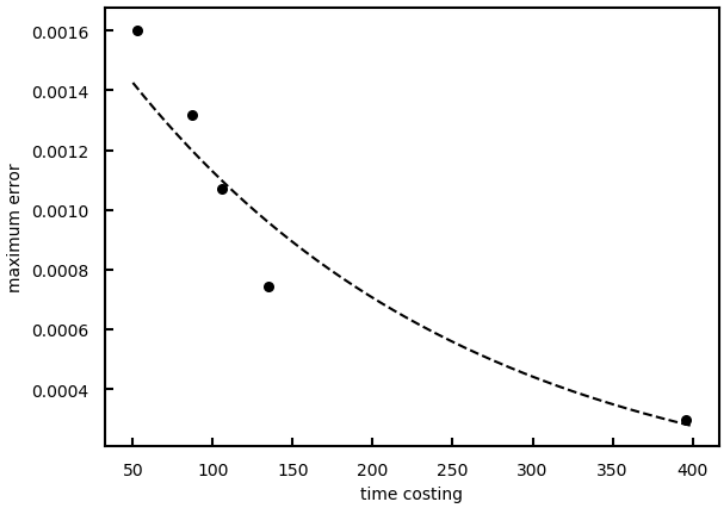
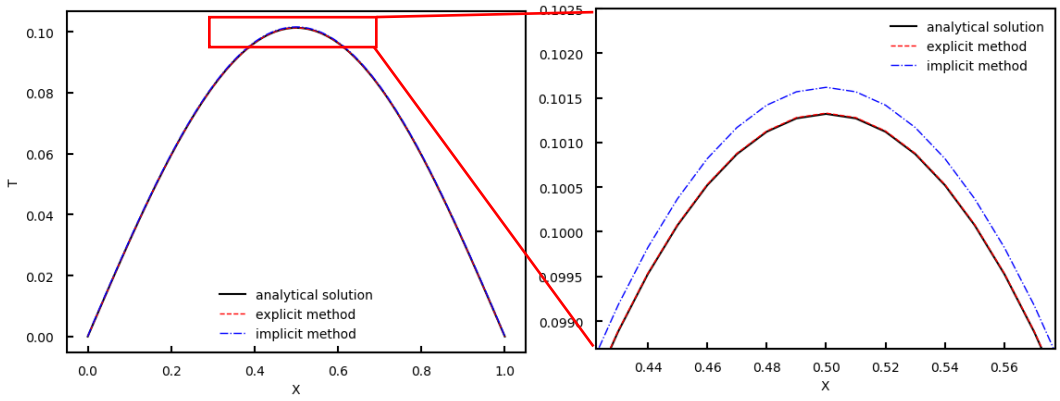


Figure 4. fitting of time costing and maximum error

From the figure 4, we can see that the maximum error will decrease with the increase of time costing, because the decrease of time step will lead to spend more time and also lead to minimize the error. In above figure, we use power functions to

approximate data points and the points underline show it have smaller error at the same time cost. So in order to explore a low convergence tolerance and larger time steps, we can use this fitting line as reference. In here, the point CFL=0.3, time step=3e-5, error=0.0007428163576622265, running time=134.7s is a good choice.



(a) (b)

Figure 5. (a) Comparison of analytical solution and numerical solution. (b) zoom in figure of (a)

It can be seen from the figure 5 that When CFL=0.1, dx=0.01, dt=1e-5, explicit is more accuracy than the implicit. This is just a small demo of visualization, we can plot more status figures like this.

