LA-UR-23-29527

Approved for public release; distribution is unlimited.

Title:	Remapping of Data Between One-Dimensional Meshes
Author(s):	Ray, Navamita Hudgins, Jahi Michael Shevitz, Daniel Wolf
Intended for:	Documenting work done during internship.
Issued:	2023-08-17

Remapping of Data Between One-Dimensional Meshes

Jahi Hudgins, Daniel Shevitz, Navamita Ray

May-August 2023

Abstract

In this report we present two approaches to data remapping between one-dimensional meshes implemented with the c++ programming language. Our goal was to test the performance of two search algorithms, linear and binary, and verify the accuracy of our implementations of the two methods. We first introduce the concept of data remap and meshing components, as well as their various uses. We then delve into the differences between point-wise and conservative remap, the algorithms used in the implementations, and lastly confirm the implementations work as intended when given various inputs. We expect that, after profiling, the binary search algorithm will be more efficient than the linear algorithm for sorted sets of data, the point-wise remap implementation to accurately approximate the data transfer between two meshes, and the conservative remap implementation to conserve the area underneath the curve of two distinct meshes.

1 Introduction

Remapping is the process of transferring known *field data* between two distinct *meshes*. It is an important concept because it allows us to see how data will be affected when being placed into a new environment. The process of data remapping is used in many ways ranging from simple image resizing to complex simulations of fluid mechanics. The main issues concerning remap are accuracy and performance. In this work, we studied two approaches to performing data remapping between one-dimensional meshes. The two methods, known as Point-wise Remapping and Conservative Remapping, and their respective algorithms will be described in detail in the following sections.

In Section 2, we describe the concepts of meshes and field data that are required by the remapping algorithms. Next, in Section 3, we present the point-wise data remapping algorithm. We provide numerical results as well as profiling results for parts of the algorithm. In Section 4, we present the conservative data remapping algorithm, and provide numerical results. We finally present our conclusions in Section 5.

2 Meshes and Fields

A mesh is a set of interconnecting, non-overlapping cells approximating the geometry of a domain. Meshes are constructed using two key elements, *cells* and *nodes*, but can also contain *field values*. While meshes can range from one to multi-dimensional domains, in this work we will use one-dimensional meshes with equidistant nodes over the interval [0, 1]. Here are the key components of our mesh and field defined on it:

- Nodes: Coordinate points existing on the x-axis
- Cells: An edge formed by connecting two or more adjacent nodes
- Field: A value defined at either individual nodes or at cell midpoints illustrated by elevation on the y-axis

Figure 1 illustrates a one-dimensional mesh with five cells. Figures 2a and 2b illustrate the two locations where a field might be defined on the source mesh.

3 Point-wise Remapper

The first method tested is the simplest form of remap: point-wise remapping. In point-wise remapping, the field values are defined at each individual node and *piece-wise linear interpolation* is used to approximate the target field from the line functions of the source cells, resulting in a node-to-node remapping of data.

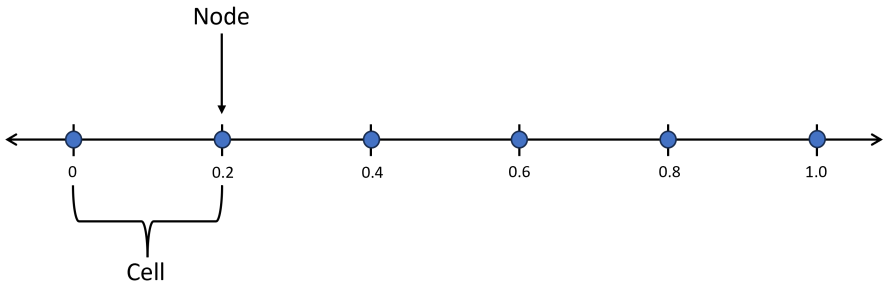
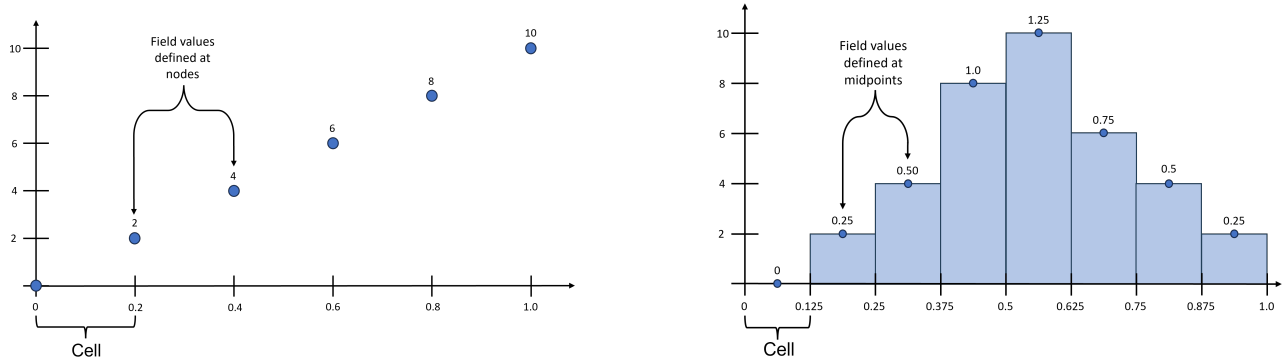


Figure 1: Example of a 1D mesh



(a) 1D mesh with field values at nodes

(b) 1D mesh with field values at cell midpoints

Figure 2: Field Definition Locations

3.1 Piece-wise Linear Approximation over Source Mesh

Piece-wise linear approximation is the process of determining the function describing the source mesh by approximating the line equation connecting the two nodes over each cell. The result of the process on a random plot is illustrated in Figure 3.

