heights.h Documentation

Haochen (Langford) Huang

June 9, 2025

Version: 2.0 Updated: 2025-06-08

Section 1

Introduction

'heights.h', together with 'parabola.h', serves as a toolbox for 'curvature.h' to compute surface curvature in multiphase flow. The aim of the file is to allocate a value representing the distance to the surface for every cell. This value is named 'height', hence the name of the header file. Before diving into details, several conceptual definitions are introduced with a 1D example.

1.1 Surface

For each cell, if a 'coherent surface' occurs within a certain distance, the cell is assigned a valid height value. Otherwise, an invalid data value 'nodata' is allocated. A 'coherent surface' is defined as a process where the color function (volume fraction in the VOF method) c changes from 0 to 1 or vice versa.

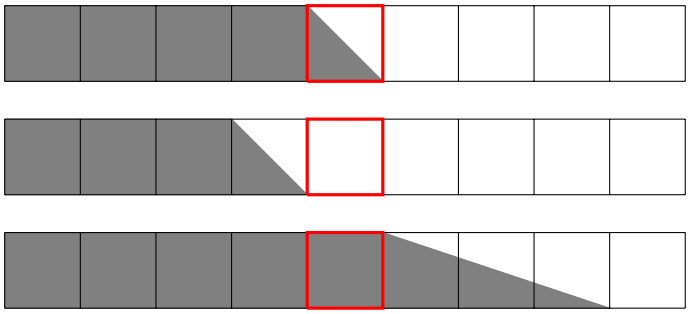

Figure 1: Graphical representation of a coherent surface. The cell highlighted by a red square represents the current cell. The grey color indicates volume fraction.

As shown in figure 1, height values are valid for the current cell (highlighted by a red square) when, within a certain distance, cells fully immersed in the color function (represented by grey) and those with a 0 value both occur. In contrast, figure 2 demonstrates cases where a 'coherent surface' is not observed, so 'nodata' is assigned.

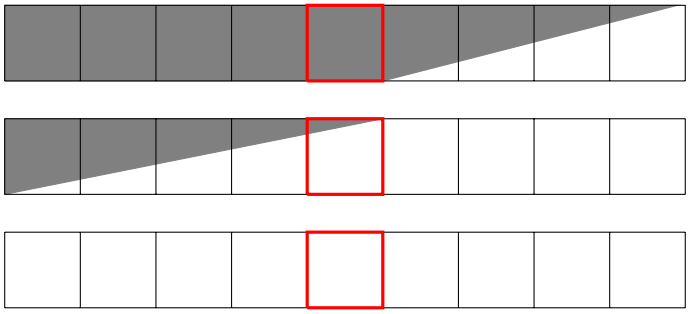

Figure 2: Graphical representation of a non-coherent surface. The cell highlighted by a red square represents the current cell. The grey color indicates volume fraction.

1.2 Height Value and Zero-Value Point

The height value is essentially a summary of the color function, with each cell's value represented at its center. To obtain a specific value, the position where the value is 0 (the zero-value point) must first be defined. Consider the condition illustrated in figure 3, where the surface spans two cells with color function values of 0.6 and 0.1. The zero-value point is located 0.7 units away from the last occupied cell, as indicated by the blue dashed line in figure 3. The value of each cell is then calculated based on the distance between its center and the zero-value point, as shown below the figure. The positive direction points toward the inner side of the surface.

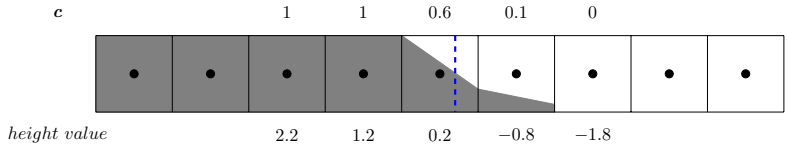

Figure 3: Graphical representation of the zero-value point. The blue dashed line indicates the exact position of the zero-value point. Numbers at the top show the color function of each cell, while those below display the corresponding height value at the center of each cell, marked by a black dot.

