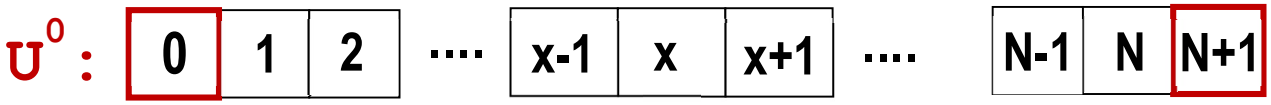
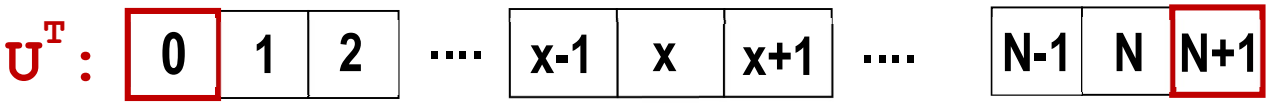
HEAT Transfer in One Dimension

We want to simulate the **heat transfer** along a one-dimensional metal bar. The model evolves following a partial differential equation that can be numerically computed using the **finite differences method**. We next show the program's input values: initial temperature in each of the **N**+2 points in the bar ($\mathbf{U}^0$[i]]), heat transfer coefficient for the metal (**HEAT_COEFF**) and number of simulation steps (**T**). The two points in the extremes of the bar represent **fixed** boundary conditions, and their value is constant along all the simulation. The output of the program is the state of the metal bar at time **T**, which is a vector with **N**+2 temperature values represented as $\mathbf{U}^T$

Input: *vector of* ***initial temperatures*** *in* ***N+2*** points with fixed **boundary conditions**,
HEAT_COEFF: heat transfer coefficient, **T**: number of simulation steps



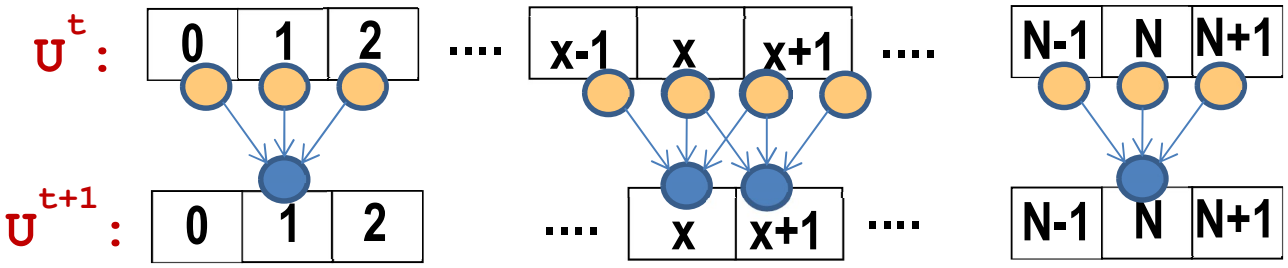
Output: *vector of* ***final temperatures*** *in* ***N+2*** points



Mathematical Description: method of finite differences

$$U^{t+1}_x = U^t_x + HEAT\_COEFF * (U^t_{x-1} + U^t_{x+1} - 2U^t_x) \quad 1<=x<=N$$

The next figure shows the **data dependences** between successive computations represented using arrows: the computation of a new element is a **stencil** that uses the value $\mathbf{U}^t$[x] and its two neighbors ($\mathbf{U}^t$[x-1] and $\mathbf{U}^t$[x+1]) at time **t**, to generate a new value for $\mathbf{U}^{t+1}$[x] at time **t+1**.



We want to develop a program that simulates the heat transfer process along a one-dimensional metal bar. To verify its correct operation, we will set the initial temperature data in a straightforward manner. Arbitrarily, we decided that at time **t**=0, all points along the bar have a temperature value of 0, except for four singular points on the bar with different temperatures: point x=0 → 1; point x=N/3 → 8; point x=4N/7 → 3; and point x=N+1 → 9. While simulating the evolution of temperature across the bar, the end points, x=0 and x=N+1, will always maintain their original values as fixed boundary conditions. In contrast, the temperatures of the remaining points change over time until the system reaches a steady state (no changes), or the simulation concludes.

