

MID-MESO Tutorial

Microstructure Intelligent Design - Mesoscale

Designer & Developer: Qi Huang

June 9th, 2022

Science center for phase diagram, phase transition, material intelligent design and
manufacture, Central South University
相图、相变及材料智能设计与制备科学中心，中南大学

目录

1.	MID-MESO 的介绍	4
2.	多元多相模型框架	5
2.1.	总能	5
2.2.	势的平衡	5
2.3.	与组元耦合后的多相的变化趋势	5
2.4.	Cahn-Hilliard 方程拓展	6
2.5.	Allen-Cahn 方程拓展	6
3.	常规的模拟流程	8
3.1.	MID-MESO 在 windows 下的安装	9
3.2.	MID-MESO 在 linux 下的安装	13
3.3.	开始一个简单的模拟	13
3.3.1.	基本设置	14
3.3.2.	main 函数编写	18
4.	初始化一个微观结构	23
4.1.	填充椭圆	24
4.2.	填充椭球	25
4.3.	填充多边形	27
4.4.	填充多面体	29
4.5.	构建二维下的 voronoi 结构	31
4.6.	构建三维下的 voronoi 结构	32
4.7.	从 BMP 图片上读取结构	33

5.	界面研究	36
5.1.	一维界面研究	37
5.2.	二维界面研究	39
5.3.	多相结	42
5.4.	给一个驱动	42
5.5.	界面上相组分研究	44
6.	其他场的简单调用	49
6.1.	温度场	49
6.2.	弹性场	50
6.3.	电场	53
6.4.	磁场	54
6.5.	流场	55
6.6.	泊松方程求解器	59
6.7.	Allen-Cahn 方程求解器	61
6.8.	Cahn-Hilliard 方程求解器	63
7.	搭建一个自己的数据库/函数库	66

1. MID-MESO 的介绍

MID-MESO 是由中南大学相图、相变及材料智能设计与制备实验室主任、教育部长江学者、973 项目首席科学家——杜勇教授提出以耦合实际热力学、扩散、弹性等数据库为目标的，耦合多物理场的多元多相的介观微结构模拟设计程序库。

MID-MESO 由黄奇博士设计，运用 C++ 高级程序语言搭建了基本数据结构，模拟网格可自动拓展以适应多晶多元的模拟。程序模拟所基于的多元多相模型框架中的关键物理参数及方程的接口都被写在程序设置类（information 类）中，它们与其他模拟参数一同皆可被使用者定义、替换，方便了使用者针对特定模拟体系的二次开发，避免了专注模型设计、材料设计的研究者对于程序底层的关注。该程序库可在 windows、linux 双系统上编译运行，使用了 OpenMP 并行加速计算，针对多相场、相浓度场方程中在界面上的数值问题进行了优化处理。

MID-MESO 受《MID-MESO 合作互助协议》保护。程序库的使用者需签订《MID-MESO 合作互助协议》与杜勇教授及程序开发者合作方有权使用 MID-MESO 获利，而非盈利性学习使用、教育等不予限制。

MID-MESO 受《MID-MESO 多方权益保护协议》保护。程序库的开发者需参加相图、相变及材料智能设计与制备科学中心相场方向培训课程，完成最终考核并签订新版的《MID-MESO 多方权益保护协议》方有权参与程序开发，以获取更多权益。

杜勇教授: yong-du@csu.edu.cn;

黄奇博士: hq5088028@csu.edu.cn; 1494968861@qq.com;

2. 多元多相模型框架

2.1. 总能

模拟域内的总能被定义为各能量密度的积,

$$F = \int_{\Omega} \{ f^{int}(\mathbf{r}, t) + f^{thermo}(\mathbf{r}, t) + f^{mech}(\mathbf{r}, t) + f^{elec}(\mathbf{r}, t) + f^{mag}(\mathbf{r}, t) \\ + f^{other}(\mathbf{r}, t) \} d\Omega$$

