heights.h Documentation

Haochen(Langford) Huang

June 1, 2024

version: 2.0

1 Introduction

'heights.h' together with 'parabola.h' serve as toolbox for 'curvature.h' to compute surface curvature in multiphase flow. The aim of the file is to allocate value which represents distance towards surface to every single cell. And the value is named 'height', therefore the name of current headfile. Before diving into details, I shall first introduce several conceptual definitions with 1D example.

1.1 Surface

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

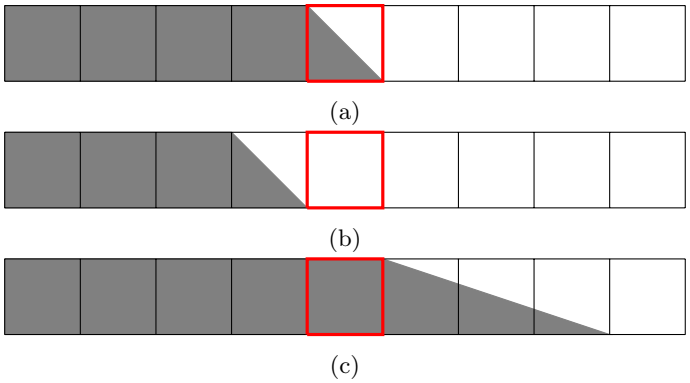

Figure 1: Graphical representation of coherent surface. Cell highlighted by red square represents current cell. Color grey indicates volume fraction.

Figure 1 displays occasions where height value is valid for current cell (the one hightlighted by red square). Within certain distance, cell fully immersed in color function (represents by color grey) and those with 0 value both occur. On the contrary, figure 2 demonstrates occasions in which 'coherent surface' is not observed. Therefore 'nodata' is assigned.

1.2 Height value and 0 value point

Nature of height value is the summary of color function, and the value of each cell is represented by that at center of the individual. In order to obtain specific value, the position where 0 is located should first be defined. Consider a condition as illustrated in figure 3 where surface is indeed two cell with color function of 0.6 and 0.1. 0 point then located at the place 0.7 away from the final occupied cell i.e. blue line in figure 3. Then value of each cell is calculated based on the distance between cell center and 0 position as demonstrated beneath the figure. The positive direction points towards inner side of surface.

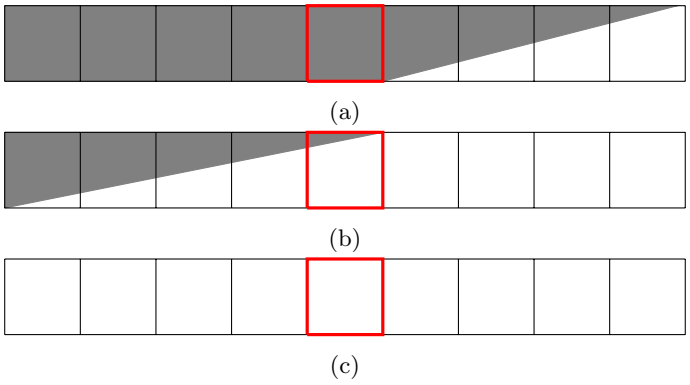

Figure 2: Graphical representation of not coherent surface. Cell highlighted by red square represents current cell. Color grey indicates volume fraction.

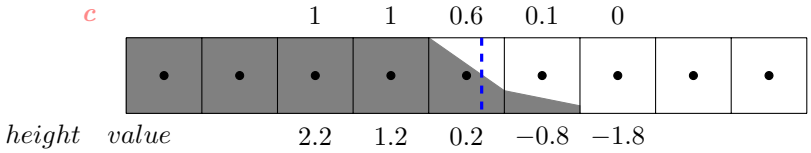

Figure 3: Graphical representation of 0 position. Blue dashed line indicates the exact position of 0 position. Numbers on the top reveals the color function of each cell while those beneath displays corresponding height value at center of each cell as shown by black dot.