The program will include a final check after the execution to verify that the result is invariant to the transformations that we will perform to the baseline implementation. This check distinguishes two cases: (1) if **N** is smaller than 100, then the temperature of all the elements of the state at time **T** are printed on the screen; but (2) if **N** is bigger, a *checksum* is computed by adding all the values at **t** = **T**+1 and printed on the screen. The first option allows comparing the detailed output for two different code implementations and tracking any possible difference. The second option is a general method to check deviations on the output results for large vectors.

Base Program: 1D Heat Transfer

The next Python-like pseudo-code presents the core fragment of the program that simulates the heat transfer process along a 1D metal bar. We use simple precision floating-point values to represent the values of the simulated system. The code inside the **for** loop, emphasized with a yellow background, is executed **T×N** times. The expression shown in yellow is a **3÷1 stencil**, and this stencil is the **basic operation** performed by the program. Therefore, the execution of the program performs **T×N stencils** (or basic operations). In other words, the computational complexity of the program is O(**N×T**).

HeatBase:

```
float HEAT_COEFF= 0.001234
float U[N+2][T+1]
for t in range(T)  # loop on time
  U[0][t+1]= U[0][t]  # copy boundary value
  for x in range(1, N+1)  # loop on elements of 1D bar
    U[x][t+1] = U[x][t] + HEAT_COEFF * ( U[x+1][t] + U[x-1][t] - 2.0 * U[x][t] )
  U[N+1][t+1]= U[N+1][t]  # copy boundary value
```

There are many opportunities for improving the performance of the previous program design. One key factor is the selection of the programming language used to implement the code: two natural choices are Python and C. A second key factor is the data layout: the amount of data read from memory and written to memory, and the order of memory accesses can determine that the performance bottleneck will be associated with the memory system. Finally, the amount of computation is also determined by how the code of the inner loop is specified. We ask you to devote 30 seconds to think on the best choices for these three questions. Then you can continue reading this document.

Data Layout

The arrangement of the **U** matrix in the Baseline version performs very badly. We propose a new version that transposes the **U** matrix so that the **N** dimension is the index on the right, and the **T** dimension is the index on the left.

HeatBaseT:

```
for t in range(T)  # loop on time
  U[t+1][0]= U[t][0]  # copy boundary value
  for x in range(1, N+1)  # loop on elements of 1D bar
    U[t+1][x] = U[t][x] + HEAT_COEFF * ( U[t][x+1] + U[t][x-1] - 2.0 * U[t][x] )
  U[t+1][N+1]= U[t][N+1]  # copy boundary value
```

Finally, since the program only needs to provide the state of the bar at the end of the simulation (at time **T**), we can codify the program using two one-dimensional arrays, instead of a single two-dimensional array. The next code implements the program with two arrays, **Uin**[] and **Uout**[], each of size **N**+2. The second array, **Uout**, is an auxiliary array that, after executing the first inner loop contains the last state of the metal bar (the temperatures at each point in the bar), including a copy of the temperatures on the boundaries. The contents of the auxiliary array are then copied to the first array.

Heat:

```
float Uin[N+2], Uout[N+2]
for t in range(T)  # loop on time
  Uout[0]= Uin[0]  # copy boundary value
  for x in range(1, N+1)  # loop on elements of 1D bar
    Uout[x] = Uin[x] + HEAT_COEFF * ( Uin[x+1] + Uin[x-1] - 2.0 * Uin[x] )
  Uout[N+1]= Uin[N+1]  # copy boundary value
  for x in range(0, N+2)  # loop on elements of 1D bar
    Uin[x]= Uout[x]  # copy values
```

Program Optimizations

Double Buffer

The program uses the strategy called "***double buffer***": in one iteration over simulation time, one array is used as the input and another array is used as the output, and on the next time iteration the role of the arrays is reversed. The output array after each iteration contains the last state of the metal bar (the temperatures at each point in the bar), and this state is used to generate the new state on the next time iteration, which is written in the other array.