1.3 Direction of Surface

Besides the example in figure 3, another possibility exists where the surface extends in the opposite direction (from right to left in this example). To distinguish these cases, the zero-value point is assigned a value of 20 for cases shown in figure 4. Typically, only cells within a 5.5-unit distance from the zero-value point have valid height values. Thus, the range of height values is $[-5, 5]$ or $[15, 25]$, depending on the surface direction.

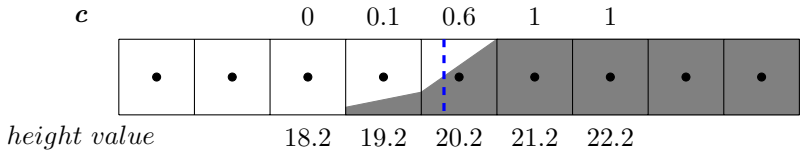

Figure 4: Graphical representation of the zero-value point. The blue dashed line indicates the exact position of the zero-value point. Numbers at the top show the color function of each cell, while those below display the corresponding height value at the center of each cell, marked by a black dot.

Section 2

Functions

2.1 Overall Configuration

Before introducing parameters and workflows, a brief overview of the configuration of the three main functions discussed in the following sections is provided. As shown in figure 5, **heights** acts as a controller, issuing commands to **half_column**, which integrates volume fractions in neighboring cells. After iterating over the eight adjacent cells (four in both positive and negative directions), the height value is assigned to the current cell. Special care is taken for tree grids, where the additional function **refine_h_x** is used to prolongate the color function.

2.2 height and orientation

2.2.1 Worth Mentioning Details

These two functions are 'tricky' functions. As discussed in section 1, to distinguish surface orientation, the final height value has two ranges: $[-5, 5]$ or $[15, 25]$. The **height** function is a reverse function that takes the height value as input, removes the orientation disguise, and returns the true distance between the current cell and the zero-value point, with an output range of $[-10, 10]$. In contrast, **orientation** returns the surface orientation based on the height value. To save memory, these tricky functions are implemented as 'inline' functions, meaning their code is substituted directly when called.

2.2.2 Program Workflow

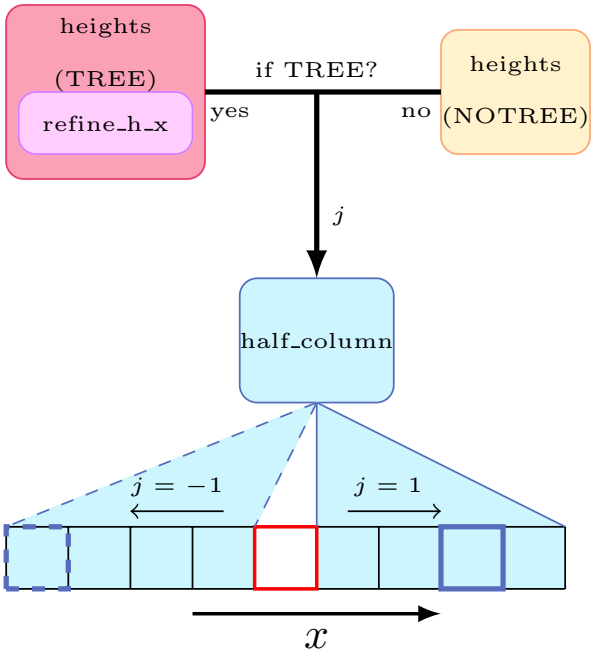

Figure 5: Overall configuration of the current header file. The red square highlights the current cell, and the blue square represents cells scanned by **half_column**.