The slope-intercept equation of a line is:

$$y = mx + b \tag{1}$$

Given two points (x_0, y_0) and (x_1, y_1), we compute the slope m between them using the formula:

$$m = \frac{y_1 - y_0}{x_1 - x_0} \tag{2}$$

We then must find b, the y-intercept of the line between the two points, with the following equation:

$$b = (-m * x_0) + y_0 \tag{3}$$

This step is necessary because these line functions are what we'll use to interpolate the target field values.

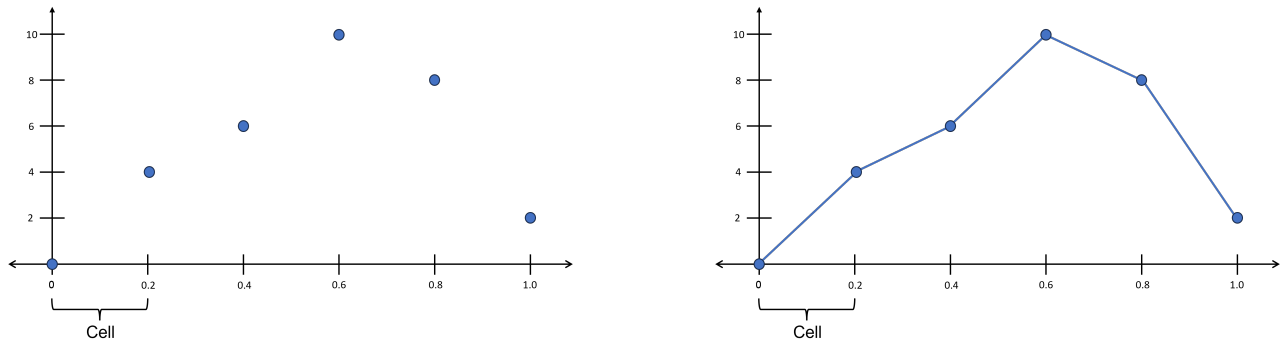
3.2 Search

The search step determines which source cell a target node lies in. This is crucial to the success of the remapping algorithm: The correct values must be returned in order for the right line functions to be passed into the interpolation function. While standard search algorithms simply check whether a value exists in an array, here the search is between two meshes. The searches have two different success cases:

i If element $a[i]$ was equal to the *key*

ii If the *key* was in between element $a[i]$ and element $a[i+1]$

This change was needed in the event that a target node existed between, but not on, two source nodes. In our code we tested two different search algorithms: *linear* and *binary*.



(a) Points before Linear Approximation

(b) Points after Linear Approximation

Figure 3: Piece-wise Linear Approximation

3.2.1 Linear Search

A sequential search algorithm that starts with the first element in a set $a[0]$ and compares each element $a[i]$ to the *key*, stopping either when the *key* is found or when the search reaches the end of the set. Figure 4 illustrates the search process for a specific target node.

Although this algorithm has the advantage of being able to search an unsorted set, the nature of the linear search results in a time complexity of *O(N)* in every case aside from when the *key* matches the first element in the set. The time complexity is not a huge deal with smaller sets, this algorithm loses more and more efficiency in proportion to the size of the set. This method is demonstrated in Algorithm 1.

Algorithm 1 Linear Search

Require: $nCell \geq 0$
  for $i \leftarrow 0$ **to** $nCell$ **do**
    if $target \geq source[i]$ and $target \leq source[i+1]$ **then**
      $return \leftarrow i$
    else
      $i \leftarrow i + 1$
    end if
  end for

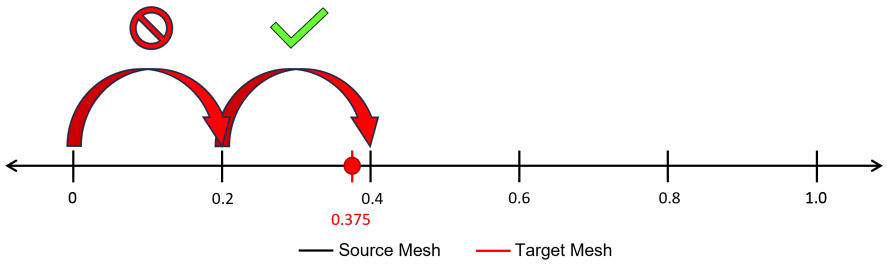
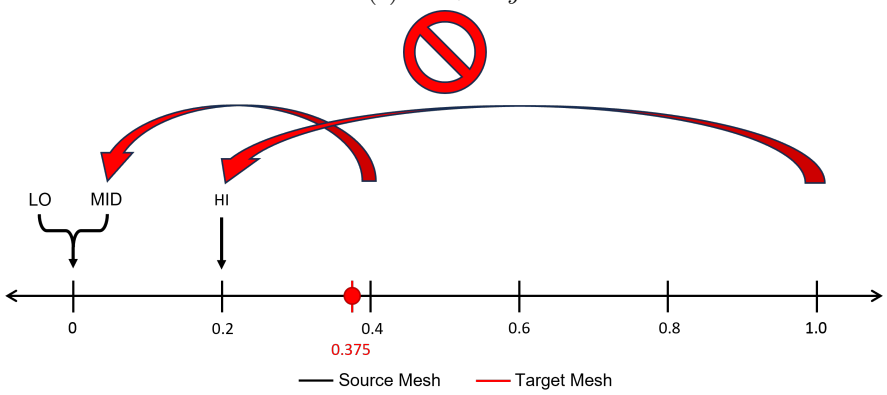