HeatV2:

```
float Uin[N+2], Uout[N+2]
for t in range(T)  # loop on time
  Uout[0]= Uin[0]  # copy boundary value
  for x in range(1, N+1)  # loop on elements of 1D bar
    Uout[x] = Uin[x] + HEAT_COEFF * ( Uin[x+1] + Uin[x-1] - 2.0 * Uin[x] )
  Uout[N+1]= Uin[N+1]  # copy boundary value
  Uin, Uout = Uout, Uin  # Exchange name of arrays (do not copy data)
```

Avoid mixing data types and expression simplification

A performance problem happens in the previous versions of the code since 2.0 is considered a **double precision** constant, and, by definition, the rules of the C language indicate that operations must be done always using the precision of the data type involved with the highest precision (double > float > int). Remember that Cython code is first translated to C, then to assembler, and then to machine code. In this particular program fragment, the constant 2.0 has exactly the same value when expressed as single-precision float or double-precision float. However, the compiler is not “smart” enough and generates extra machine instructions for converting at run-time the data read from the arrays from float to double precision before performing the multiplication by 2.0, and then converting from double back to float precision before storing the result in the output array. A way to fix that problem is to create a float variable named **DOS** and assign the value 2.0 to that variable.

HeatV3:

```
float Uin[N+2], Uout[N+2]
float DOS = 2.0
for t in range(T)  # loop on time
  Uout[0]= Uin[0]  # copy boundary value
  for x in range(1, N+1)  # loop on elements of 1D bar
    Uout[x] = Uin[x] + HEAT_COEFF * ( Uin[x+1] + Uin[x-1] - DOS * Uin[x] )
  Uout[N+1]= Uin[N+1]  # copy boundary value
  Uin, Uout = Uout, Uin  # Exchange name of arrays
```

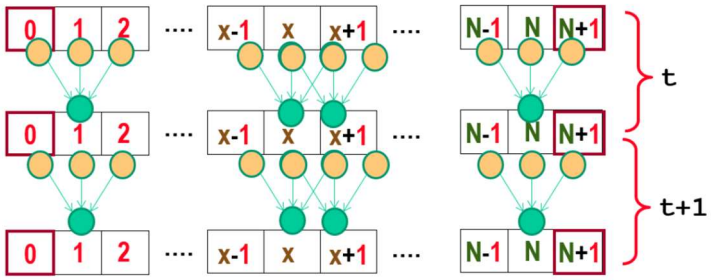
The expression implementing the 3÷1 stencil can be simplified by pre-computing the expression stored in the variable **C** shown in the next code, which is constant during the execution of the nested loops.

HeatV4:

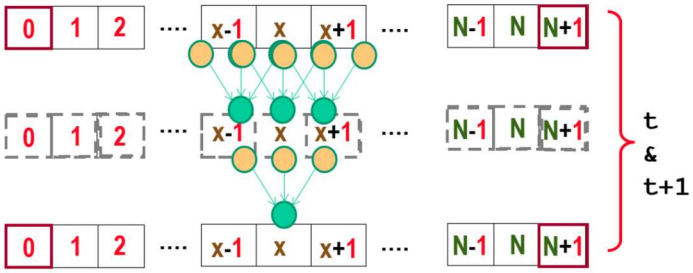
```
float Uin[N+2], Uout[N+2]
float C = 1.0 - 2.0*HEAT_COEFF
for t in range(T)  # loop on time
  Uout[0]= Uin[0]  # copy boundary value
  for x in range(1, N+1)  # loop on elements of 1D bar
    Uout[x] = C *Uin[x] + HEAT_COEFF * ( Uin[x+1] + Uin[x-1])
  Uout[N+1]= Uin[N+1]  # copy boundary value
  Uin, Uout = Uout, Uin  # Exchange name of arrays
```

V5: Time Blocking

The next figure illustrates the execution of two consecutive time iterations (**t** and **t**+1). On the iteration corresponding to time step **t**, the baseline algorithm simply reads the input elements in the first row (three at a time) and uses those elements to generate and write the elements on the second row.



The proposed optimization is called **time blocking** or **time unrolling**: two (or more) iterations of the time loop (**t** and **t**+1) are done simultaneously while the input vector is read only once. The intermediate results generated for the time iterations are reused, and need not be written to the slow memory. The next figure illustrates the idea.