Starting Point for height
Input: *H* (height value).
The disguise value *HSHIFT* is 20. By adding or removing this value, the true interfacial distance is revealed.

```c
#define HSHIFT 20.

static inline double height (double H) {
  return H > HSHIFT/2. ? H - HSHIFT : H < -HSHIFT/2. ? H + HSHIFT : H;
}
```

Starting Point for orientation
Input: *H* (height value).
Returns `TRUE` or `FALSE` based on the disguise value.
Two layers of ghost values are set on every boundary.

```c
static inline int orientation (double H) {
  return fabs(H) > HSHIFT/2.;
```

```
}

#define BGHOSTS 2
```

2.3 half_column

Based on the previous discussion, **half_column** plays a major role in height computation by scanning neighboring cells and returning the surface status and height value. Due to its complexity, the algorithm is explained part by part.

2.3.1 Parameters

Name	Data type	Status	Option/Default	Representation (before/after)
point	Point	unchanged	compulsory	current cell position
c	scalar	unchanged	compulsory	c (volume fraction)
h	vector	**output**	compulsory	**h**
cs	scalar	unchanged	compulsory	$c[2 * j]$
j	int	unchanged	compulsory	j (direction control)

2.3.2 Purpose of the Function

half_column has two purposes:

1. To check whether a coherent surface (defined in section 1) exists within eight adjacent cells.

2. If a coherent surface exists, compute the corresponding height value by integrating the color function of each cell.

There are three possible situations for the current cell: $f_c = 1$, $f_c = 0$, or $f_c \in (0, 1)$, where f_c is the color function value. For the first two cases, the threshold for a coherent surface is finding a neighbor with $f = 0$ or $f = 1$, respectively. For the third case (a surface cell), both directions must be iterated to find a pair of opposite (full and empty) cells on each side.

Once a coherent surface is found, the remaining task is to calculate the height value. The key is understanding position switching between cells. As shown in figure 6, where f_c, f_r, f_l represent the color function values of the current cell and its neighbors, following the surface direction (defined as a vector pointing from the inner to the outer side of the surface, e.g., figure 3), the relationship between these values is:

$$f_c - (k + 1) = f_r \tag{1}$$
$$f_c + (h + 1) = f_l \tag{2}$$

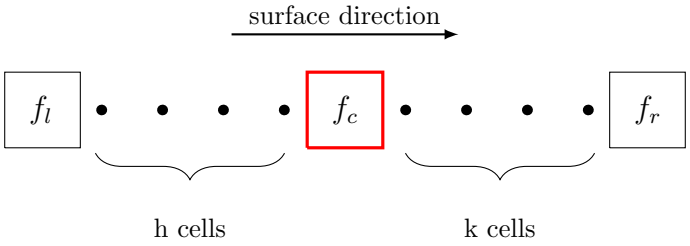

Figure 6: Sketch of position switching between cells. f_c, f_r, f_l represent the color function values of the current cell (red square) and its right/left neighbors, respectively.

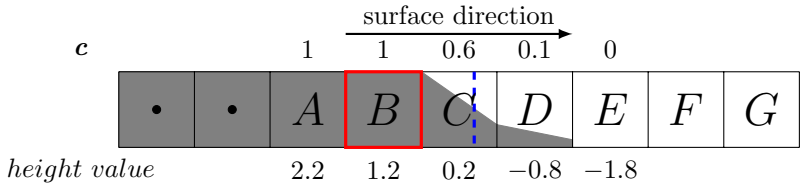

Figure 7: Same example as in figure 3 with two example stencils A, B. The red square highlights the final full cell next to the surface.

Among all cells, the easiest height value to compute is that of the final full cell close to the surface (hereinafter called the final cell). Figure 7 revisits the example from figure 3, highlighting the final cell and surface direction. Based on section 1, the height value of the final cell is computed by integrating the color function along the surface direction until the first empty cell (cell E):

$$h_B = h_B + h_C + h_D + h_E - 0.5 = 1 + 0.6 + 0.1 + 0 - 0.5 = 1.2 \tag{3}$$

The four terms on the right-hand side represent the color function values of cells B, C, D, and E. The -0.5 accounts for the fact that the height value is determined from the cell's center, not its left boundary. Using position switching, the height value of every cell can be obtained. For example, the height value of cell F is $h_F = h_B - 4 = -2.8$, and that of cell A is $h_A = h_B + 1 = 2.2$.