Figure 4: Point-wise Linear Search

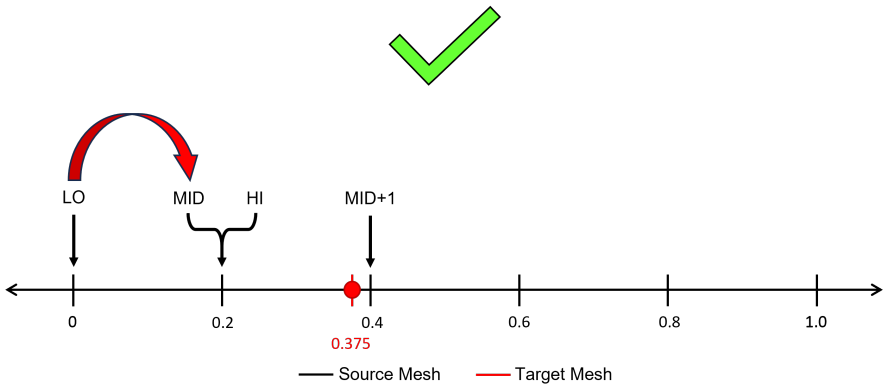
3.2.2 Binary Search

A search algorithm that is used on *sorted sets* that works by repeatedly splitting the search interval in half and comparing the middle of the list $a[m]$ with the *key* for each iteration until either the *key* is found or until the entire set has been searched. Figure 5 illustrates the process for a specific target node. While this search algorithm has the drawback of only working on sorted sets, because the search interval is constantly being halved, the binary search is very efficient with an average time complexity of *O(log N)*. This method is demonstrated in Algorithm 2.

(a) $mid > key$



(b) $mid < key$

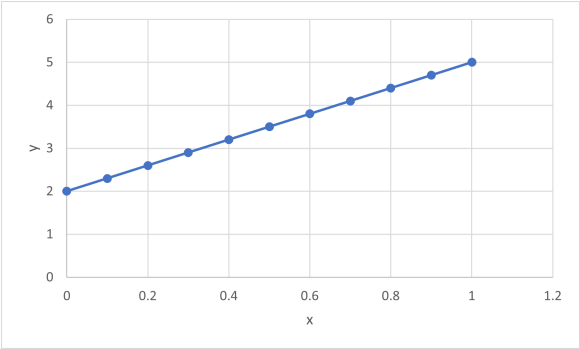


(c) $mid \geq key \geq mid + 1$

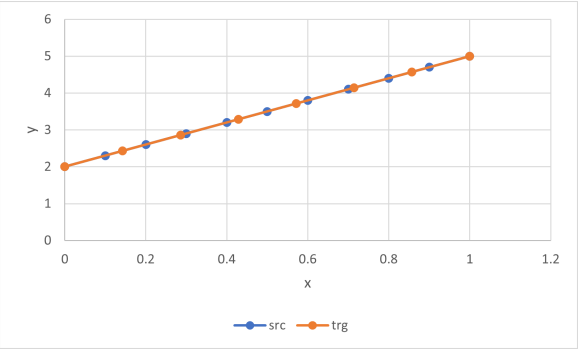
Figure 5: Point-wise Binary Search [$key = 0.375$]

Algorithm 2 Binary Search

Require: $nCell \geq 0$
  $lo \leftarrow 0$
  $hi \leftarrow nCell$
  while $lo \leq hi$ **do**
    $mid \leftarrow lo + ((hi - lo)/2)$
    if $target \geq source[mid]$ and $target \leq source[mid + 1]$ **then**
      $return \leftarrow mid$
    else if $source[mid] < target$ **then**
      $lo \leftarrow mid + 1$
    else if $source[mid] > target$ **then**
      $hi \leftarrow mid - 1$
    else
      $return \leftarrow -1$
    end if
  end while



(a) Source Mesh



(b) Meshes Overlapped

Figure 6: Linear Function Verification

3.3 Interpolation on Target Mesh

Linear interpolation is the process of looking at a known *source mesh* and its *field*, then looking at a *target mesh*, figuring out where each target cell lies on the source mesh, then calculating the *target field* using the slope and y-intercepts of the respective source mesh cell.

$$y_{trg} = (m_{src} * x_{trg}) + b_{src} \tag{4}$$

3.4 Verification

To verify our point-wise remap implementation, we tested the meshes with the parameters shown in Table 1 against both a linear and a quadratic equation.

	Source Mesh	Target Mesh
Number of cells	10	7
Number of nodes	11	8

Table 1: Point-wise Verification Mesh Sizes

The verification process involved manually calculating the solution at each step, such as what cells the search functions should return and what the line equations of each cell should be, then placing asserts in the algorithm to confirm that the experimental results matched what was expected.