引入总浓度与相浓度的关系作为限制条件, 可得

$$\Gamma(\mathbf{r}, t) = f^{total}(\mathbf{r}, t) + \sum_{i=1}^{n-1} \lambda_i \left(x_i(\mathbf{r}, t) - \sum_{\alpha} \phi^{\alpha}(\mathbf{r}, t) x_i^{\alpha}(\mathbf{r}, t) \right)$$

2.2. 势的平衡

$x_i^{\alpha}(\mathbf{r}, t)$ 演化的终点, 即为势的平衡

$$\frac{\delta}{\delta x_i^{\alpha}(\mathbf{r}, t)} \Gamma(\mathbf{r}, t) = 0 = \frac{\delta}{\delta x_i^{\alpha}(\mathbf{r}, t)} f^{total}(\mathbf{r}, t) - \lambda_i \phi^{\alpha}(\mathbf{r}, t)$$

平衡时存在

$$\tilde{\mu}_i^{\alpha}(V_m)^{-1} = \lambda_i = \frac{1}{\phi^{\alpha}(\mathbf{r}, t)} \frac{\delta}{\delta x_i^{\alpha}(\mathbf{r}, t)} f^{total}(\mathbf{r}, t) = \frac{1}{\phi^{\beta}(\mathbf{r}, t)} \frac{\delta}{\delta x_i^{\beta}(\mathbf{r}, t)} f^{total}(\mathbf{r}, t)$$

即非平衡状态下, 各相内的组元的势

$$\tilde{\mu}_i^{\alpha}(V_m)^{-1} = \frac{1}{\phi^{\alpha}(\mathbf{r}, t)} \frac{\delta}{\delta x_i^{\alpha}(\mathbf{r}, t)} f^{total}(\mathbf{r}, t)$$

2.3. 与组元耦合后的多相的变化趋势

考虑多相场与组分场的耦合, 对 $\Gamma(\mathbf{r}, t)$ 取变化

$$\frac{\delta}{\delta \phi^{\alpha}(\mathbf{r}, t)} \Gamma(\mathbf{r}, t) = \frac{\delta}{\delta \phi^{\alpha}(\mathbf{r}, t)} f^{total}(\mathbf{r}, t) - \sum_{i=1}^{n-1} \lambda_i x_i^{\alpha}(\mathbf{r}, t)$$

即得到在耦合组分场情况下的 $\phi^{\alpha}(\mathbf{r}, t)$ 变化的趋势, 而在 $x_i^{\alpha}(\mathbf{r}, t)$ 不存在的情

况下该式退化为一般方程

$$\frac{\delta}{\delta \phi^{\alpha}(\mathbf{r}, t)} \Gamma(\mathbf{r}, t) = \frac{\delta}{\delta \phi^{\alpha}(\mathbf{r}, t)} f^{total}(\mathbf{r}, t)$$

2.4. 扩散方程的拓展

一般扩散方程为

$$(V_m)^{-1} \frac{\partial x_i^\alpha}{\partial t} = -\nabla \cdot \mathbf{J}_i^\alpha + S_i^\alpha = \nabla \cdot \left(\sum_{j=1}^n M_{ij}^\alpha \nabla \tilde{\mu}_j^\alpha \right) + S_i^\alpha$$

拓展为多相形式

$$(V_m)^{-1} \phi^\alpha \frac{\partial x_i^\alpha}{\partial t} = \phi^\alpha \nabla \cdot \mathbf{J}_i^\alpha + \phi^\alpha S_i^\alpha = \nabla \cdot (\phi^\alpha \mathbf{J}_i^\alpha) - \nabla \phi^\alpha \cdot \mathbf{J}_i^\alpha + \phi^\alpha S_i^\alpha$$

假设相区内物质的量不随相变改变，考虑相转变的影响

$$\frac{\partial [\phi^\alpha(\mathbf{r}, t) x_i^\alpha(\mathbf{r}, t)]}{\partial t} = 0 = \phi^\alpha(\mathbf{r}, t) \frac{\partial x_i^\alpha(\mathbf{r}, t)}{\partial t} + x_i^\alpha(\mathbf{r}, t) \frac{\partial \phi^\alpha(\mathbf{r}, t)}{\partial t}$$

