Thresholding Method for simulating the alternate KWC model Code User Manual

Jaekwang Kim, Matt Jacobs, Nikhil Chandra Admal

May 18, 2021

1 Introduction

This is the user manual for *GB thresholding method*, the details of which is illustrated in the numerical scheme suggested in the white paper [1]. A `C++` based template library is designed to simulate grain growth governed by KWC (Kobayashi–Warren–Carter) model [2, 3, 4].

The KWC grain boundary model of material free energy density $\mathcal{W}_{\text{kwc}}$ is defined by two field variables: the crystal-order phase parameter η and the orientation-phase parameter θ. We write the alternative form of $\mathcal{W}_{\text{kwc}}$ in the white paper [1] takes argument of continuous function η and piecewise constant function θ

$$\mathcal{W}_{\text{kwc}}[\eta,\theta] = \int_{\Omega} \left[\frac{(1-\eta)^2}{2\epsilon} + \frac{\epsilon}{2}|\nabla\phi|^2 + g(\eta)\mathcal{J}\big([\![\theta]\!]\big)\delta(x-x_0) \right] dV, \tag{1}$$

where δ is the dirac-delta function, ϵ the length scale parameter, $[\![\theta]\!]$ the jump of θ, x_0 the position of grain boundary. We take a logarithmic $g(\eta) = -\log(1-\eta)$. Lastly, the function $\mathcal{J}\big([\![\theta]\!]\big) : \mathbb{R}^+ \to \mathbb{R}^+$ is the non-diffused part of grain boundary energy [1].

We will evolve a given polycrystal system by repeatedly solving the two optimization problems:

$$\eta^{k+1} = \arg\min_{\eta} \mathcal{W}[\eta,\theta^k] \tag{2}$$

$$\theta^{k+1} = \arg\min_{\theta} \mathcal{W}[\eta^{k+1},\theta] \tag{3}$$

We employ two separate numerical methods for each problem: we use the Primal-dual algorithm [5, 6], for the first η-sub problem and thresholding method for the second θ-sub problem. The resulting solution describes the motion by curvature with grain boundary energy with arbitrary function of misorientation.

2 Simulation Procedures

The numerical code is developed in the form of `C++` header-template library, which a user can freely refer. `C++` function & class are categorized into different header files based on their task, see the list of header files in Table 1.

To simulate an evolution of a polycrystal, a user first needs to define initial crystal on a regular rectangular domain. Several initial grain configurations such as bicrystal, tricrystal, or randomly

[1] In our approach, the total grain boundary energy of KWC model γ_{kwc} has two parts. The contribution from η and its gradients appears as diffused energy over some finite length scale, and the contribution from $\mathcal{J}$ remains sharp.