3.4.1 Linear Function

The linear function of the line over the domain of our reference is:

$$y = 3x + 2 \tag{5}$$

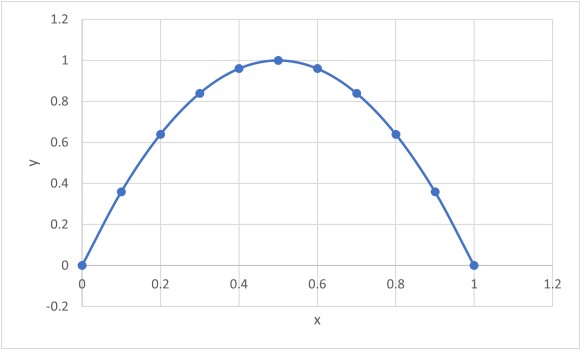
and the mesh generated by this function can be seen in Figure 6a. Figure 6b illustrates the success of the linear verification: the line connecting the target points lies in line with the source points.

3.4.2 Quadratic Function

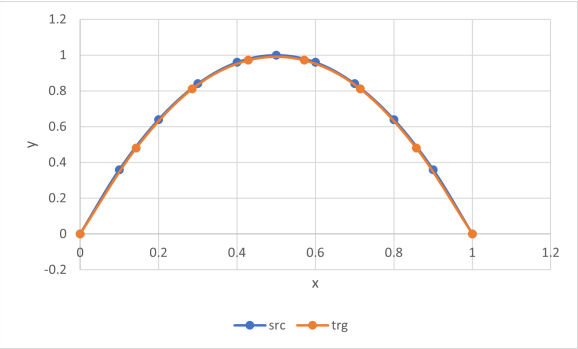
We next used a quadratic function over the domain to interpolate from the source mesh to the target mesh. The equation of the quadratic function used to obtain field values at the source nodes are:

$$y = -4(x - \frac{1}{2})^2 + 1 \tag{6}$$

and the mesh generated is shown in Figure 7a. Again, the success of the quadratic verification can be seen in 7b as the source points lie on the line connecting the target points.



(a) Source Mesh



(b) Meshes Overlapped

Figure 7: Quadratic Function Verification

3.5 Performance of Searches

Finally, we profiled the linear and binary searches. We fixed the target mesh size (200 cells) and performed the search with source meshes starting with 100 cells up-to 100,000 cells. We used the steady clock function from the c++ "chrono" library and placed the timing blocks around the search functions to time our searches. For each source mesh, we then got the average time for each of the searches and plotted the results as shown in Figure 8.

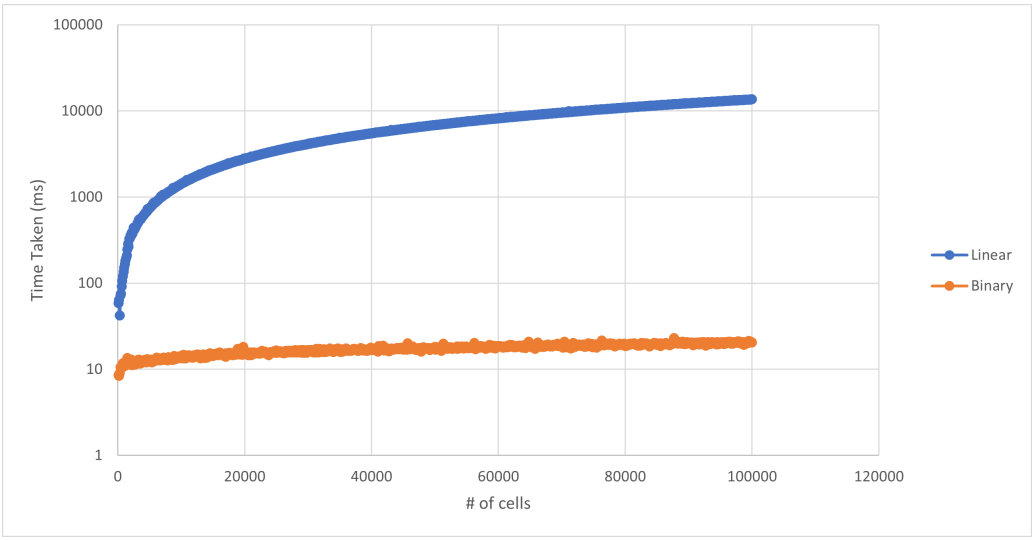


Figure 8: 10^2 to 10^5 Profiling

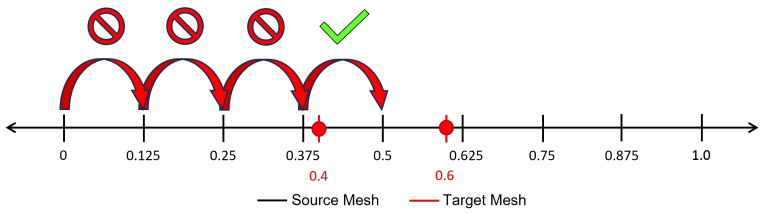
4 Conservative Remapper

The second method tested was conservative remap. This method differs from point-wise remap in quite few ways: Rather than simply defining the field values at individual nodes, the field values in conservative remap are only defined at the cell midpoints and are used to calculate the area of the cell. Because point-wise remap defines the field values at individual nodes and approximates the target field values using linear interpolation, some detail may be lost in the event that the target mesh has:

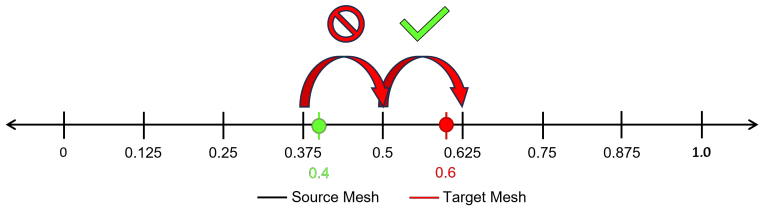
- the target mesh's sub-interval is larger than that of the source mesh
- the sub-intervals of the source and target meshes differ too much

Conservative remap, however aims to distribute the area of the source mesh to the target cells so that the total area is conserved between the meshes. In order to do this, there are three steps involved:

- Search: Find which source cells overlap the target cell
- Intersection: Obtain how much each source cell overlaps with the target cell



(a) Searching for head node



(b) Searching for tail node

Figure 9: Conservative Linear Search

- Interpolation: Accumulate the area from the overlapping source cells and then find the target field value at the cell-center.

4.1 Search

Searching in the conservative remap algorithm is slightly different from that of the point-wise remap. Because the entire target cell is being looked at, rather than a single node at a time, each search must find the *head* and *tail* nodes of the target cell. This means that for each iteration of the search function, the mesh is actually being looked through twice.

4.1.1 Linear Search

The linear search algorithm for conservative remap starts just like the point-wise linear search: first searching for the source cell that the target cell's head node lies in. The next iteration of the search starts in the source cell where target head node was found, then searches again until the target tail node is found. Figure 9 illustrates the process for a specific target node. This method is demonstrated in Algorithm 3.

4.1.2 Binary Search

A search algorithm that finds the desired value in a *sorted list* by comparing the target value with the middle of the list, then dividing the search interval in half for each iteration until either the target is found or until the entire list has been searched. While this search algorithm requires the list to be sorted, this type of search has an average time complexity of *O(log N)*. This method is demonstrated in Algorithm 4.

4.2 Intersection Approximation over Source Mesh

Once we know which source cells overlap with a target cell, we next need to determine how much each source cell overlaps with the target cell. To do this, we first check for one of the three cases depicted in Figure 12. If the target cell is contained in just one source cell, the amount of intersection is the target cell width as seen in Figure 12a. If the head and tail nodes of the target cell are located in two adjacent source cells, illustrated in Figure 12b, the two intersection amounts are:

- the positive difference between the target head node and the shared source node
- the positive difference between the target tail node and the shared source node

If the head and tail nodes of the target cell are located in two non-adjacent source cells, shown in 12c, the first and last intersections are calculated the same way as Case 2, and the intersections for any source cells between the head and tail will be the width of the source cell. The method dealing with these cases are shown in Algorithm 5.

Algorithm 3 Linear Search

Require: $nCell \geq 0$
  for $i \leftarrow 0$ **to** $nCell$ **do**
    if $source[i] \leq targetStart$ and $source[i+1] \geq targetStart$ **then**
      $head \leftarrow i$
    else
      $i \leftarrow i + 1$
    end if
  end for

  $i \leftarrow head$
  for i **to** $nCell$ **do**
    if $source[i] \leq targetEnd$ and $source[i+1] \geq targetEnd$ **then**
      $tail \leftarrow i$
    else
      $i \leftarrow i + 1$
    end if
  end for

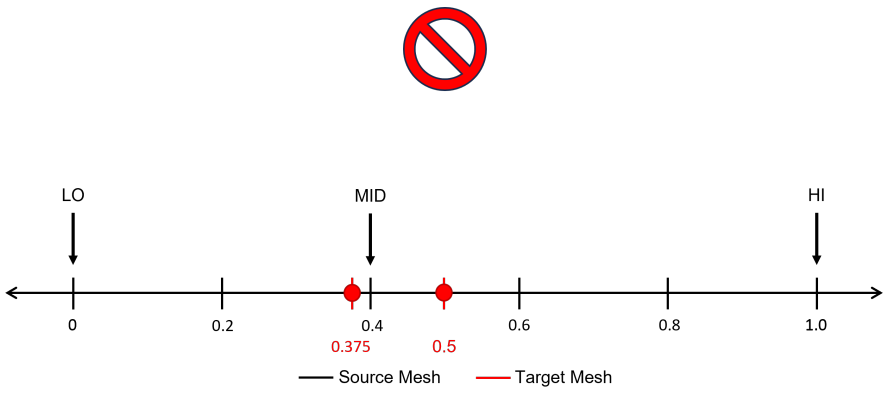
  $i \leftarrow head$
  while $i < tail$ **do**
    $return[i] \leftarrow i$
  end while

Algorithm 4 Binary Search

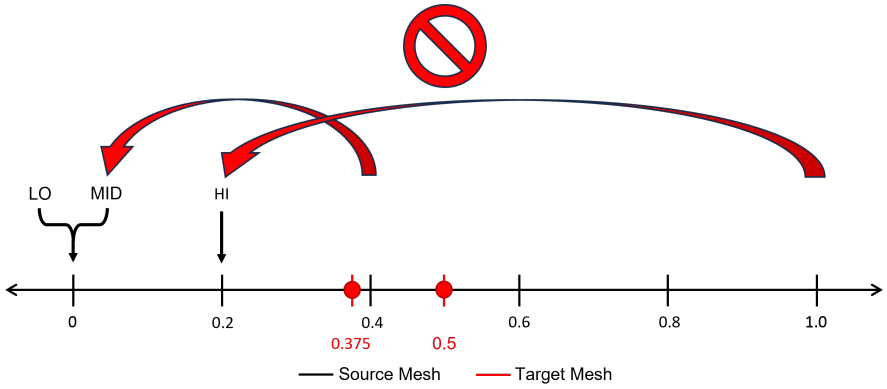
Require: $nCell \geq 0$
  $lo \leftarrow 0$
  $hi \leftarrow nCell$
  while $lo \leq hi$ **do**
    $mid \leftarrow lo + ((hi - lo)/2)$
    if $source[mid] \leq targetStart$ and $source[mid+1] \geq targetStart$ **then**
      $head \leftarrow mid$
    else if $source[mid] < target$ **then**
      $lo \leftarrow mid + 1$
    else if $source[mid] > target$ **then**
      $hi \leftarrow mid - 1$
    else
      $return \leftarrow -1$
    end if
  end while