得，相浓度场方程一般形式

$$(V_m)^{-1} \frac{\partial x_i^\alpha}{\partial t} = \frac{1}{\phi^\alpha} \left[\nabla \cdot (\phi^\alpha \mathbf{J}_i^\alpha) - \nabla \phi^\alpha \cdot \mathbf{J}_i^\alpha + \phi^\alpha S_i^\alpha - (V_m)^{-1} x_i^\alpha \frac{\partial \phi^\alpha}{\partial t} \right]$$

界面扩散项 $\nabla \phi^\alpha \cdot \mathbf{J}_i^\alpha$ 可拓展为

$$-\nabla \phi^\alpha \cdot \mathbf{J}_i^\alpha \equiv - \sum_{\beta \neq \alpha} \nabla \phi^{\alpha\beta} \cdot \mathbf{J}_i^\alpha = - \sum_{\beta \neq \alpha} (\nabla \phi^{\alpha\beta} \cdot \tilde{\mathbf{J}}_i^{\alpha\beta}) \equiv \sum_{\beta} |\nabla \phi^{\alpha\beta}| (\mathbf{n}^{\alpha\beta} \cdot \tilde{\mathbf{J}}_i^{\alpha\beta})$$

其中 $\nabla \phi^{\alpha\beta} = \phi^\beta \nabla \phi^\alpha - \phi^\alpha \nabla \phi^\beta$ 为两相间得梯度， $\mathbf{n}^{\alpha\beta} = -\nabla \phi^{\alpha\beta} / |\nabla \phi^{\alpha\beta}|$ 向 α 相

内的单位向量， $\mathbf{n}^{\alpha\beta} \cdot \tilde{\mathbf{J}}_i^{\alpha\beta}$ 为朝 α 相的两相间扩散量、反应量。

2.5. Allen-Cahn 方程拓展

一般 AC 方程为

$$\frac{\partial \phi^\alpha(\mathbf{r}, t)}{\partial t} = -L^\alpha(\mathbf{r}, t) \frac{\delta}{\delta \phi^\alpha(\mathbf{r}, t)} \Gamma(\mathbf{r}, t)$$

加入相分数归一限制条件

$$\frac{\partial \phi^\alpha(\mathbf{r}, t)}{\partial t} = -L^\alpha(\mathbf{r}, t) \frac{\delta}{\delta \phi^\alpha(\mathbf{r}, t)} \left[\Gamma(\mathbf{r}, t) + \lambda_\phi \left(\sum_\gamma \phi^\gamma(\mathbf{r}, t) - 1 \right) \right]$$

已知归一后相分数的变化量归零

$$\sum_\alpha \frac{\partial \phi^\alpha(\mathbf{r}, t)}{\partial t} = 0$$

导出反对称形式相分数演化方程

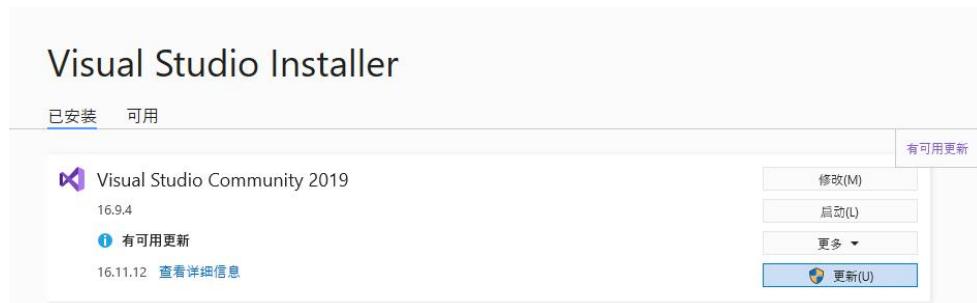
$$\begin{aligned}
\frac{\partial \phi^\alpha(\mathbf{r}, t)}{\partial t} &= -L^\alpha(\mathbf{r}, t) \frac{\delta}{\delta \phi^\alpha(\mathbf{r}, t)} \Gamma(\mathbf{r}, t) + L^\alpha(\mathbf{r}, t) \frac{1}{N} \sum_\beta \frac{\delta}{\delta \phi^\beta(\mathbf{r}, t)} \Gamma(\mathbf{r}, t) \\
&= \frac{1}{N} \sum_\beta L^{\alpha\beta}(\mathbf{r}, t) \left(\frac{\delta}{\delta \phi^\beta(\mathbf{r}, t)} - \frac{\delta}{\delta \phi^\alpha(\mathbf{r}, t)} \right) \Gamma(\mathbf{r}, t) \\
&= \sum_\beta \left[\tilde{L}^{\alpha\beta}(\mathbf{r}, t) \frac{1}{\eta} \left(\frac{\delta}{\delta \phi^\beta(\mathbf{r}, t)} - \frac{\delta}{\delta \phi^\alpha(\mathbf{r}, t)} \right) \tilde{\Gamma}(\mathbf{r}, t) + \dot{S}^{\alpha\beta}(\mathbf{r}, t) \right]
\end{aligned}$$