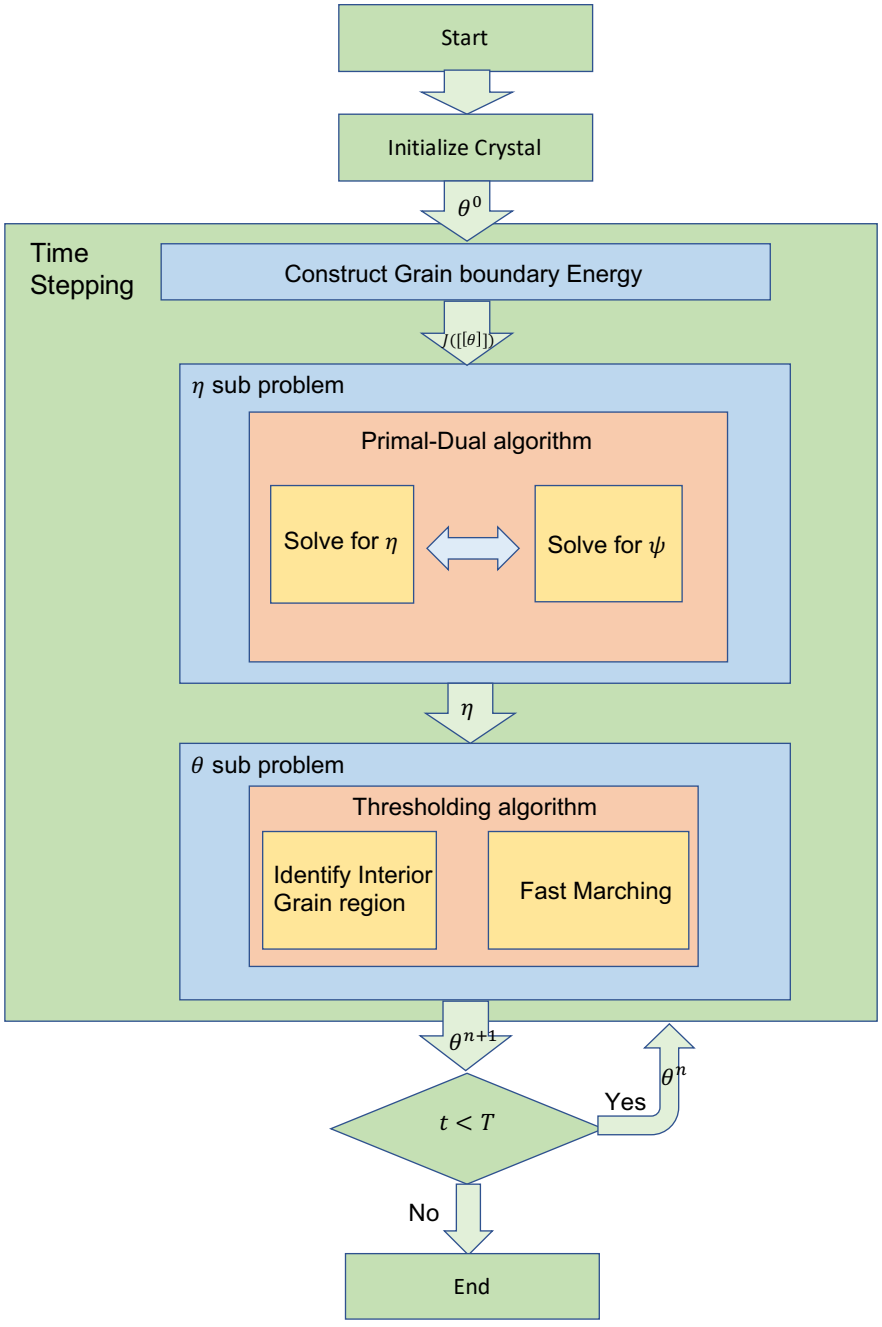

Figure 1: Algorithm Flow chart

Header file	Contents
`InitCrystal.h`	Functions that set the initial condition of simulation, i.e. initialize $\theta(x)$
`DataOut.h`	Functions related to output solution for visualization
`Material.h`	A class that defines GB energy $\mathcal{J}(\llbracket\theta\rrbracket)$ of different types of material
`PostProcessor`	Functions that compute the KWC free energy of polycrystal
`PrimalDual.h`	A class that solves η-sub problem at a given $\theta(x)$ configuration
`KWCThresholding.h`	A class that solves θ-sub problem at a given $\eta(x)$ configuration
`KWCJumpFunction.h`	A class that design $\mathcal{J}$ from a provided external GB data
`Metrics.h`	A list of simple mathematical functions

Table 1: Table of header files

distributed polycrystal can be easily constructed using functions defined in `InitCrystal.h`. Next, a user is required to define the jump function $\mathcal{J}(\llbracket\theta\rrbracket)$. The `Material` class defined in `Material.h` is useful in this step. In the following, the two algorithm classes, `PrimalDual` and `KWCThreshold`, should be constructed to solve η-sub problem (2) and θ-sub problem (3) respectively. Once they are ready, simulation will run until it reaches to a given final time. The flow chart of the overall algorithm is summarized in Figure 1.

3 An example: 2-D polycrystal simulation

In this section, we will look into an example code file `KWC_Polycrystal.cpp` in `E5` folder and show how to execute this file. This example code simulates grain boundary motion under given $\mathcal{J}(\llbracket\theta\rrbracket)$, which is fitted for FCC [110] symmetric tilt grain boundary energy. Because this simulation includes the most of features of the developed code, it will be helpful to understand how the procedures in Fig. 1 are being implemented using the template library.

3.1 Implementation file

We will read the implementation code file `KWC_Polycrystal.cpp` block by block. Bullet points highlights the purpose of each block.