2.3.3 Configuration of the Function

Figure 9 details the configuration of **half_column**, which consists of three layers. As shown in figure 5 and section 2, **half_column** is called twice: first for the negative direction ($j = -1$), then for the positive direction ($j = 1$). Except for the second layer ('iteration'), the other two layers are adjusted based on the direction.

Parameter S records the process status: if a surface is found, the height computation is complete ($S = -1$); if one end is found, $S = 1$ or 0; if no end is found, $S \in (0, 1)$. Since S is defined within **half_column** and cleared when the function ends, a method is needed to preserve results from the negative direction. Basilisk achieves this by encoding the status and height result H into $h[]$ and decoding it at the start of the second cycle ($j = 1$).

In the overall process, decoding (if needed) occurs with initial settings in the first layer. Height values are calculated and stored as H in the second layer by checking neighboring

cells, with S updated accordingly. The output $h[]$ is processed in the final layer, either encoded or directly output, depending on the direction. Detailed explanations follow.

2.3.4 First Layer

The initial status S is derived from the volume fraction of the current cell (highlighted by a red square) and can be 0, 1, or f (empty, full, or surface cell). In the second cycle ($j = 1$), an additional decoding step restores the status and height from the previous cycle into $stats.s$ and $stats.h$. Four decoding scenarios are possible:

1. $h[] = 300$, $(stat.s, stat.h) = (-1, \text{nodata})$: Non-surface cells ($c[] = 1$ or 0) that fail to find a coherent surface, or surface cells that fail to find an empty/full cell in the first cycle.
2. $90 \leq h[] < 190$, $(stat.s, stat.h) = (0, h[] - 100)$: Surface cells that find an empty cell in the first cycle.
3. $190 \leq h[]$, $(stat.s, stat.h) = (-1, h[] - 200)$: Surface cells that find a full cell in the first cycle.
4. $-10 \leq h[] < 90$, $(stat.s, stat.h) = (-1, h[])$: Empty/full cells that find a coherent surface in the first cycle.

For scenarios 2 and 3, S and H are set to $stats.s$ and $stats.h$ to continue searching for the other end. Otherwise, S and H retain their initial values.

2.3.5 Second Layer

As shown in figure 9, there are five categories of scenarios leading to four types of results, stemming from three types of S rather than the current cell's volume fraction:

A $0 < S = f < 1$: Find an empty/full cell, S switches to 0/1, resulting in condition I/II. One end of a coherent surface is found.

B $S = 1$: Find an empty cell, indicating a coherent surface. S switches to -1, completing the iteration (condition III).

C $S = 0$: Find a full cell, indicating a coherent surface. S switches to -1, completing the iteration (condition III).

D $S = 1$ or $S = 0$: Scan a surface cell and return to one with the same status as S. Since the program scans only four neighboring cells, this suggests failure to find a coherent surface in this direction, addressing cases like figure 8. This leads to condition IV, with S unchanged.

Others : All other cases fail to find a coherent surface or at least one end, leaving S unchanged and leading to condition IV.

Height value accumulation occurs in this layer, implicitly shown in figure 9. Following the protocol in section 2.3.2, the height value is computed and adjusted by adding $HSHIFT = 20$ if the surface direction opposes the coordinate, as discussed in section 2.1.

Figure 8: The situation that the second layer occasion D aims to tackle.

2.3.6 Third Layer

For the negative direction ($j = -1$), the output status and height value are encoded into $h[]$ in this layer. The four conditions from the second layer are grouped into three categories:

A Condition IV, encoded as $h[] = 300$: Cases failing to find a coherent surface, marked as 'inconsistent'.

B Condition III, not encoded, $h[] = H$: Cases successfully finding a coherent surface.

