Thresholding Method for simulating the alternate KWC model Code User Manual

Jaekwang Kim, Matt Jacobs, Nikhil Chandra Admal

September 30, 2020

1 Introduction

This is the user manual for illustrating details of the numerical scheme suggested in the white paper. The code solves simulate the grain growth described by KWC (Kobayashi–Warren–Carter) model [1, 2, 3]

The material free energy density of the KWC model $\mathcal{W}_{\text{kwc}}$ is defined by two field variable: the crystal-order phase parameter η and the orientation phase parameter θ. The free energy of the alternate KWC on domain Ω is

$$\mathcal{W}[\eta,\theta] = \int_\Omega \left[\frac{(1-\eta)^2}{2\epsilon} + \frac{\epsilon}{2}|\nabla\phi|^2 + sg(\eta)\mathcal{J}\big(\llbracket\theta\rrbracket\big)\delta(x-x_0) \right] dV. \tag{1}$$

where δ is the dirac-delta function, $\llbracket\theta\rrbracket$ the jump of θ, x_0 the position of grain boundary, and $\mathcal{J}\big(\llbracket\theta\rrbracket\big) : \mathbb{R}^+ \to \mathbb{R}^+$ is the non-diffused part of grain boundary energy function. [1] We evolve a given polycrystal by iteratively 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 separate numerical method for problem: we use the Primal-dual algorithm [4, 5], for the first η-sub problem and thresholding method for the second θ-sub problem. The resulting solution is motion by curvature with anisotropic grain boundary energy [2].

2 Procedures

The code is developed as `C++` header-template library, which a user can freely refer. `C++` Function & Class are categorized into different header files based on their task. These Functions & Classes can be replaced by a user-defined form, as long as the input & output format is consistent. 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 configuration such as bicrystal, tricrystal, and randomly distributed polycrystal can be easily constructed using a function defined in `InitCrystal.h`. Next,

[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.

[2] Here, the "anisotropic" does not include GB plane inclination dependence.

