Control of a Crystallization Process

http://people.sc.fsu.edu/~jburkardt/presentations/crystal_1996.pdf

.....

John Burkardt
Mathematics Department
Virginia Tech, Blacksburg, VA

February 2, 2024

Abstract

This report summarizes my attempts to introduce a numerical control algorithm into the crystallization simulation in the $MASTRAPP2D$ code.

1 Initial Control Choices

The two big choices to make are the inputs and outputs of the control process, that is, what physical quantities will be assumed to be controllable, and what physical quantities will measure the success of the control.

At the kickoff project meeting, it was clear that the full specification of the crystallization simulation would not be determined for some time, but that it would be profitable to gain some initial experience with the currently available simulation code, $MASTRAPP2d$. For this testing and developmental phase, it was decided that the quantity to control would be a cost functional which represented the total velocity magnitude in the melt region, represented as:

$$\mathcal{J}_1 \equiv \int_{t_0}^{t_1} \int_{\Omega} (u(x,y,t)^2 + v(x,y,t)^2)^{\frac{1}{2}} \, d\Omega \, dt, \tag{1}$$

and the physical quantity to be varied would be the shape of the bottom of the crucible, with the goal of finding a crucible shape that produced a flow that minimized $\mathcal{J}_1$.

An approximation to the cost functional was easily computable within the program, because the velocity (u, v) is computed at every primary node, as well as the area of the control volume surrounding that primary node, so that a simple summation gives a suitable estimate.

The shape of the crucible bottom was represented in the program as a set of coordinates for nodes, (xb_i, yb_i). These nodes defined a cubic spline which in turn determined the location of the corner and primary nodes, and the control

volumes. It seemed logical to fix the values of the xb_i coordinates, and allow the local heights yb_i to vary.

Thus, the control problem could be posed as a simple optimization; we would regard the quantity $\mathcal{J}_1$ as an implicit function of the crucible bottom shape, and seek the values yb_i which minimize $\mathcal{J}_1$. Such a problem could easily be handled by any package for unconstrained optimization of a scalar function of several arguments.

2 Computational Results

The first attempts at implementing the algorithm proceeded for several optimization steps, and then suddenly failed because of floating point overflows. These errors occurred in the portion of the code that computes the physical quantities, which include the velocity (u, v), but also temperature, pressure, and so on. It was suspected that these problems depended, in turn, on some peculiarity of the geometry, and this was verified by plotting the row of control volumes that lay along the crucible boundary. For the failing values of yb_i, this row reached a point where the quadrilateral volume became a degenerate triangle, because the mesh generator allowed an interior node to coincide with the boundary. This happened again for the next interior node, so that the next control volume had practically zero area.

One set of parameters that caused this behavior was:

$$yb = (0.18, 0.30, 0.29, 0.28, 0.27, 0.26, 0.255). \tag{2}$$

In Figure **??**, the overall region corresponding to this set of data is shown, with the control volumes and crucible nodes displayed as well. The region of trouble lies along the crucible boundary, a little to the left of center.

A closeup of the problem region is shown in Figure **??**, and shows in detail how badly the grid behaves near the problem area.

No explanation for this problem could be found, but it only seemed to occur when some of the values of yb_i rose above the height of the extreme right node, which was 0.25. Therefore, a new optimization code was employed, which allowed all the parameters yb_i to be restricted to lie between 0.0 and 0.25. This attempt seemed to succeed; that is, the optimization claimed to have converged. However, a plot of the resulting solution showed a very surprising fact: as shown in Figure **??**, the crucible bottom and the mesh had parted. At one portion of the crucible boundary, a particular node had sunk below its neighbors. The mesh generating algorithm specifically searches for the two nearest boundary nodes, and does not require that they be consecutive. Therefore, it constructed a mesh that completely ignored a small portion of the flow region associated with the depressed node. This made the optimization results meaningless.

From these experiences, it seemed that any variation of the original crucible boundary would have to be done with nodes that were monotonically increasing from left to right. This would automatically enforce the range limits used in the previous calculation, but also rule out the possibility of any node sinking