C Conditions I and II, encoded as I: $h[] = H + 100$, II: $h[] = H + 200$: Surface cells finding an empty/full cell as one end of a coherent surface.

For the positive direction ($j = 1$), the third layer serves as the final output. Before assigning $h[]$, two scenarios update the data as $(stats.s, stats.h) = (S, H)$:

1. Condition V: Surface cell as the current cell, finding one end in the first cycle.
2. Condition III: Coherent surface found in the second cycle, with the second cycle's height value less than the first cycle's, i.e., $H < stats.h$.

The second scenario ensures that when a cell has two valid height values, the smaller one is used. Since $nodata$ is a large integer in Basilisk, this also covers cases failing in the first cycle but succeeding in the second.

The final height value $h[]$ is assigned based on $(stats.s, stats.h)$, as shown in figure 9.

2.3.7 Program Workflow

First Layer: Initial Setup
Input: $point$, c, h, cs, j.
Initialize $S = f$, $H = f$, where ci and a are used in the second layer. Iterates over each dimension to compute $h.x$, $h.y$, $h.z$.

```
static void half_column (Point point, scalar c, vector h, vector cs, int
 ↪  j)
{
  const int complete = -1;
  foreach_dimension() {
    double S = c[], H = S, ci, a;
```

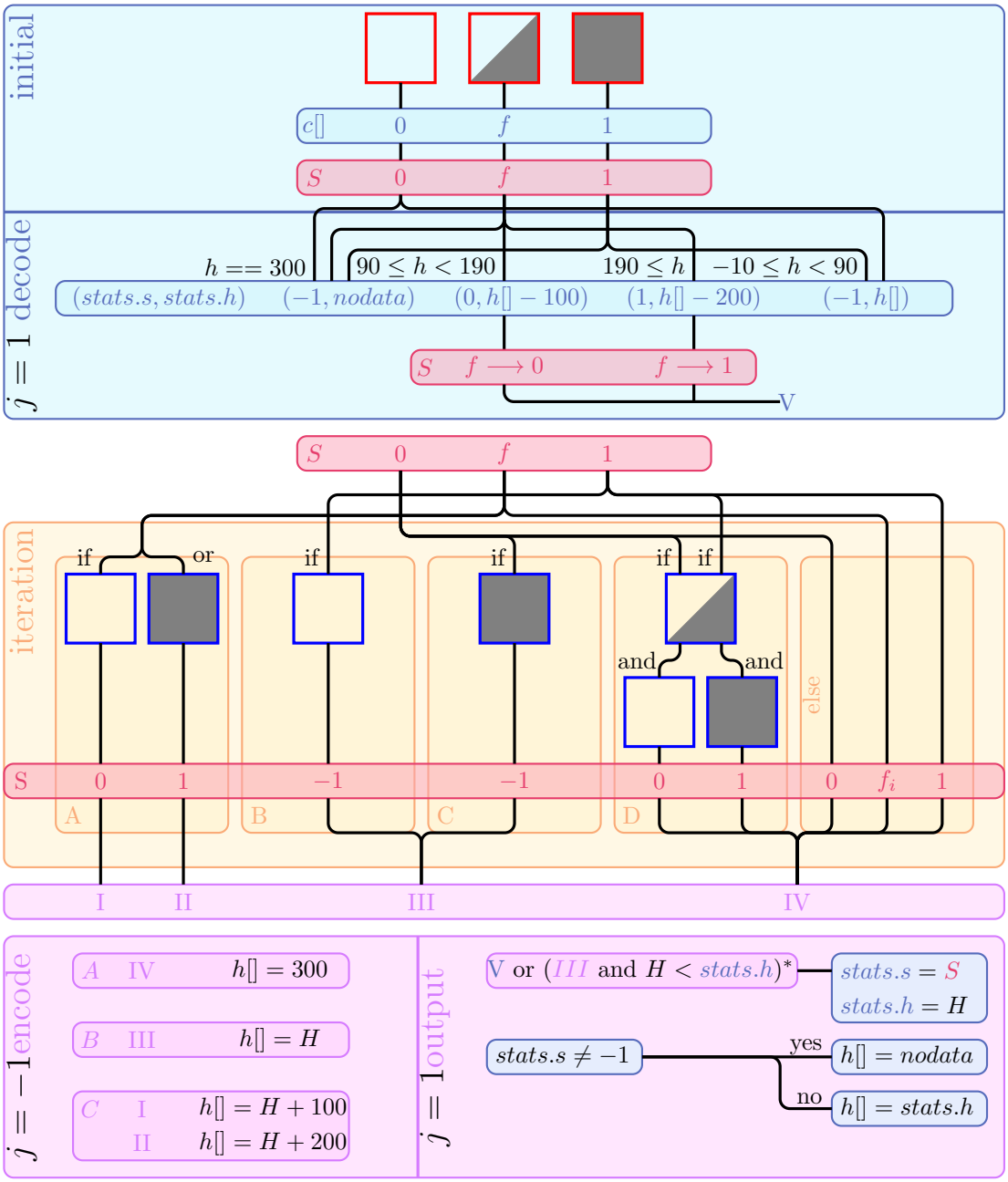

Figure 9: Detailed description of **half_column**. For simplicity, some tricky treatments are hidden. For instance, the judgment in the third layer highlighted by $*$ compares surface distance rather than direct values of H and $stats.h$. Thus, the actual judgment is $fabs(height(H)) < fabs(height(stats.h))$.

First Layer: Decode (Part 1)
If the condition is inconsistent (condition IV), set $state.h = -1$ and begin a new iteration.

```
typedef struct { int s; double h; } HState;
HState state = {0, 0};
if (j == 1) {
  if (h.x[] == 300.)