3. 常规的模拟流程

MID-MESO 软件包：



在 windos 下的模拟需要 visual studio2019 及以上的版本，同时需要将 vs 软件更新到最新



在 linux 系统下的模拟需要 g++ 11.4.0 及以上的编译器版本

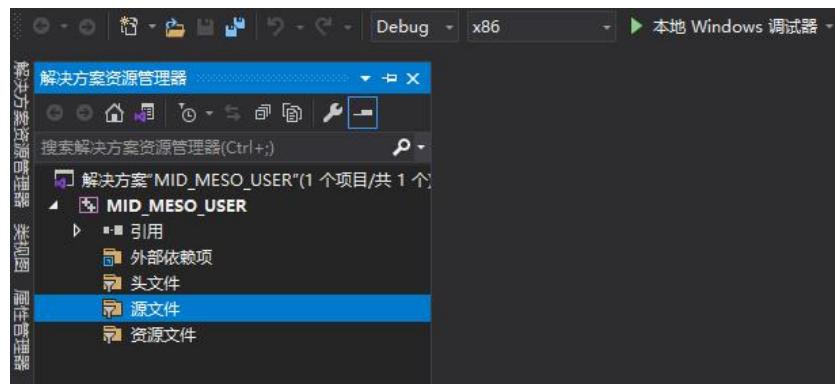
```
hq5088028@DESKTOP-JD54JCB:~$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/11/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none:amdgcn-amdhsa
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 11.4.0-1ubuntu1
c/gcc-11/README.Bugs' --enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c
on-only --program-suffix=-11 --program-prefix=x86_64-linux-gnu- --enable-share
sr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --e
ale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-li
--disable-vtable-verify --enable-plugin --enable-default-pie --with-system-zl
ith-target-system=zlib=auto --enable-objc-gc=auto --enable-multiarch --disable
-with-abi=m64 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=
e=/build/gcc-11-XeT9LY/gcc-11-11.4.0/debian/tmp-nvptx/usr,amdgcn-amdhsa=/buil
d/usr --without-cuda-driver --enable-checking=release --build=x86_64-linux-gnu
linux-gnu --with-build-config=bootstrap-lto-lean --enable-link-serialization=2
Thread model: posix
Supported LTO compression algorithms: zlib zstd
gcc version 11.4.0 (Ubuntu 11.4.0-1ubuntu1~22.04)
```