1.3 Direction of surface

Except example shown in figure 3 there do exist another possibility that surface stems following opposite direction (from right to left in this example). In order to distinguish these two occasions the value of 0 position is defined to be 20 for cases show in figure 4. Typically, only cells positioned within 5.5 distance away from 0 position has valid height value. Therefore the range of height value is $[-5, 5]$ or $[15, 25]$ based on surface direction.

2 Functions

2.1 Overall configuration

Before introducing parameters and workflow, I shall firstly deliver a brief introduction about overall configuration of three functions which will be discussed in the following sections. Figure 5 depicts the overall function configuration of current headfile. **heights** serves as console and delivers command to **half_column** in which volume fraction integration is conducted on neighbourhood cells. After iterating the adjoining 8 cells (4 for both positive and negative direction), the height value is allocated to the current cell. Note that special care is taken for tree girds and the additional function **refine_h_x** is employed to prolongate color function on tree grid.

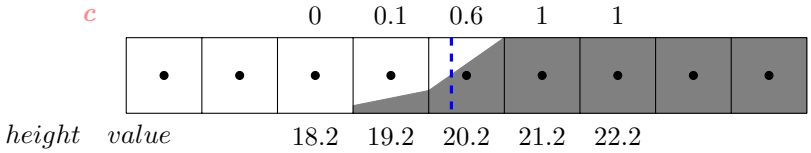

Figure 4: Graphical representation of 0 position. Blue dashed line indicates the exact position of 0 position. Numbers on the top reveals the color function of each cell while those beneath displays corresponding height value at center of each cell as shown by black dot.

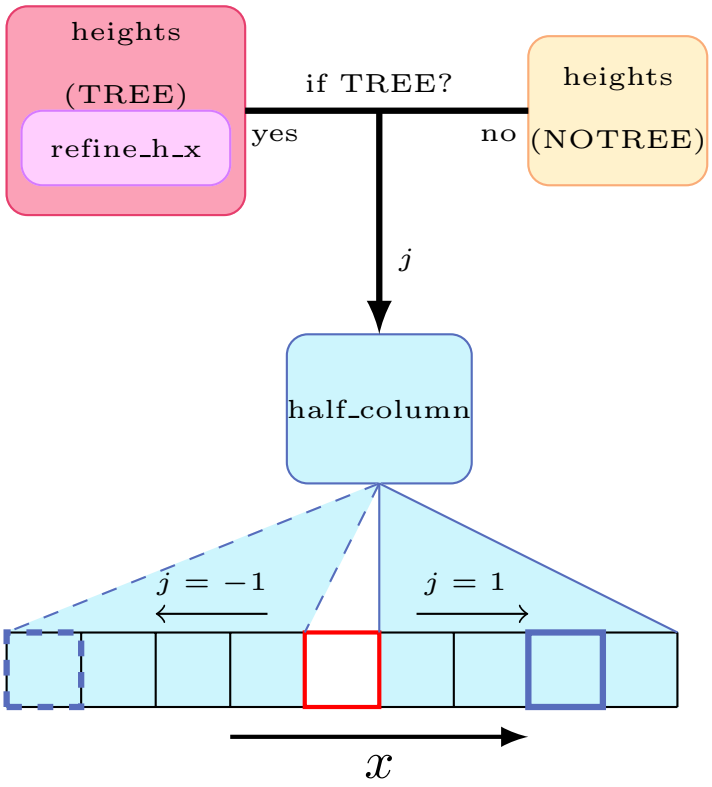

Figure 5: Overall configuration of current headfile. Red square highlights current cell and the blue square represents cell scanned by **half_column**.

2.2 height&orientation

2.2.1 Worth mentioning details

These two functions are 'tricky' functions. As introduced in section 1, in order to distinguish the orientation of the surface, range of final value has two options: $[-5, 5]$ or $[15, 25]$. **height** is the reverse function which takes height value as input, erases disguise of orientation and returns the real distance between current cell and 0 position. The range of corresponding output turns out to be $[-10, 10]$.

By contrary **orientation** returns surface orientation based on height value. To save memory, tricky function manifests as 'inline' function, i.e. substitute corresponding code when it is called.