There is a explicit_CFL0.1_valgrind.log in the explicit_Euler folder and the same that of in the implicit_Euler folder. It is shown in figure 5, we can see that it will still loss some bytes in a block, but it is no much. it may cause by some library functions.

```
==294402==
==294402== HEAP SUMMARY:
==294402==     in use at exit: 54,884 bytes in 1,223 blocks
==294402==   total heap usage: 3,408 allocs, 2,185 frees, 141,147 bytes allocated
==294402==
==294402== LEAK SUMMARY:
==294402==    definitely lost: 10 bytes in 1 blocks
==294402==    indirectly lost: 0 bytes in 0 blocks
==294402==      possibly lost: 0 bytes in 0 blocks
==294402==    still reachable: 54,874 bytes in 1,222 blocks
==294402==         suppressed: 0 bytes in 0 blocks
==294402== Rerun with --leak-check=full to see details of leaked memory
==294402==
==294402== For counts of detected and suppressed errors, rerun with: -v
==294402== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==293284==
==293284== HEAP SUMMARY:
==293284==     in use at exit: 58,325 bytes in 1,208 blocks
==293284==   total heap usage: 3,691 allocs, 2,483 frees, 147,299 bytes allocated
==293284==
==293284== LEAK SUMMARY:
==293284==    definitely lost: 0 bytes in 0 blocks
==293284==    indirectly lost: 0 bytes in 0 blocks
==293284==      possibly lost: 0 bytes in 0 blocks
==293284==    still reachable: 58,325 bytes in 1,208 blocks
==293284==         suppressed: 0 bytes in 0 blocks
==293284== Rerun with --leak-check=full to see details of leaked memory
==293284==
==293284== For counts of detected and suppressed errors, rerun with: -v
==293284== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 6. Valgrind.log for explicit method at CFL=0.1

2.2 Manufactured solution method

For explicit method, Fix the Δx and change the Δt, we find that the error is irrelevant of time. Then we choose Δt=1e-6, it is very fine temporal mesh to rule out the temporal error source. We chose the grid numbers as 100, 200, 300, 400, 500, and see the error each of them, so we can get table 2.