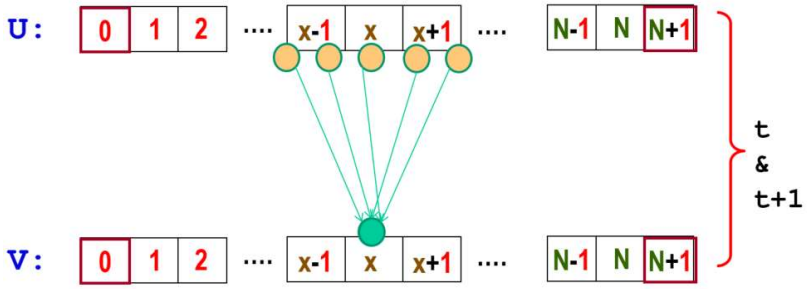
One can generate the output corresponding to time **t**+1 (third row) using the input for time step **t** (first row), without having to store the output for time step **t** (second row). Every new stencil uses 5 input values (corresponding to positions **x**-2, **x**-1, **x**, **x**+1 and **x**+2 at time **t**) to generate three intermediate results (corresponding to positions **x**-1, **x** and **x**+1 at time **t**+1). Those intermediate results are not written to a vector, but are immediately used to compute the value in position **x** at time **t**+2, which is written to the output vector (as shown in the Figure). The reward of this strategy is that only half the rows have to be read and written from and to memory. The price to pay is a higher amount of computation operations.

The next code shows the inner loop for a temporal blocking naïve implementation that groups two steps of the time-loop into a single iteration. For simplicity, the code assumes that **T** is an even number. The loop needs to check two cases, one corresponding to the position ***x***=1 (since the value at position **x**=0 is fixed) and one corresponding to the position ***x***=N (since the value at position **x**=**N**+1 is fixed).

HeatV5:

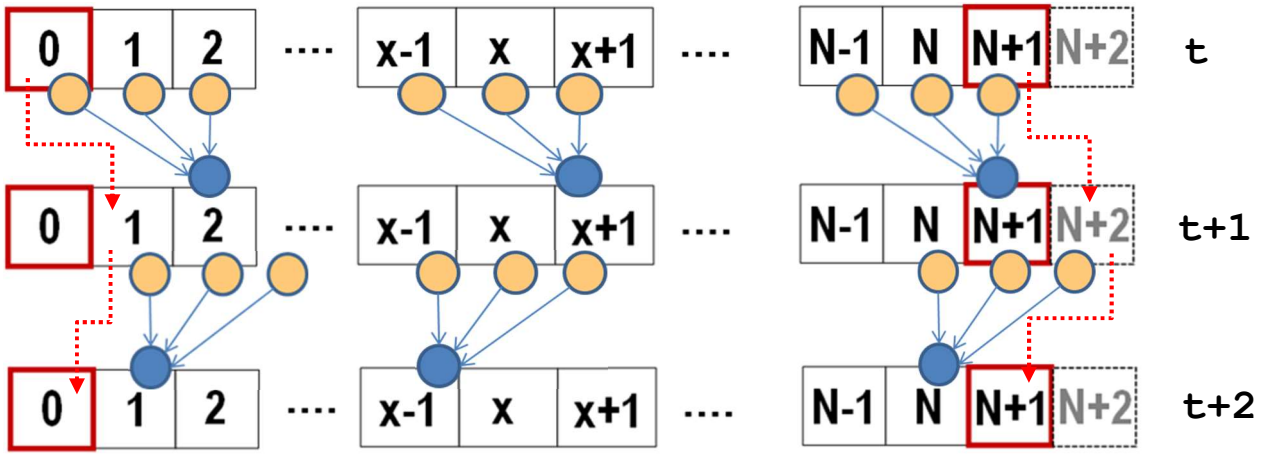
```
float Uin[N+2], Uout[N+2]
float C = 1.0 - 2.0*HEAT_COEFF
for t in range(0,T,2)  # loop on time, two time steps per iteration
  Uout[0]= Uin[0]  # copy boundary value
  for x in range(1, N+1)  # loop on elements of 1D bar
    if (x==1):    # special case for x==1
      Left = Uin[0]
    else:
      Left = C*Uin[x-1] + HEAT_COEFF*(Uin[x]+Uin[x-2])
    Middle = C*Uin[x] + HEAT_COEFF*(Uin[x+1]+Uin[x-1])
    if (x==N):    # special case for x==N
      Right = Uin[N+1]
    else:
      Right = C*Uin[x+1] + HEAT_COEFF*(Uin[x+2]+Uin[x])
    Uout[x] = C*Middle + HEAT_COEFF*(Right+Left)
  Uout[N+1]= Uin[N+1]  # copy boundary value
  Uin, Uout = Uout, Uin  # Exchange name of arrays
```

It is still possible to **improve the performance** of the code by eliminating the internal **if** statements and reducing the computation operations by mathematical transformations. The next figure illustrates the idea of computing the stencil for time iterations **t**+2 directly from **t**, without needing to compute the intermediate values for time step **t**+1. The goal is to apply mathematical transformations (distributive & associative properties) to reduce the total number of computation operations.