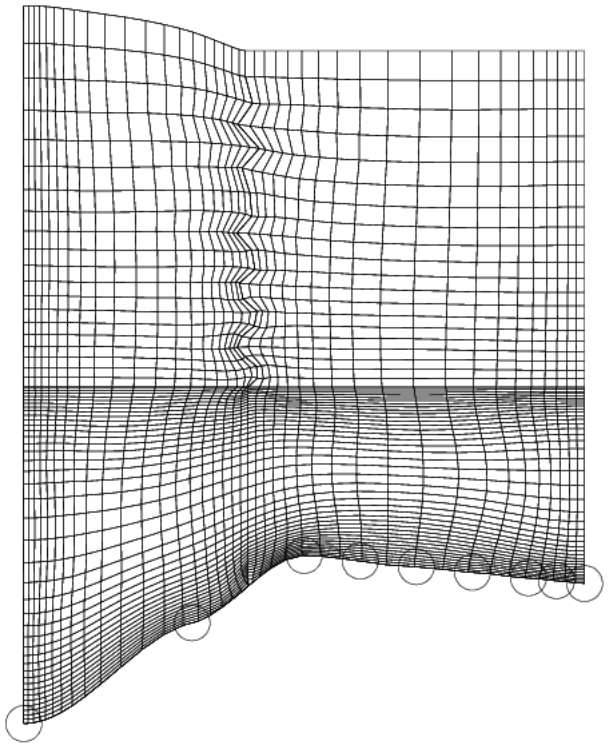

Figure 1: The grid generator creates degenerate control volumes.

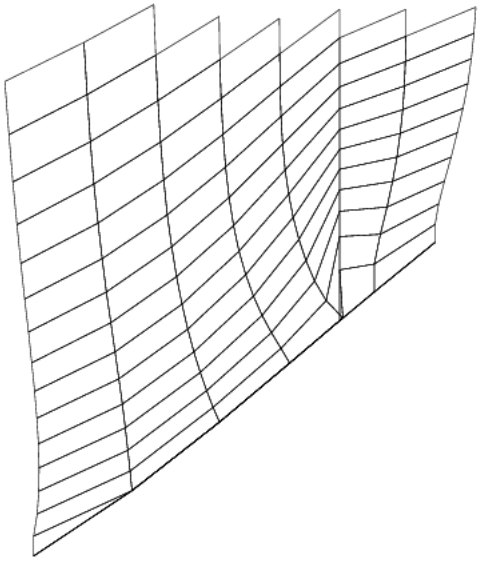

Figure 2: A closeup of the degenerate control volumes.

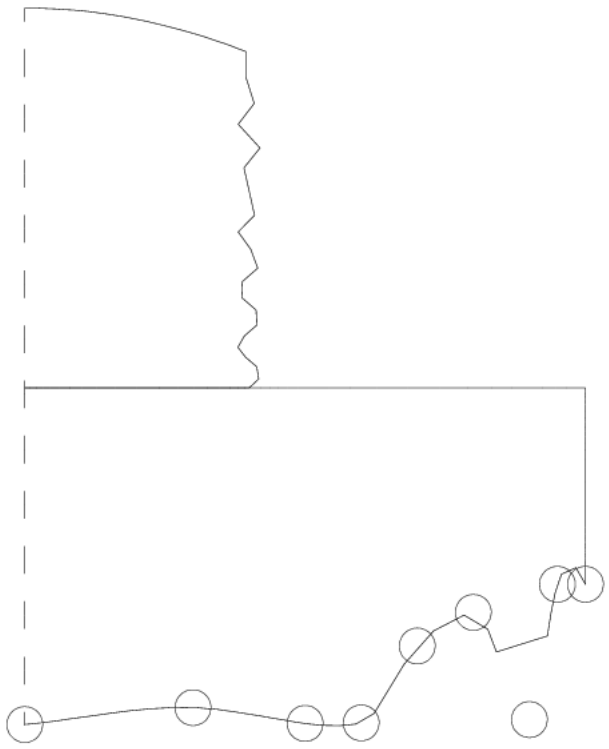

Figure 3: The grid generator misses the boundary..

The circles are the nodes generating the crucible boundary;
The grid generator redraws the boundary, missing one node.

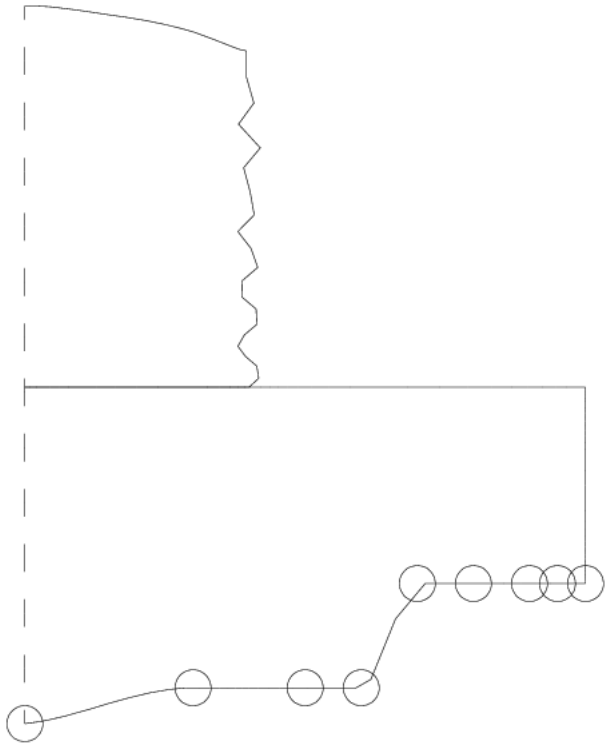

Figure 4: The optimal shape with monotonic nodes.

below the line defined by its two neighbors. With these strong restrictions, the optimization code was able to find a minimizing set of parameters, defining a crucible shape that suggested a pair of flat steps, as shown in Figure **??**.