2.2.2 Workflow

height
The disguise value *HSHIFT* is 20. By adding/removing such value, true interfacial distancne is revealed.

```
#define HSHIFT 20.

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

orientation
return TRUE or FALSE based on disguise value. Two layers of ghost value is set on every boundaries.

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

#define BGHOSTS 2
```

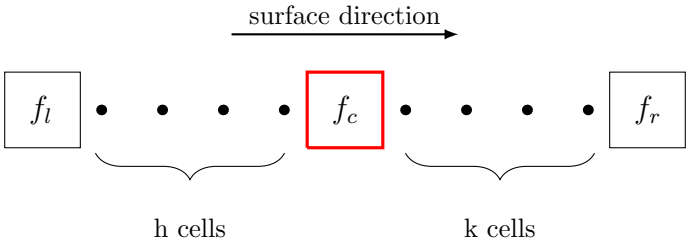

Figure 6: Sketch of position switching between cells. f_c, f_r, f_l represent color function value of current cell (red square) and its right/left side neighbor, respectively.

2.3 half_column

Based on previous discussion **half_column** plays major role in height computing whose duty is to scan over neighborhood cells and feedback status value of surface finding as well as height value. Owing to its complexity, I shall carefully interpret the algorithm part by part instead of briefly introducing selected details.

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 there is a coherent surface (defined in section 1) within 8 adjoining cells.
2. If there is coherent surface, compute corresponding height value by integrating color function of each cell.

There are three possible situations for current cell: $f_c = 1, f_c = 0, f_c \in (0, 1)$. Where f_c represents color function value of current cell. For first two situations, the threshold for coherent surface is to find certain neighborhood in which $f = 0, f = 1$ respectively. However for third condition i.e. the cell located on surface, one should iterate both directions to find a pair of opposite (full and empty) cells on each side.

Once coherent surface is found, the problem remaining is how to calculate the height value. The key to understand such process is position switch between cells. Figure 6 displays an example of such condition, where f_c, f_r, f_l represent color function value of corresponding cell. Following the surface direction (defined as vetor points from inner side to outside of surface, e.g. figure 3.), relationship between three values reads

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

Among all the cells, the one whose height value is easiest to obtain is the final full cell close to the surface (hereinafter as final cell). Figure 7 demonstrates same example in figure 3 but highlight the final cell and corresponding surface direction. Based on discussion in section 1, the height value of final cell can be compute by integrating the color function along surface direction till the very first empty one (cell E) reads:

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

Three terms in R.H.S represents color function of cell B, C, D, E. While the -0.5 stems from the fact that height value of one cell is determined from its center instead of left boundary. Combined with the

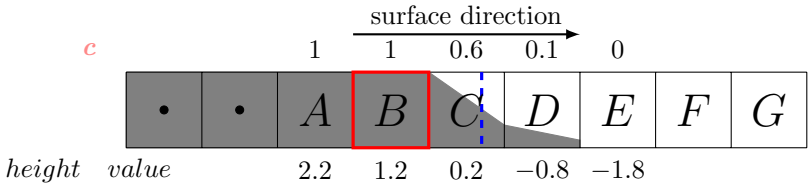

Figure 7: Same example as in figure 3 with two example stencils A, B. Red square here used to highlight the final full cell next to surface.

position switch discussed previously, we can then obtain height value of every cell. In short, to obtain a height value, one should first switch to 'final cell' and integrate through surface. For example, height value of cell F yields $h_F = h_B - 4 = -2.8$ and that of cell A yields $h_A = h_B + 1 = 2.2$.

2.3.3 Configuration of the function

Figure 9 carefully depicts configuration of **half_column** which consists of three layer. As illustrated in figure 5 and section 2, **half_column** will be called twice during the process, first for negative direction ($j = -1$) then for positive direction ($j = 1$). Except for second layer 'iteration', the rest two will be altered according to direction.

Parameter S is employed to record status of process: if a current surface is found then the height function is complete (S=-1) or one end is found (S=1 or S=0) or no end is found ($S \in (0, 1)$). Note that S is defined within **half_column** and thus will be cleared once function ends therefore a method is needed to preserve results of negative direction. Basilisk achieve such goal by encoding status along with current height result H into $h[]$ and decode such value to restore at beginning of second cycle when $j = 1$.