- First, we begin by listing the required header files

```
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <time.h>
#include <float.h>
#include <stdio.h>
#include <assert.h>
#include <iostream>
#include <vector>

//Relevant Library
#include "DataOut.h"
```

```
#include "InitCrystal.h"
#include "PostProcessor.h"
#include "PrimalDual.h"
#include "KWCThresholding.h"
```

- Then, in the `main` function, we define domain size.

```
int main(int argc, char *argv[]){

  //Read global variables from bash
  //Define grid size and set model parameter epsilon
  int n1=atoi(argv[1]);
  int n2=atoi(argv[2]);
  int n3=atoi(argv[3]);
  double epsilon=atof(argv[4]);

  int const DIM=3;

  const unsigned int lcount=50;

  int pcount = n1*n2*n3; //Total number of grid point
  double dt= epsilon * epsilon; //initial choice of dt

  int Nthread = 1; //Total number of threads to be used
```

Note that we will have to pass the domain size N_x, N_y, N_z and the parameter ϵ from the command line when we execute the code. In our thresholding scheme, the choice of ϵ determines the time step size $\delta t = \epsilon^2$.

- We declare global variables

```
  double *Xangles = new double[lcount]();
  double *Yangles = new double[lcount]();
  double *Zangles = new double[lcount]();
  double *eta = new double[pcount]();
  int *labels = new int[pcount]();
  double *JField = new double[pcount]();
```

We use pointer variables for these global variables and each independent `C++` class will communicate through their address.

- Then, we construct an initial polycrstal

```
  double maxZangle = 70.0* M_PI/180.0;
  InitializeCrystal::RandomCrystalConfiguration2D(n3,n2,n1,lcount,labels,
        maxZangle, Xangles,Yangles,Zangles);
```

These two lines of the code will generate a randomly distributed 2-D polycrystal system using a voronoi tessellation. The possible maximum Z-orientaion value is the provided value. Other crystal configurations can also be found in `InitCrystal.h` header file.

- Next, we construct a material class which calculates grain boundary energy $\mathcal{J}$

```
char materialType='C';
Material material(n3,n2,n1,materialType);

//Designate the location of Jump function data
if(materialType =='C')  {
  int dataNum=361;
  material.setCovarianceModel(dataNum, "inputs/jfun_cu_110.txt");
}
```

Here, we choose the material type 'C' which stands for the covariance model [7, 8]. Then, we need to provide the information of $\mathcal{J}$ by indicating the external data file. On the other hand, if a use want to simulate the original KWC model, it can be done by simply designating the `materialType` as 'S'.

- Construct numerical algorithm `C++` classes and link the pointers of global variables

```
//Construct PD Algorithm class
double PDerror=1e-6; // tolerance of Primal-dual algorithm
int PDmaxIters=10000; // allowable iteration number of the algorithm

PrimalDual<DIM> EtaSubProblem(n3, n2, n1, PDerror, PDmaxIters,
                        lcount,epsilon, Nthread);

//Construct Thresholding Algorithm class
double initThresCriteria = 0.9 ;
KWCThreshold<DIM> FastMarching(n3,n2,n1,lcount,initThresCriteria);

//Link pointers of Global variables to the algorithm classes

EtaSubProblem.setUpClass(eta, Xangles, Yangles, Zangles,
                        labels, JField,materialType);

FastMarching.setUpClass(eta, Xangles, Yangles, Zangles, labels, JField,'P');
```

Primal-dual class requires additional inputs for maximum allowable iteration counts and stopping criteria. Thresholding class requires the criteria for identifying the interior regions of grains. The code will identify the interior grains where $\mathcal{J}\left([\![\theta]\!]\right) <$ `initThreCrietria`. The argument of 'P' when we set up KWCThreshold class means that we will use periodic boundary condition.

- Now, we are finally ready to run the simulation

```
for (int i =0 ; i<20 ; i++)
{
  std::cout << "  " << i << "time-step begins...." << std::endl;

  material.calculateFieldJ('P');
  EtaSubProblem.run(epsilon);
  FastMarching.run(epsilon);
}
```

The argument of 'P' when we calculate FieldJ again means that periodic boundary condition is considered.