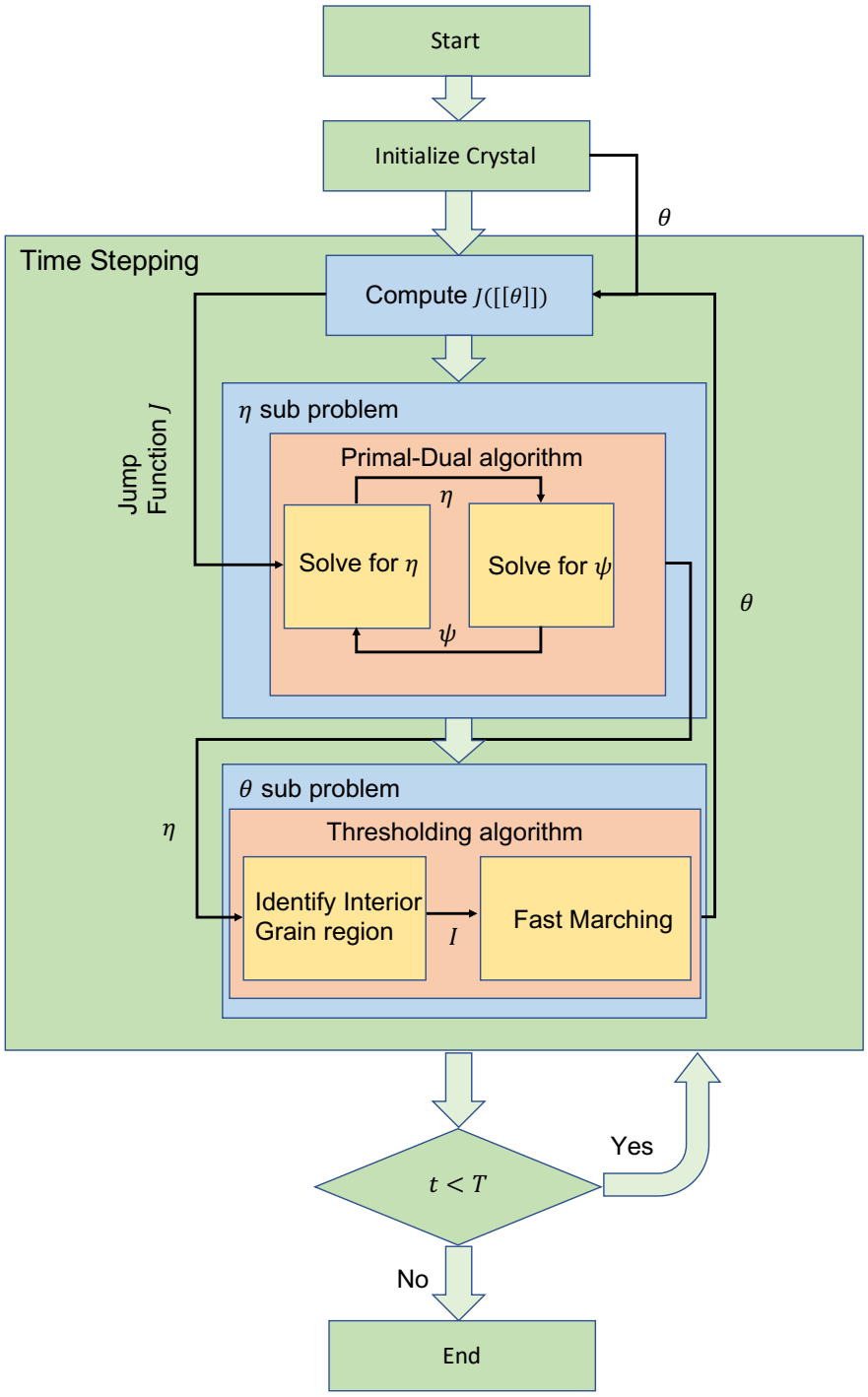

Figure 1: Algorithm Flow chart

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

a user is required to define the jump function $\mathcal{J}(\llbracket\theta\rrbracket)$. The `Material` class defined in `Material.h` is useful for defining $\mathcal{J}$. In the following, the `PrimalDual` class and `KWCThreshold` class should be constructed to solve η-sub problem (2) and θ-sub problem (3). After these classes are constructed, 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 against 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 steps in Fig. 1 can be implemented using the template library.

3.1 Implementation file

We will read the implementation code file block by block. A bullet point is used to emphasize 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=2;
  const unsigned int lcount=50;

  int pcount = n1*n2*n3;
  double dt= epsilon * epsilon; //initial choice of dt

  int Nthread = 1; //Number total thread to be used for FFTW
```

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. Also, note that 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);
```

This line will generate a randomly distributed 2-D polycrystal with the given maximum Z-orientaion value, using a voronoi tessellation. Other crystal configurations can also be found in `InitCrystal.h` header file. A user can also define new crystal configuration independently for one's own needs.

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

```
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 material, of which grain boundary energy $\mathcal{J}$ is described by the covariance model [6, 7]. Then, we 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 the link 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);

//Link pointers of Global variables to the Primal-Dual algorithm class

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

double initThresCriteria = 2*epsilon ;
KWCThreshold<DIM> FastMarching(n3,n2,n1,lcount,initThresCriteria);
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 $\mathcal{J}\left(\llbracket\theta\rrbracket\right) <$ `initThreCrietria`.

- 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 final step is to release memory space

```
  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 discuss required system environment and how to execute the code. The developed code has been tested with and has been tested with a g++ compiler. Because the Primal-dual algorithm 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, an example format of which is as follow.

```
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 use `ffmpeg` libraries and `Paraview`. 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

In this section, we summarize how data is being stored in the current code and recommend useful built-in C++ functions that visualizes this data. Understanding the data structure and knowing how to quickly visualize will be useful, when a user wants to modify or build something more on the current library for one's own purpose.

4.1 Discrete field data stored in one dimensional array

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

```
/* 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 each component of orientation values corresponding to the label is saved separately. For example, the θ 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 previous example code.

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 [6, 7]. The format of the input file is 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 `KWCDesignJ C++ class`. 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 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. 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 need to prepare it 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");
```

Then, we 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] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.