Coming back to the overall process, the decode step, if needed, is carried out along with initial settings in the first layer. After which height value is calculated and stored as H in the second layer 'iteration' by checking neighborhood cells. S is also altered with respect to the iterating results. $h[]$ which serves as output is processed in the final layer. Depending on the direction, it will be encoded or directly output. Careful introductions are presented in the following sections.

2.3.4 first layer

There are three possible occasions for S as initial status obtained from volume fraction of current cell (highlighted by red square), i.e. $0, 1, f$ corresponds to empty, full and surface cell. If is in the second cycle ($j = 1$), an additional step is carried out to decode $h[]$ from which status and height value obtained in previous cycle can be restored in $stats.s$ and $stats.h$ respectively.

There are totally four occasions which will be encountered in decoding process:

1. $h[] = 300, (stat.s, stat.h) = (-1, nodata)$: Those whose current cell is not surface cell ($c[] = 1$ *or* 0) and fails to find coherent surface, or current cell is surface cell and fails to find empty/full cell during first cycle.

2. $90 \leq h[] < 190, (stat.s, stat.h) = (0, h[] - 100)$: Current cell is surface cell and find empty cell during first cycle.

3. $190 \leq h[], (stat.s, stat.h) = (-1, h[] - 200)$: Current cell is surface cell and find full cell during first cycle.

4. $-10 \leq h[] < 90, (stat.s, stat.h) = (-1, h[])$: Current cell is empty/full cell and find coherent surface during first cycle.

For occasion 2 and 3, S and H are assigned with $stats.s$ and $stats.h$ and continue its journey on finding another end to accomplish a coherent surface. Otherwise S and H preserve to be the outcome of initial.

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

2.3.5 second layer

As depicted in figure 9, there are overall 5 categories of occasions and lead to 4 kinds of results. Note all those occasions stem from three types of S instead of volume fraction of current cell.

A $0 < S = f < 1$: Find empty/full cell, S switch to 0/1 and results in condition I/II respectively. One end of coherent surface has been found.

B $S = 1$: Find empty cell which indicates occurrence of coherent surface. S switch to -1, the iteration completes and leads to condition III.

C $S = 0$: Find full cell which indicates occurrence of coherent surface. S switch to -1, the iteration completes and leads to condition III.

D $S = 1$ *or* $S = 0$: Scan over a surface cell then returning to the one with the same status as S. Consider the program is set to scan over only four neighborhood cells, such occasion suggests failure of finding coherent surface on this direction. Indeed, such filter is set to tackle situation demonstrated in figure 8. This leads to condition IV and S keeps unchanged.

others : Except occasions have been mentioned, the rest all fail to find coherent surface or at least one end of it (for surface cell). These occasions do not change S and leads to condition IV.

The accumulation of height value is carried out in this layer but expresses implicitly in figure 9. Following the protocol introduced in section 2.3.2, the height value is first output and added by $HSHIFT = 20$ if surface direction is against coordinate, as discussed in section 2.1.

2.3.6 third layer

If direction is negative ($j = -1$), the output status along with height value will be encode into $h[]$ in this layer. Four kinds of conditions output by second layer will be divided into three kinds of categories.

A condition IV, height value is encoded as $h[] = 300$: The cases that fail to find coherent surface marked as 'inconsistent'.

B condition III, height value is not encoded $h[] = H$: Those cases successfully find coherent surface.

C condition I and II, height value is encoded as I: $h[] = H + 100$, II: $h[] = H + 200$: Cases whose current cell is surface cell and find empty/full cell as one end of coherent surface.

If direction is positive ($j = 1$) the third layer then acts as the final output. Before finally assigning $h[]$, two occasions

1. condition V: Surface cell as current cell and find one end during first cycle.
2. condition III: coherent surface is found in second cycle and height value in second cycle appears less than the previous cycle i.e. $H < stats.h$.

update data as $(stats.s, stats.h) = (S, H)$. The second occasion indicates when single cell obtains two valid height value, the less one will be admitted. Since $nodata$ defined in Basilisk is an extreme large integer, occasion 2 also covers situations which fails to find coherent surface in first cycle but succeed in the second cycle.