- The final step is to release memory space for algorithm classes and global variables

```
EtaSubProblem.freeMemory();
FastMarching.freeMemory();

delete[] eta; eta=NULL;
delete[] labels; labels=NULL;
delete[] Xangles; Xangles=NULL;
delete[] Yangles; Yangles=NULL;
delete[] Zangles; Zangles=NULL;
```

This is the end of `KWC_Polycrystal.cpp`.

3.2 Environment and Execution

In this section, we clarify system environment requirement and execution of the code. The developed code has been tested with and has been tested with a g++ compiler. The Primal-dual algorithm internally necessitates the use of Fast Fourier Transform (FFT) and inverse FFT, we borrow relevant libraries from `FFTW3` library. We suggest to use a 'makefile' tool to export these environment variables. For example, a `Make` file may look like

```
IDIR  += -I../../include/KWC_Simulation

CC= g++
CFLAGS += -Ofast
CFLAGS += -std=c++14
CFLAGS += -lm
CFLAGS += -lfftw3_threads
CFLAGS += -lfftw3
program:
        $(CC) KWC_polycrystal.cpp -o main $(CFLAGS) $(IDIR)
```

For data visualization options, we provide `vtu` format (can be read by `Paraveiw`) and `ffmpeg` library based animation. Yet, those two softwares are not mandatory.

Once the implementation code file `KWC_Polycrystal.cpp` is successfully compiled, it will create an executable, say that it is 'main'. The executable can be run with following arguments, which stands for size of computational domain N_x, N_y, N_z, and ϵ value.

```
./main 512 512 1 0.05
```

4 Data structure and visualization

In this section, we summarize how variable data is being stored in the code library and recommend some useful built-in `C++` functions that visualizes this data. Understanding the data structure and knowing how to quickly visualize it will be useful, when a user wants to modify or build something more on the current library for one's own purpose. As long as the input & output format is consistent, the functions & classes provided in the library can be replaced by a user-defined form.

4.1 Discrete field data stored in one dimensional array

Any discrete scalar field data (either 2D or 3D) in the code library is stored in one-dimensional array. The index of the array first increases in x direction, then y and z direction. Fo example, the discrete field, say $\eta(x, y, z)$, in a regular square domain, can be assessed as follow

```
/* Example: assess eta(x,y,z) and save it to Pvalue */

double Pvlalue;

for(int i=0; i<n3; i++) { // z loop
  for(int j=0; j<n2; j++) { //y loop
    for(int k=0; k<n1; k++) { //x loop

      double x = k/n1
      double y = j/n2
      double z = i/n3;

      Pvalue= eta[i*n2*n1+j*n1+k];
    }
   }
}
```

The only exception is the orientation field θ. In case of θ, we save `int-type` labels at each grid point and the components of orientation values corresponding to the label is stored separately. For example, the orientation value in Z-direction at point (x, y, z) can be assessed as follow

```
/* Example: assess theta_Z (x,y,z) and save it to Ztheta */

double Ztheta;

for(int i=0; i<n3; i++) { // z loop
  for(int j=0; j<n2; j++) { //y loop
    for(int k=0; k<n1; k++) { //x loop
```

```
        Ztheta= Zangles[labels[i*n1*n2+j*n1+k]];
      }
    }
  }
```

4.2 Output data in vtu format

`C++` functions defined in `DataOut.h` can be useful to visualize results in a format that many visualization software (e.g. `Paraview`) can read. Any scalar field data in 2D in the above data structure (one dimensional array) and orientation field data can be simply output to `.vtu` as follow

```
  /* Example: Output eta and output to the file  "myeta_0.vtu" */
Output2DvtuScalar(n1, n2, n3, eta, "eta value", "myfile_",0);

 /* Example: Output Z_theta and output to the file  "mytheta_0.vtu" */
Output2DvtuAngle(n1, n2, n3, Zangles, labels, "theta_Z", "myfile_",0);
```

5 Additional Features

This section is devoted to introduce additional features of the developed code that has not been included in the example code in Section 3

5.1 Design of the jump function

The example code `Design_J.cpp` in the `E4` folder uses a Newton's iteration to fit $\mathcal{J}(\llbracket\theta\rrbracket)$ to an external grain boundary energy data. The code takes an input file `STGB_cu_110.txt`, which is the covariance model prediction of FCC [110] Symmetric-tilt-grain-boundary energy [7, 8]. In fact, in the previous example code, we were actually using the $\mathcal{J}$-design results. The format of the input file should be as follow

```
 # tiltAngle(deg)          # W_data
 # ————————————             ————————————
                  0                 0.0
                0.5             0.10458
                  1            0.203463
                1.5            0.296963
                  2            0.385376
                2.5            0.468977
                  3            0.548028
                3.5            0.598243
... (continues)
```