  $lo \leftarrow head$
  $hi \leftarrow nCell$
  while $lo \leq hi$ **do**
    $mid \leftarrow lo + ((hi - lo)/2)$
    if $source[mid] \leq targetEnd$ and $source[mid+1] \geq targetEnd$ **then**
      $tail \leftarrow mid$
    else if $source[mid] < target$ **then**
      $lo \leftarrow mid + 1$
    else if $source[mid] > target$ **then**
      $hi \leftarrow mid - 1$
    else
      $return \leftarrow -1$
    end if
  end while

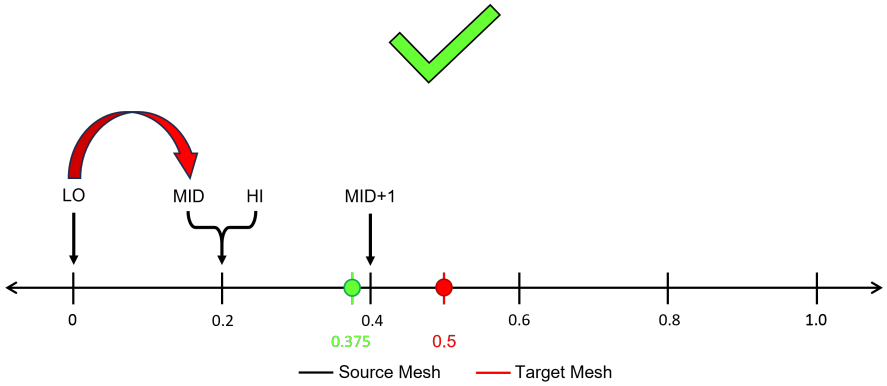
  $i \leftarrow head$
  while $i < tail$ **do**
    $return[i] \leftarrow i$
  end while



(a) $mid > key$



(b) $mid < key$

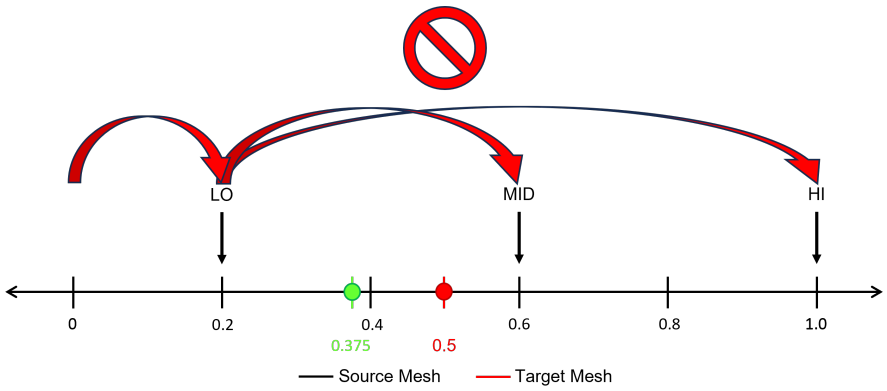


(c) $mid \geq key \geq mid + 1$

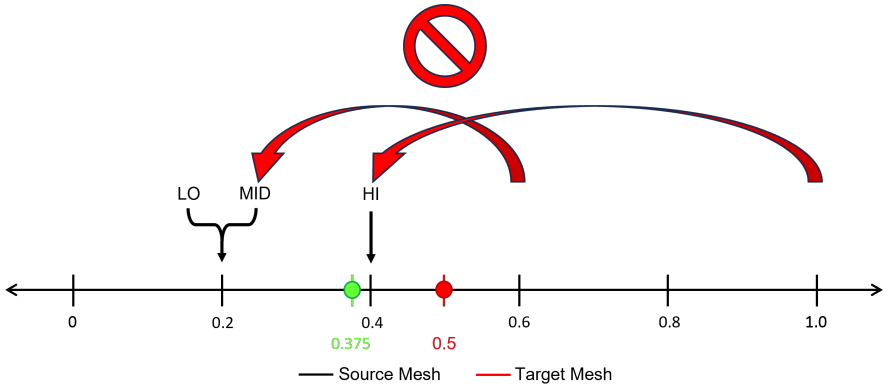
Figure 10: Conservative Binary Search Head [$key = 0.375$]