Final height value $h[]$ therefore assigned according to $(stats.s, stats.h)$ as shown in figure 9.

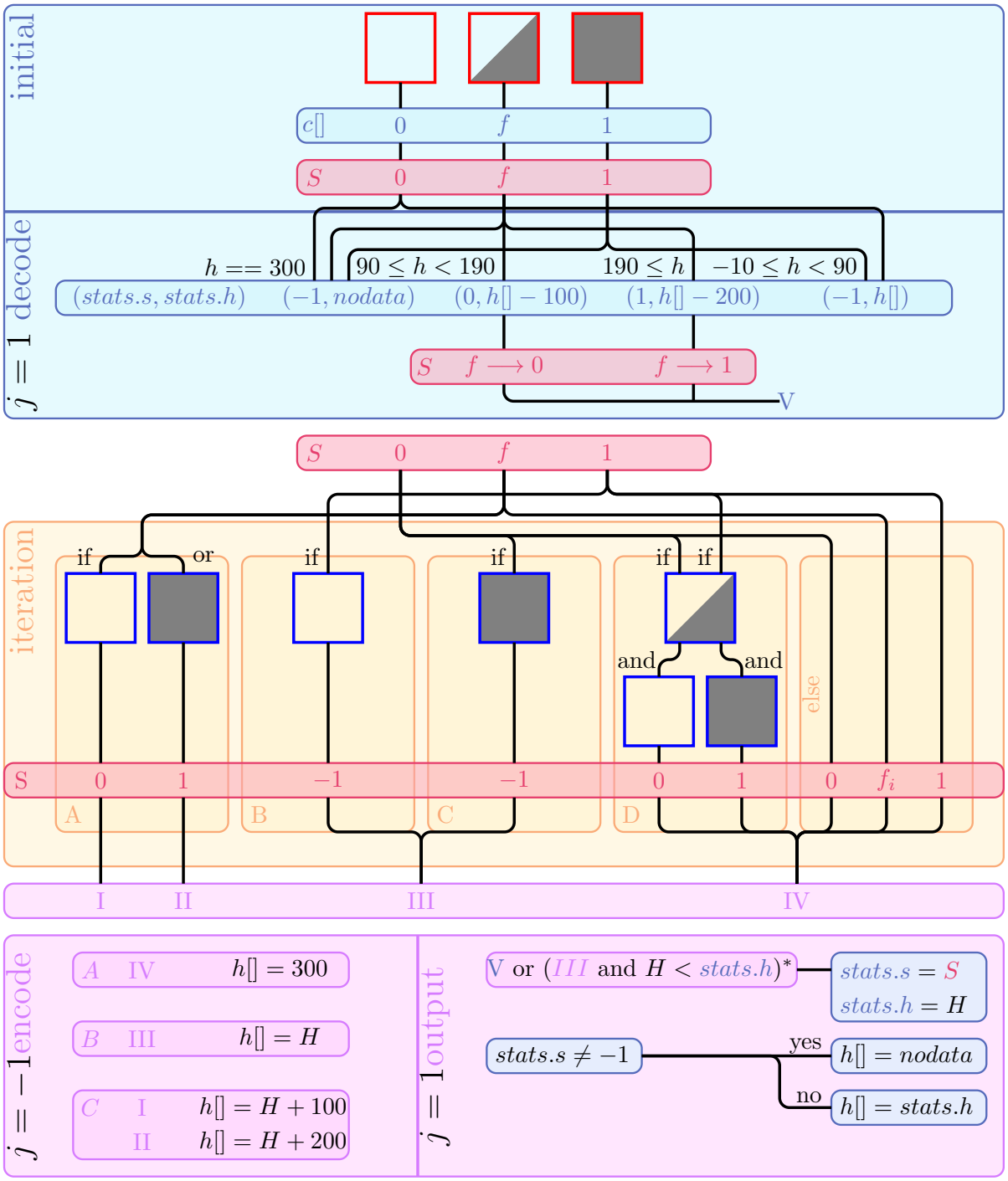

Figure 9: Detailed description about **half_column**. For sake of simplicity, some tricky treats have been hidden. For instance judgement in third layer highlighted by $*$ compares surface distance instead of direct value of H and $stats.h$. Therefore the actual form of judgement reads $fabs(height(H)) < fabs(height(stats.h))$.