To construct $\mathcal{J}$, we can call a `C++` class named `KWCDesignJ`. We only need to provide the number of data and the name of input&output file as follow.

```
int main(int argc, const char * argv[]) {
     int nData=361;
     KWCDesignJ optimizer(nData,"STGB_cu_110.txt","J_cu_110.txt");
     optimizer.run();
     return 0;
}
```

Then, the output file format, `J_cu_110.txt` can be later used for grain growth simulation.

```
    # tiltAngle       # Jtheta # Total Energy
# ------------  --------------  -------------
            0               0              0
          0.5       0.0432744        0.10458
            1        0.102464       0.203463
          1.5        0.171975       0.296963
            2        0.250429       0.385376
          2.5        0.337459       0.468977
            3        0.433323       0.548028
          3.5        0.502413       0.598243
... (continues)
```

5.2 (Optional) Setting FFMEPG for evolving grain animation

The developed code can generate grain evolution movie while simulation is running. This necessitates that `ffmpeg` be installed on the machine. In the following example, we will convert the Z-orientation value of grains into a number between [0,255]. Then, we will use black-and-white images to collect pictures of grains at each time step. To do this, we first prepare the conversion as follow

```
/* movie data */
unsigned char *pixels=new unsigned char[pcount];

unsigned char *colors=new unsigned char[lcount];
for(int l=0;l<lcount;l++){
    // Distribute colors to angles
      colors[l]=255*Zangles[l]/maxZangle;
}

char *string;
asprintf(&string,"ffmpeg -y -f rawvideo -vcodec rawvideo -pix_fmt gray -s
 %dx%d -r 30 -i - -f mp4 -q:v 5 -an -vcodec mpeg4 out.mp4", n1,n2);

//open an output pipe
FILE *pipeout = popen(string, "w");
```

Now, we can use "`PrepareFFMPEG2DPixel`" function defined it `DataOut.h` to convert orientation data to image

```
for (int i =0 ; i<20 ; i++)
{
  std::cout << "  " << i << "time-step begins...." << std::endl;
  PrepareFFMPEG2DPixels(n1,n2,0,pixels, labels, colors);
  fwrite(pixels, 1, pcount, pipeout);
  material.calculateFieldJ('P');
  EtaSubProblem.run(epsilon);
  FastMarching.run(epsilon);
}
```

References

[1] J. Kim, M. Jacobs, and N. C. Admal. A crystal-symmetry-invariant kwc grain boundary model and its simulation. *In Preparation*, 2020.

[2] R. Kobayashi, J. A. Warren, and W. C. Carter. Vector-valued phase field model for crystallization and grain boundary formation. *Physica D: Nonlinear Phenomena*, 119:415–423, 1998.

[3] A. E. Lobkovsky and J. A. Warren. Sharp interface limit of a phase-field model of crystal grains. *Physical Review E*, 63:051605, 2001.

[4] J. A. Warren, R. Kobayashi, A. E. Lobkovsky, and W. C. Carter. Extending phase field models of solidification to polycrystalline materials. *Acta Materialla*, 51:6035–6058, 2003.

[5] A. Chambolle and T. Pock. A first-order Primal-Dual algorithm for convex problems with applications to imaging. *Journal of Mathematical Imaging and Vision*, 40:120–145, 2011.

[6] M. Jacobs, F. Leger, W. Li, and S. Osher. Solving large-scale optimization problems with a convergence rate independent of grid size. *SIAM Journal on Numerical Analysis*, 57:1100–1123, 2019.

[7] B. Runnels, I. J. Beyerlein, S. Conti, and M. Ortiz. An analytical model of interfacial energy based on a lattice-matching interatomic energy. *Journal of Mechanics and Physics of Solids*, 89:174–193, 2016.

[8] B. Runnels, I. J. Beyerlein, S. Conti, and M. Ortiz. A relaxation method for the energy and morphology of grain boundaries and interfaces. *Journal of Mechanics and Physics of Solids*, 94:388–408, 2016.