Algorithm 5 Intersection

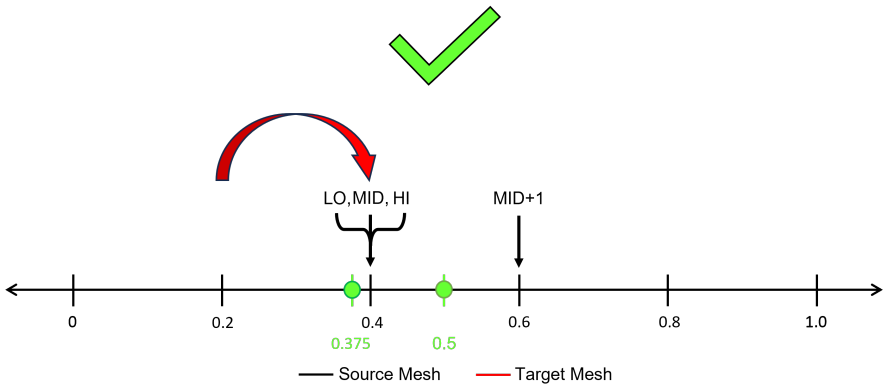
```
if found.size = 1 then
    return ← head − tail
else
    for i to found.size do
        if i = 0 then
            return[i] ← src[found[i + 1]] − head
        else if i = found.size − 1 then
            return[i] ← tail − src[found[i]]
        else
            return[i] ← src[found[i + 1]] − src[found[i]]
        end if
    end for
end if
```



(a) $mid > key$

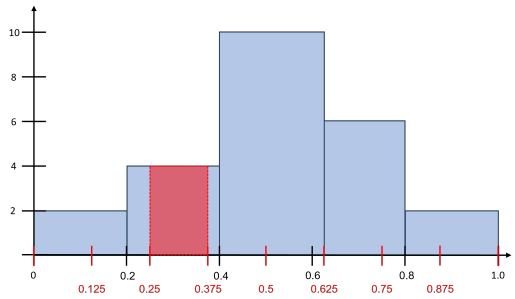


(b) $mid < key$

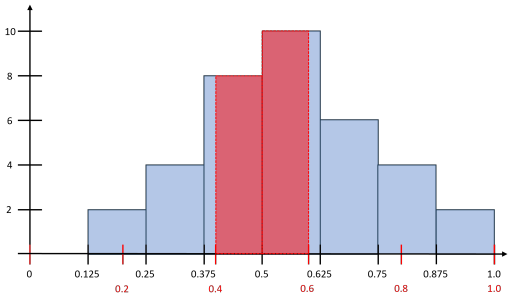


(c) $mid \geq key \geq mid + 1$

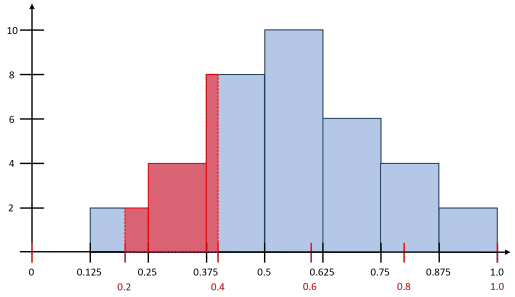
Figure 11: Conservative Binary Search Tail [$key = 0.5$]



(a) Case 1: Contained inside one source cell



(b) Case 2: Intersecting with 2 source cells



(c) Case 3: Intersecting with ≥ 3 source cells

Figure 12: Intersection Cases

4.3 Interpolation on Target Mesh

Once we have obtained the amount of intersection of each source cell overlapping a target cell, the next step is to obtain the area proportional to the intersection from each such source cell and then accumulate it over the target cell. The target cell's area A_{trg} can be calculated using Equation 7, where i_{src} is the amount of intersection with the source cell and y_{src} is the height of the source cell.

$$a_{trg} = \sum_{n=1}^{N_c} i_{src} * y_{src} \tag{7}$$

After the target areas have been found, the target field values y_{trg} can be calculated using Equation 8, where w_{trg} is the width of the target cell.

$$y_{trg} = a_{trg}/w_{trg} \tag{8}$$

4.4 Verification

We verified the implementation of our conservative remap against meshes with the parameters shown in Table 2 against a linear and a quadratic equation.

	Source Mesh	Target Mesh
Number of cells	10	7
Number of nodes	11	8

Table 2: Conservative Verification Mesh Sizes

As the conservation of area between meshes is the goal of conservative remap, we manually calculated the total area of the meshes, the search and intersection vectors, and the field values, then used asserts to confirm the experimental results matched the expected output.

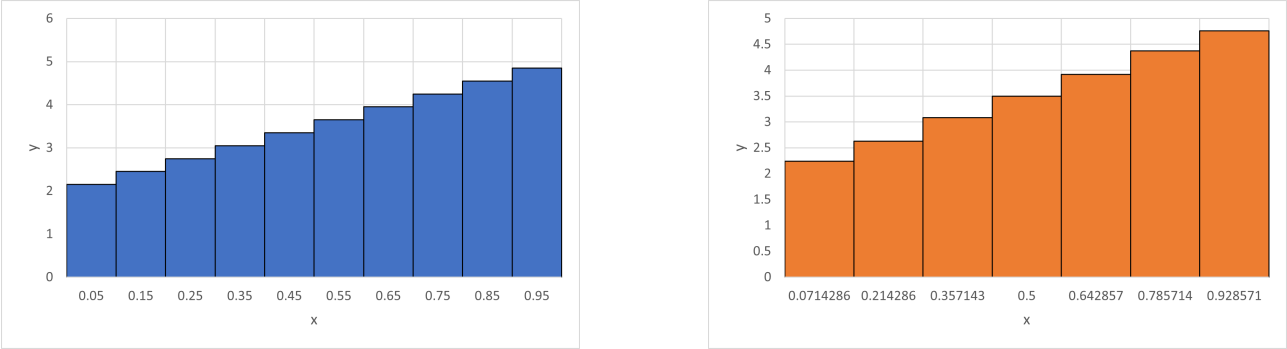
4.4.1 Linear Function

The mesh in Figure 13a was generated from the approximation of Equation 9 and the interpolated target mesh is presented in Figure 13b. Table 3 shows the conservation of area between the two meshes