Table 2. error change with Δx at Δt=1e-6

N	Δx	Maximum error
100	0.01	8.816357662227992e-06
200	0.005	2.4833571247440123e-06
300	0.00333	1.3718624483904929e-06
400	0.0025	9.413415984482754e-07
500	0.002	8.163576622199908e-07

According to $\log(e) = \alpha \log(\Delta x) + \log(C_1)$ we can approximate α =1.5091, $\log(C_1)$ = -2.0813, $\log(e) = 1.5091 \times \log(\Delta x) - 2.0813$

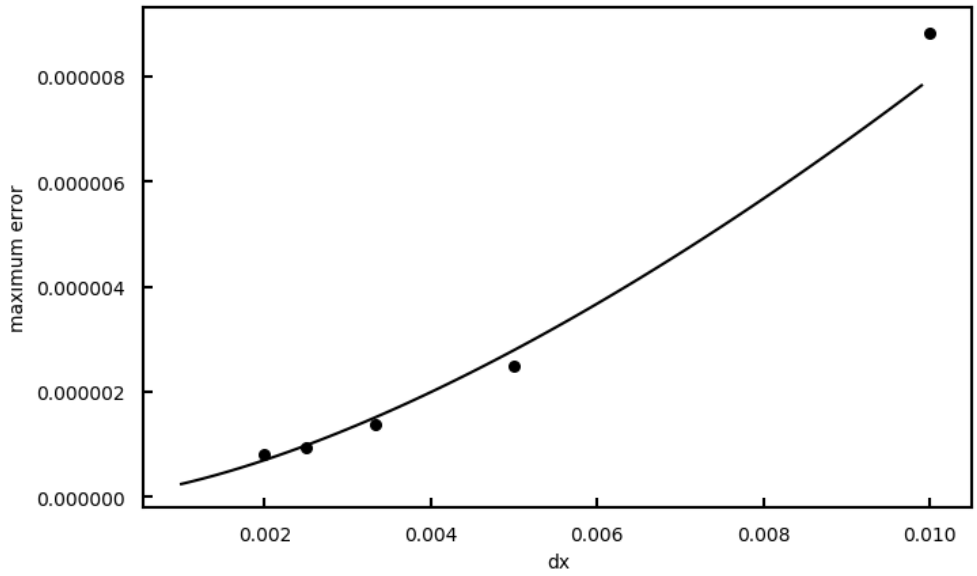


Figure 7 relation between maximum error and dx

For implicit method, we choose Δx=0.002, and time step are: 0.00001, 0.00003, 0.00005, 0.00007, 0.0001.

Table 3. error change with Δt at Δx=0.002

Δt	Maximum error
0.00001	0.00045553746635569603
0.00003	0.000562537466355692
0.00005	0.0005902759963379023
0.00007	0.0006029763589889686
0.0001	0.0006129592870992573

According to $\log(e) = \beta \log(\Delta t) + \log(C_2)$ we can approximate β = 0.1312, $\log(C_2)$= - 2.674, $\log(e) = 0.1312 \times \log(\Delta t) - 2.674$

Choose Δt=1e-6, and grid numbers as 100, 200, 300, 400, 500.

Table 4. error change with Δx at Δt=1e-6

N	Δx	Maximum error
100	0.01	3.981635766223124e-05
200	0.005	6.376415385268841e-05
300	0.00333	8.891670397859208e-05
400	0.0025	0.00011076415385269378
500	0.002	0.00012812593769832847

According to $\log(e) = \alpha \log(\Delta x) + \log(C_1)$ we can approximate α =-0.7363, $\log(C_1)$ = -5.8777, $\log(e) = -0.7363 \times \log(\Delta x) - 5.8777$

3 Parallelism

For explicit method:

When we run the same program with different cores, we example the scalabiling and isogranular scaling analysis. We test the strong scalability by using N=5000, and Δt=2e-8, using 1, 2, 4, 8 cores. And the result are: 5436(s), 7914(s), 7908(s), 12560(s), The performance as figure 8.

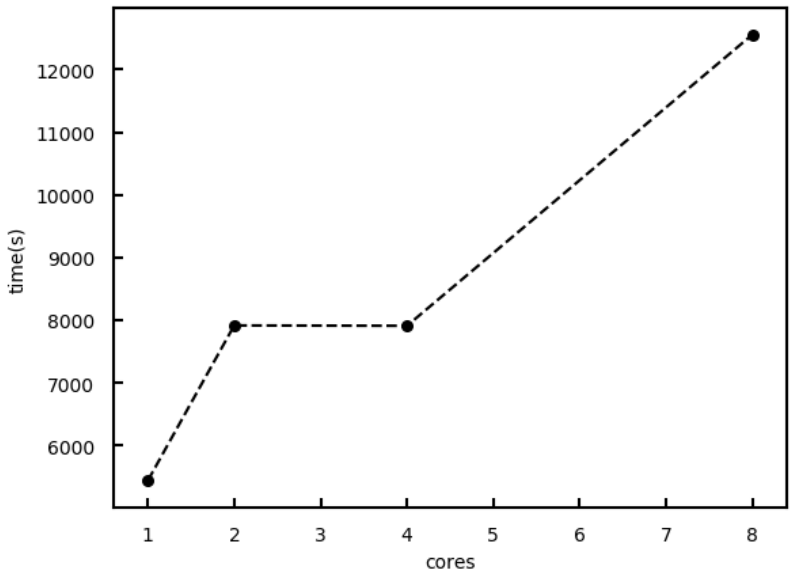


Figure 8 relation between cores and time cost for strong scalability