    state.s = complete, state.h = nodata;
```

First Layer: Decode (Part 2)
Decode cases other than condition IV. Since s is an integer, only cases with $h[] \geq 90$ (conditions II and III) yield non-zero s, restoring previous status as $(S, H) = (state.s, state.h)$. Otherwise (condition III), the iteration uses the original initial values.

```
  else {
    int s = (h.x[] + HSHIFT/2.)/100.;
    state.h = h.x[] - 100.*s;
    state.s = s - 1;
  }
  if (state.s != complete)
    S = state.s, H = state.h;
}
```

Second Layer: Iteration (Part A)
Scan four neighboring cells. If $1 > S > 0$ and an empty or full cell is found ($ci = 0$ or $ci = 1$), update S and modify H based on its position relative to the zero-value point.

```
for (int i = 1; i <= 4; i++) {
  ci = i <= 2 ? c[i*j] : cs.x[(i - 2)*j];
  H += ci;
```

```
      if (S > 0. && S < 1.) {
        S = ci;
        if (ci <= 0. || ci >= 1.) {
          H -= i*ci;
          break;
        }
      }
```

Second Layer: Iteration (Parts B and C)
Part B: For full current cells finding an empty cell, stop iteration, set S to complete, and compute height as discussed.
Part C: For empty current cells finding a full cell.

```
      else if (S >= 1. && ci <= 0.) {
        H = (H - 0.5)*j + (j == -1)*HSHIFT;
        S = complete;
        break;
      }
      else if (S <= 0. && ci >= 1.) {
        H = (i + 0.5 - H)*j + (j == 1)*HSHIFT;
        S = complete;
        break;
      }
```

Second Layer: Iteration (Part D)
If a surface cell is scanned but no coherent surface is found, stop the iteration.

```
      else if (S == ci && modf(H, &a))
        break;
    }
```

Third Layer: Encoding (Part A)
Inconsistent surfaces (condition IV) are encoded as *h[]* = 300.

```
    if (j == -1) {
      if (S != complete && ((c[] <= 0. || c[] >= 1.) || (S > 0. && S <
      ↪  1.)))
        h.x[] = 300.; // inconsistent
```