2.3.7 Program Workflow

1. First Layer
A. initial:
$S = f$ $H = f$, ci and a are employed in the second layer. Note every dimension is iterated to obtained $h.x$, $h.y$ and $h.z$.

```
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;
```

1. First Layer
B. decode1:
If the condition is incosistent (condition IV), the status is recorded as $state.h = -1$ 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;
```

1. First Layer
C. decode2:
Cases except condition IV will be decode accordingly.
Note since s is defined as integer, only cases whose $h[] \geq 90$, i.e. condition II and III in figure 9, can obtain non-zero s then restoring previous status as initial condition $(S, H) = (state.s, state.h)$. Otherwise (condition III) the iteration begins with original initial.

```
      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;
    }
```

2. Second Layer
A. iteration A:
Scanning over 4 neighborhood cells. If $1 > S > 0$ and meet an empty or full cell ($ci = 0$ or $ci = 1$), S first updated accordingly then modifying H according to its position against 0 position.

```
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;
    }
  }
```

2. Second Layer
B. iteration B:
For those whose current cell is full and find empty cell, the iteration stops and S switches to complete. Height value is computed as discussed before.
C. iteration C:
For those whose current cell is empty and find full cell in iteration.

```
  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;
  }
```

2. Second Layer
D. iteration D:
For those have scanned over surface cell but never penetrate and find coherent surface, the iteration will stop right away.

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

3. Third Layer
A. encode A:
Incosistent surface i.e. condition IV encode with $h[]{=}300$.
B. encode B:
Complete cases i.e. condition III assign $h[]$ directly as H.

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

3. Third Layer
C. encode C:
Partial height (condition I and II) is encoded depending on whether empty of full cell they have found.

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

3. Third Layer
D. output 1:
For certain situation *states.s* and *states.h* are updated. 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;
    }
```

3. Third Layer
E. output 2:
Output *h[]* given *states.s* and *states.h*.

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

2.4 column_propagation

Function **column_propagation** serves as supplement of **half_column** which allocates height value to those located within $5.5R$ from the 0 position but gain *nodata* from **half_column**. Take cell G in figure 7 as an example, such cell is indeed valid and should be assigned with -3.8 as its height value. However, owing to failure of finding coherent surface in its four neighbors, the cell is actually assigned with *nodata*. **column_propagation** is therefore built for tackling such issue.

2.4.1 Parameters

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

2.4.2 Worth Mentioning Details

Similar to the process in **half_column**, current function will iterate two neighbors in both directions and update height value base on spatial relation with the smallest height cell.

2.4.3 Program Workflow

Propagation
A. input:
h = **h**
B. propagation
Finding in the four neighbors for valid height value, then updating the value on current cell. inline function **height** is called to reveal real distance against 0 position.

```
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 introduced in section 2.1, **heights** serves as a controller over **half_column** and **column_propagation** to assign height value to valid cells. The non-tree grid version is first introduced which shares the same construction with 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	**update**	compulsory	**h**

2.5.2 Worth Mentioning Details

Since there are only two cells for the boundary in default settings, an additional vector s is create to meet the need for iterating 4 cells in both direction. In particular, the value stored in position $[x_0, y_0]$ for s is

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

where j is the parameter implies the direction (see figure 5). Notably, each components of s indicates value translation of corresponding direction.

2.5.3 Program Workflow

1. Starting Point
A. input:
$c = c$, h = **h**
B. boundary settings
Make sure the boundary settings of s is the same as c
C. direction iteration
j represents current iterating direction for $j = -1$ (resp. 1) indicating negative (resp. positive) direction.
D. value assignment for s
Implementation of 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];
```

2. Height Value Assignment
E. half_column:
Assign height value for each cell on every direction.
F. column_propagation
Propagate height value to ensure cells within $5.5R$ are equipped with valid height value.

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