We set the same task in a core to test its weak scalability. We set each core to run 1000 grid data. Let Δt=1e-8 and using N=1000 run in 1core cost 3304(s), N=2000 run

in 2 cores cost 13160(s), N=3000 run in 3 cores cost 24070, N=4000 run in 4 cores cost 13040, N=5000 run in 5 cores cost 21490. Then plot the result as the figure 9.

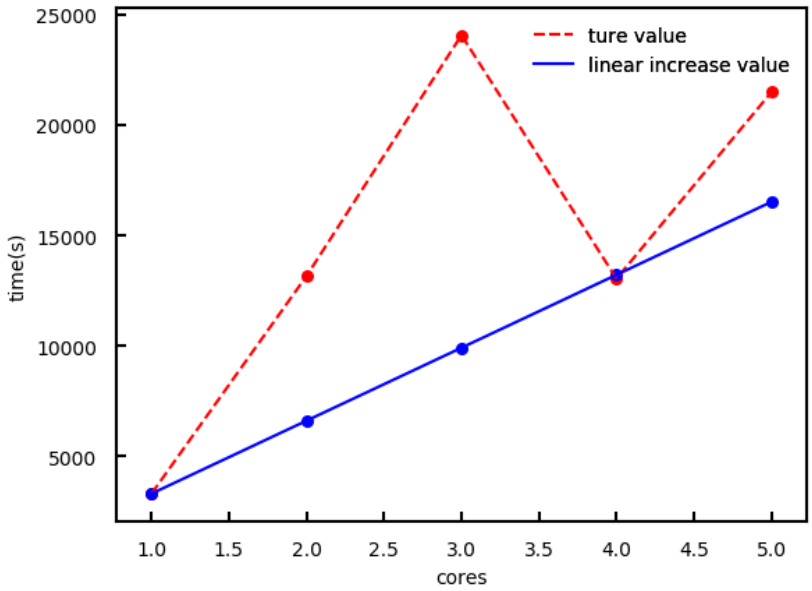


Figure 9. relation between cores and time cost for weak scalability

From figure 8 and figure 9 we can see that when the task is the same, we increase the cores, the time costing will increase, it may because the data exchange take more time than the parallelism efficient. If we increase the N the speed will increase by using parallelism. It perform better in the larger N situation.

For implicit method:

When we run the same program with different cores, we example the scalabiling and isogranular scaling analysis. We test the strong scalability by using N=5000, and Δt=1e-5, using 1, 2, 4, 8 cores. and the result are: 8250(s), 9204(s), 21340(s), 16790(s), The performance as figure 10.

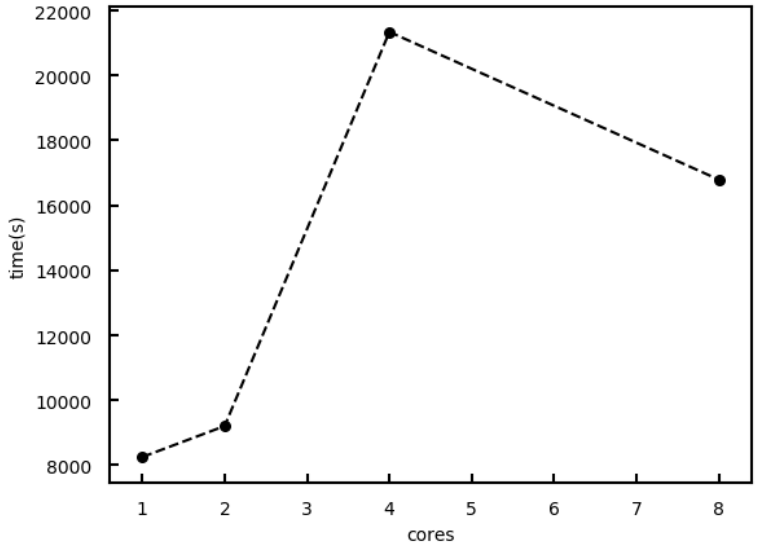


Figure 10 relation between cores and time cost for strong scalability

We set the same task in a core to test its weak scalability. We set each core to run 1000 grid data. Let Δt=1e-5 and using N=1000 run in 1 core cost 585.8(s), N=2000 run in 2 cores cost 3236(s), N=3000 run in 3 cores cost 9804, N=4000 run in 4 cores cost 17950, N=5000 run in 5 cores cost 12430.the result as the figure 11.

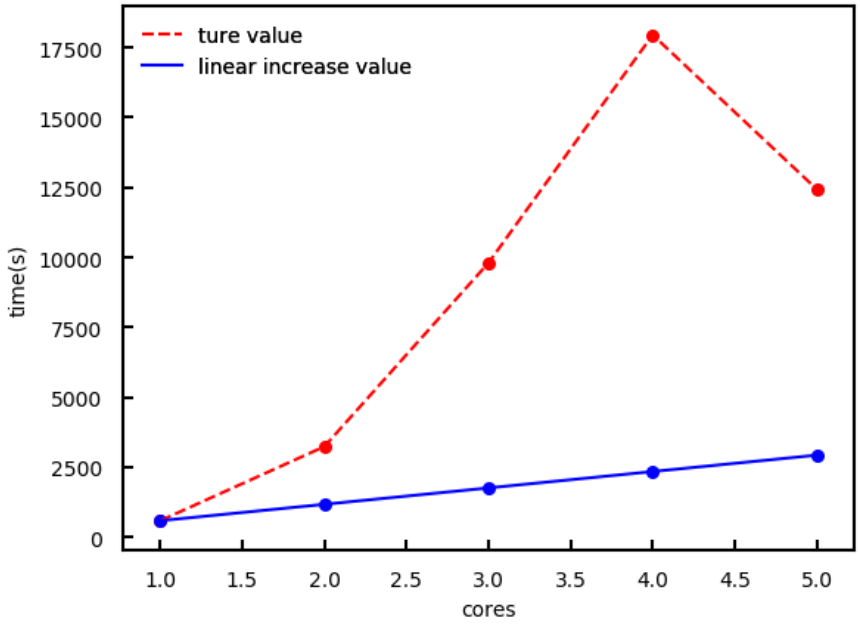


Figure 11 relation between cores and time cost for weak scalability

From the figure 10 and figure 11, we can see that when cores larger than 4, the parallelism will decrease with the cores increase. Which shows a good scalability.

Linear solver comparison

We investigate the different PC options from PETSc and the result as follows:

Using (a)Jacobi:

```
./implicit_heat.out on a  named r01n03 with 1 processor, by mae-lizj Thu Jun  9 13:57:01 2022
Using Petsc Release Version 3.16.6, Mar 30, 2022

                         Max       Max/Min     Avg       Total
Time (sec):           2.796e+02     1.000   2.796e+02
Objects:              5.000e+01     1.000   5.000e+01
Flop:                 2.560e+11     1.000   2.560e+11  2.560e+11
Flop/sec:             9.153e+08     1.000   9.153e+08  9.153e+08
Memory:               5.880e+05     1.000   5.880e+05  5.880e+05
MPI Messages:         0.000e+00     0.000   0.000e+00  0.000e+00
MPI Message Lengths:  0.000e+00     0.000   0.000e+00  0.000e+00
MPI Reductions:       0.000e+00     0.000
```

Figure 12. PC option is Jacobi

Using (b)Additive Schwarz:

```
./implicit_heat.out on a  named r01n20 with 1 processor, by mae-lizj Thu Jun  9 13:54:14 2022
Using Petsc Release Version 3.16.6, Mar 30, 2022

                         Max       Max/Min     Avg       Total
Time (sec):           2.785e+01     1.000   2.785e+01
Objects:              4.400e+01     1.000   4.400e+01
Flop:                 4.655e+09     1.000   4.655e+09  4.655e+09
Flop/sec:             1.671e+08     1.000   1.671e+08  1.671e+08
Memory:               6.307e+05     1.000   6.307e+05  6.307e+05
MPI Messages:         0.000e+00     0.000   0.000e+00  0.000e+00
MPI Message Lengths:  0.000e+00     0.000   0.000e+00  0.000e+00
MPI Reductions:       0.000e+00     0.000
```

Figure 13. PC option is Additive Schwarz

Using (c)LU with MUMPS:

```
./implicit_heat.out on a  named r01n20 with 1 processor, by mae-lizj Thu Jun  9 13:54:46 2022
Using Petsc Release Version 3.16.6, Mar 30, 2022

                         Max       Max/Min     Avg       Total
Time (sec):           1.340e+01     1.000   1.340e+01
Objects:              2.500e+01     1.000   2.500e+01
Flop:                 5.816e+09     1.000   5.816e+09  5.816e+09
Flop/sec:             4.340e+08     1.000   4.340e+08  4.340e+08
Memory:               5.527e+05     1.000   5.527e+05  5.527e+05
MPI Messages:         0.000e+00     0.000   0.000e+00  0.000e+00
MPI Message Lengths:  0.000e+00     0.000   0.000e+00  0.000e+00
MPI Reductions:       0.000e+00     0.000
```

Figure 14. PC option is LU with MUMPS

From the figure 12-14, we can see that LU with MUMPS is faster than Additive Schwarz, and both of them are faster than Jacobi.