Third Layer: Encoding (Part B)
Complete cases (condition III) assign *h[]* directly as *H*.

```
      else if (S == complete)
        h.x[] = H;
```

Third Layer: Encoding (Part C)
Partial heights (conditions I and II) are encoded
based on whether an empty or full cell was found.

```
      else
        h.x[] = H + 100.*(1. + (S >= 1.));
    }
```

Third Layer: Output (Part D)
For certain cases, update *state.s* and *state.h*. See section 2.3.6 for details.

```
    else { // j = 1
      if (state.s != complete ||
          (S == complete && fabs(height(H)) < fabs(height(state.h))))
        state.s = S, state.h = H;
```

Third Layer: Output (Part E)
Assign *h[]* based on *state.s* and *state.h*.

```
      if (state.s != complete)
        h.x[] = nodata;
      else
        h.x[] = (state.h > 1e10 ? nodata : state.h);
    }
  }
}
```

2.4 column_propagation

The **column_propagation** function supplements **half_column** by assigning height values to cells within $5.5R$ of the zero-value point that received 'nodata' from **half_column**. For example, cell G in figure 7 is valid and should have a height value of -3.8, but it receives 'nodata' due to failing to find a coherent surface in its four neighbors. **column_propagation** addresses this issue.

2.4.1 Parameters

Name	Data type	Status	Option/Default	Representation (before/after)
h	vector	**updated**	compulsory	**h**

2.4.2 Worth Mentioning Details

Similar to **half_column**, this function iterates two neighbors in both directions and updates the height value based on the spatial relation with the smallest height cell.

2.4.3 Program Workflow

Propagation
Input: $h = \mathbf{h}$.
Search four neighbors for valid height values, then update the current cell's value. The inline function **height** is called to reveal the true distance to the zero-value point.

```
static void column_propagation (vector h)
{
  foreach (serial) // not compatible with OpenMP
    for (int i = -2; i <= 2; i++)
      foreach_dimension()
        if (fabs(height(h.x[i])) <= 3.5 &&
            fabs(height(h.x[i]) + i) < fabs(height(h.x[])))
          h.x[] = h.x[i] + i;
}
```

2.5 heights for Non-Tree Grid

As discussed in section 2.1, **heights** controls **half_column** and **column_propagation** to assign height values to valid cells. The non-tree grid version is presented first, sharing the same structure as the tree grid version.

2.5.1 Parameters

Name	Data type	Status	Option/Default	Representation (before/after)
c	scalar	unchanged	compulsory	color function c
h	vector	**updated**	compulsory	**h**

2.5.2 Worth Mentioning Details

Since the default boundary settings provide only two cells, an additional vector s is created to support iteration over four cells in both directions. The value stored at position $[x_0, y_0]$ for s is:

$$s.x[x_0, y_0] = c[x_0 + 2j, y_0] \tag{4}$$

where j indicates the direction (see figure 5). Each component of s represents a value translation in the corresponding direction.

2.5.3 Program Workflow

Starting Point
Input: $c = c$, $h = \mathbf{h}$.
Boundary Settings: Ensure s has the same boundary settings as c.
Direction Iteration: j represents the current direction, with $j = -1$ (resp. 1) indicating negative (resp. positive) direction.
Value Assignment for s: Implement equation 4.

```
#if !TREE
trace
void heights (scalar c, vector h)
{
  vector s[];
  foreach_dimension()
    for (int i = 0; i < nboundary; i++)
      s.x.boundary[i] = c.boundary[i];
  for (int j = -1; j <= 1; j += 2) {
    foreach()
      foreach_dimension()
        s.x[] = c[2*j];
```

Height Value Assignment
Call half_column: Assign height values for each cell in every direction.
Call column_propagation: Propagate height values to ensure cells within $5.5R$ have valid height values.

```
    foreach (overflow)
      half_column (point, c, h, s, j);
  }
  column_propagation (h);
}
```