3 Prospects for Shape Control

I have had discussions with Hui Zhang, the author of the code, who explained that his mesh generator could not handle a universal set of crucible shapes, but rather was tailored to a quite specific shape generated by the original values of the boundary data (xb_i, yb_i). He also said that a code to handle general shapes would take a long time to build. Wing Chui, who has been working with the code in an effort to create a parallel version, also said that it makes very strong assumptions about the boundary when generating the grid.

From my experiences and discussions, it seems that computational variation of the crucible shape can only be carried out under severe restrictions. To add this capability to the program would require changes to the automatic grid

generator. In particular, the grid generator should not rely on the (nonphysical) location of the nodes that define the cubic spline, but only on the actual shape of the boundary curve, as specified by the user; perhaps the user would be allowed to identify points of high curvature, or places where the mesh generator would be required to place a node. The crucible boundary curve should not be restricted to monotonicity or bounded in variation. Such restrictions can be handled by the optimization code, if desired, but the mesh generator should not impose them. In any case, if input data violates any implicit assumptions of the mesh generator, it should detect them, print a warning message, and stop.

4 Using Temperature as the Control

An effort was made to find some other physically reasonable quantities to vary and control. It seemed reasonable to try using the quantity TW, which in the $MASTRAPP2d$ program represents the uniform temperature of the crucible. This quantity is easily measured and controlled experimentally, and it is the most obvious means by which the behavior of the system is affected.

Instead of trying to minimize the total velocity magnitude, it was decided instead to try to achieve some desired velocity magnitude. This would allow us to seek a solution which had enough flow to keep the melted region mixing, while avoiding solutions with very high velocity spikes that might harm the steady accretion of the crystal.

For any time t, we defined the total deviation from the desired average velocity as:

$$f(t) \equiv (\int_{\Omega} (V_{ave} - (u(x,y,t)^2 + v(x,y,t)^2)^{\frac{1}{2}})^2 d\ \Omega)^{\frac{1}{2}} \tag{3}$$

which allowed us to define the cost functional as:

$$\mathcal{J}_2(TW) \equiv \int_{t_0}^{t_1} f(t)\ dt. \tag{4}$$

We then set the initial value of TW to 1712 degrees, and the desired average velocity V_{ave} to 1850, which I came up with as a figure not too far from the average velocity of the original data in Zhang's example problem. For the time integration, I only took two time steps. The cost function began at $\mathcal{J}_2(1712) = 1132.75$, and after 13 optimization steps, we reached a solution with $TW = 1683.90$ and $\mathcal{J}_2(1683.90) = 1097.23$, a small but respectable drop.

For a problem with a single parameter and a reasonable starting guess, this behavior can be taken as typical; the number of optimization steps should be about the same even if the time steps were increased to the value of 200 that Zhang recommends for a production run.

5 Open Questions

The initial efforts have shown that a variety of cost functionals and controls can be chosen, and that, barring failures of the simulation algorithm, an optimizing set of controls can be found. As mentioned earlier, however, these efforts have been experimental, and have chosen the costs and controls somewhat arbitrarily. Before further development efforts are made, it is important to decide what the costs and controls really should be, and to make the simulation code more flexible. In particular:

- First, it is necessary to decide on the physical quantities or costs to be controlled. These could be integrals over the region, or the curvature or length of some interface, or the amount of energy expended; moreover, some weighted combination of such quantities could be handled. It is only necessary to clearly detail what the quantities are, and how they are related to data computed within the program.

- Secondly, the physical quantities that represent the controls must be chosen. For meaningful computations, these should represent items that can actually be varied experimentally. Moreover, the program must be able to represent such quantities with a discrete set of data, and solve the state equations that correspond to a wide range of values of that data.

- Thirdly, the simulation code must be made more robust, flexible and general. The code has broken down repeatedly when handling variations of the original crucible specification. While it may be decided not to vary the crucible, there are many other parts of the code which are currently "hard-wired", particularly geometric quantities, including the location and shape of the crystal, the location of the melt interface, the total number of grid lines, and the number of grid lines drawn in the crystal and melt subregions. Even to do a problem with a finer grid requires a fair amount of tinkering with obscure lines of code. During the optimization of a particular problem, a variety of physical states must be computed, and it is necessary that the code be able to handle a wide range of problem data smoothly. Since the boundary conditions, physical parameters, and geometric specifications are also likely to be of interest, the code should be modified to make them easy to specify and vary.