V6: Array Reuse

The following figure shows a strategy that avoids having to duplicate the storage space of vector **U**, which contains the temperature data. Apart from the advantage of requiring approximately half the memory storage, which can be critical if a single vector occupies more than half the available main memory, the proposal below improves memory performance due to the underlying policy for handling cache misses. In short, it is now not necessary to transfer from memory into the cache the blocks corresponding to a second large vector, whose contents are really not used by the program, since those memory blocks will be completely updated with new data by the program.

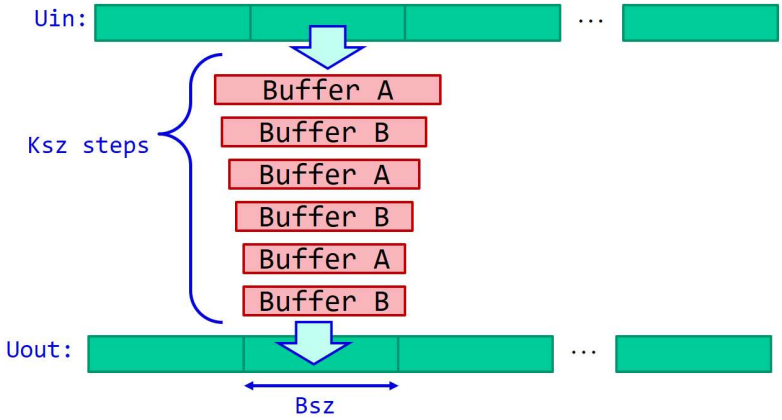


The strategy involves repeating two stages, where each stage shifts the data by one position—either to the right or to the left. To implement this, an additional data element (element N+2) is required at the end of the vector. In the first stage, a new vector state is generated by shifting the data one position to the right. To prevent data dependencies that could degrade performance and produce incorrect results, the data must be processed starting from position N+1 and proceeding downwards. In the second stage, the results are generated by shifting the data results one position to the left. This process starts from position 1 and progresses forwards. The values at the frontiers, positions 0 and N+1, must also be shifted right and left as required. Do not forget to allocate an extra element on the array. Check that the code generated has been vectorized (SIMD) by the compiler.

V7: Data Blocking + Time Blocking

A complementary idea to the previous one, which also allows improving the performance of the program for large problem sizes, consists of dividing the temperature vector into blocks of size **Bsz**, and calculating **Ksz** time steps for each block, before calculating the following blocks. The figure below shows the idea, applied to the second block of the vector. The vectors, **Uin** and **Uout**, are still used, but two small vectors are also necessary (which in the figure are identified as **Buffer A** and **Buffer B**), which also work in the form of a double buffer (or circularly) to obtain the results corresponding to times **t**, **t**+1, **t**+2, **t**+3, … up to **t** + **Ksz**. It is important to note two things: (1) in order to calculate multiple time steps the data on vector **Uin** that is on the frontier with neighboring blocks must be read; and (2) the first and last blocks of the temperature

vector must be treated as particular cases, to consider the boundary conditions of the problem (that the edges of the vector always have a constant value).



The figure above shows that generating **Bsz** results requires computing more than **Bsz** intermediate results. As time progresses (**t**+1, **t**+2...) fewer temporal results need to be generated.

It is advisable to reduce the complexity of the program as much as possible, to achieve a correct implementation by making as few changes as possible to the code. Once the first milestone is achieved, complexity can then be added to the implementation incrementally, always verifying that this complexity translates into improved performance. For example, it is recommended to design a first data blocking implementation that do not use ***time blocking***.

One way to simplify the algorithm is to initially copy the data block from vector **Uin** to **buffer A**, without performing stencil operations; then the stencil operations are performed using **buffers A** and **B** alternately; and finally, the results are copied to the vector **Uout**. The size of **buffers A** and **B** must be sufficient to accommodate the amount of data generated in the first step. Specifically, the buffer size should be **Bsz** + 2×**Ksz**.