$$y = 3x + 2 \tag{9}$$

	Area
Source Mesh	3.50000
Target Mesh	3.50000

Table 3: Approximated Area Under the Linear Function



(a) Source Mesh

(b) Target Mesh

Figure 13: Linear Function Verification

4.4.2 Quadratic Function

We then used Equation 10 to approximate the mesh seen in Figure 14a and the result of the interpolation is shown in Figure 14b.

$$y = -4(x - \frac{1}{2})^2 + 1 \tag{10}$$

As displayed in Table 4, the area between the meshes has been conserved using our conservative remap implementation.

4.5 Performance of Searches

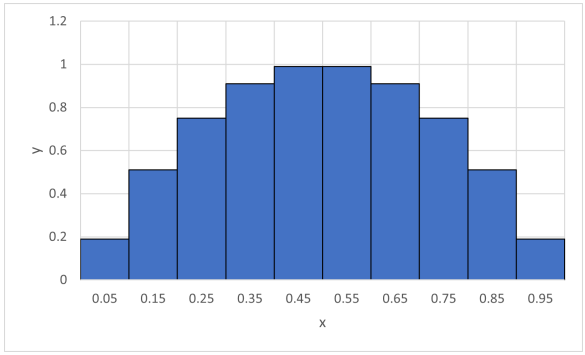
We also profiled the linear and binary searches for the conservative remapper. As before, we observe that the binary search is significantly faster than the linear search as we increase the source mesh size.

5 Conclusion

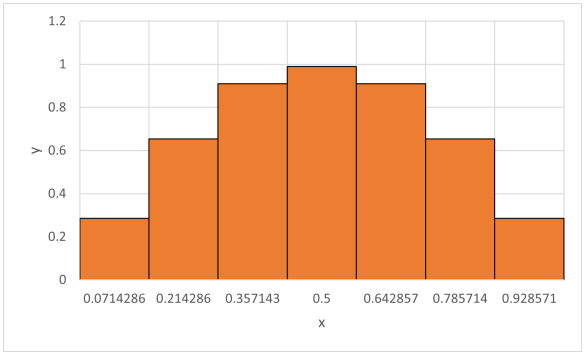
Our goal with this project was to learn about the concept of data remap and successfully implement two remapping methods between one-dimensional meshes. In our implementations, we also aimed to confirm the higher efficiency of the binary searching algorithm against the linear search algorithm. In our experiments, we were able to certify the correct operation of our implementations: the point-wise remapper approximated the source field from given functions and accurately interpolated target mesh field values from the source mesh; the conservative remapper approximated the area under the curve of a given function onto a source mesh and successfully interpolated a target mesh that conserved the area from the source. Our studies of the searching algorithms also proved a binary search is faster than a linear search for sorted sets of data.

	Area
Source Mesh	0.67000
Target Mesh	0.67000

Table 4: Approximated Area Under the Quadratic Function



(a) Source Mesh



(b) Target Mesh

Figure 14: Quadratic Function Verification

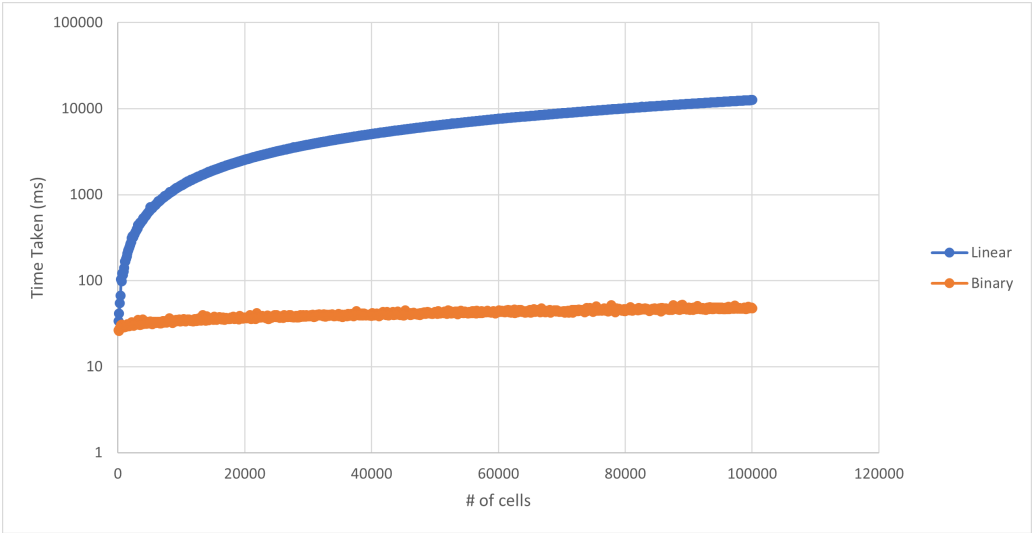


Figure 15: 10^2 to 10^5 Profiling