而程序的模拟结果包含的.vts 文件则需要 paraview 制图软件制图

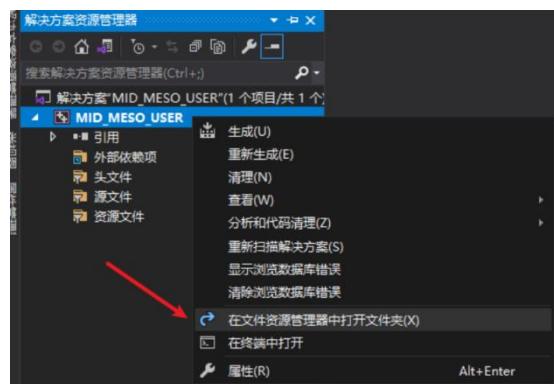


3.1. MID-MESO 在 windows 下的安装

首先新建一个空项目



右键项目，打开项目文件夹

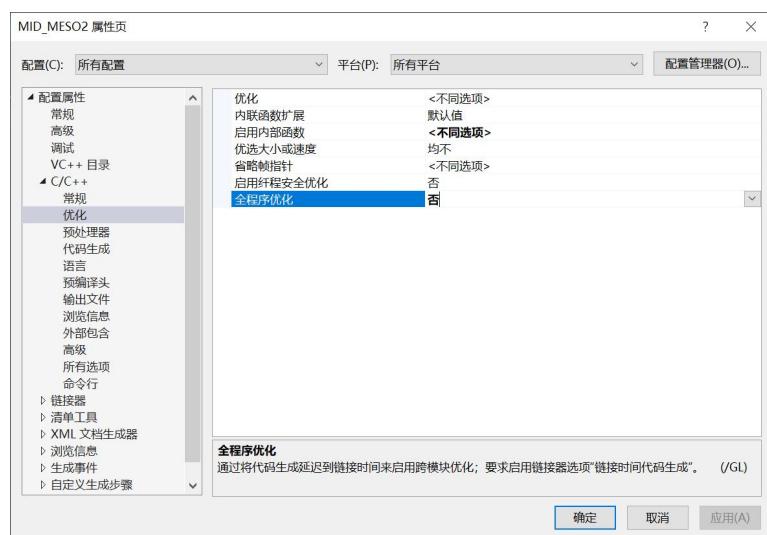


将 MID-MESO 软件包复制解压到项目文件夹内

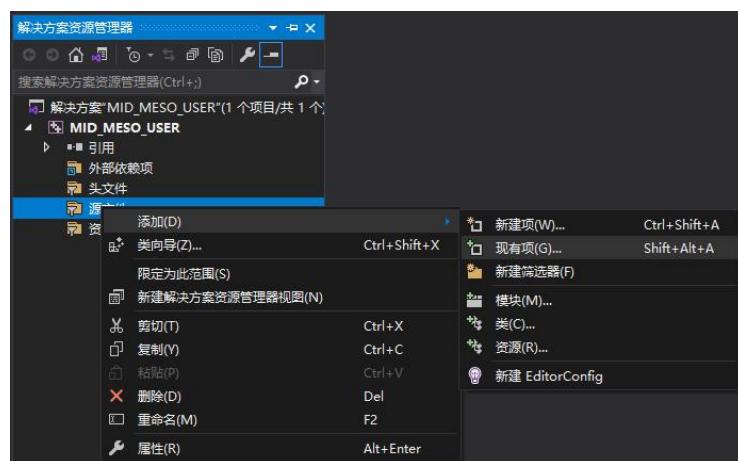
work (D:) > program > MID_MESO_USER > MID_MESO_USER

名称	修改日期	类型	大小
benchmark	2022/4/15 14:37	文件夹	
examples	2022/4/15 14:37	文件夹	
include	2022/4/15 14:37	文件夹	
lib	2022/4/15 14:37	文件夹	
compile_trunk.sh	2022/4/14 16:16	SH 文件	1 KB
libfftw3-3.dll	2016/7/30 23:38	应用程序扩展	2,650 KB
libfftw3f-3.dll	2016/7/30 23:42	应用程序扩展	2,708 KB
libfftw3l-3.dll	2016/7/30 23:44	应用程序扩展	1,219 KB
MID_MESO_USER.vcxproj	2022/4/14 21:30	VC++ Project	8 KB
MID_MESO_USER.vcxproj.filters	2022/4/14 21:30	VC++ Project Fil...	1 KB
MID_MESO_USER.vcxproj.user	2022/4/14 17:13	Per-User Project...	1 KB

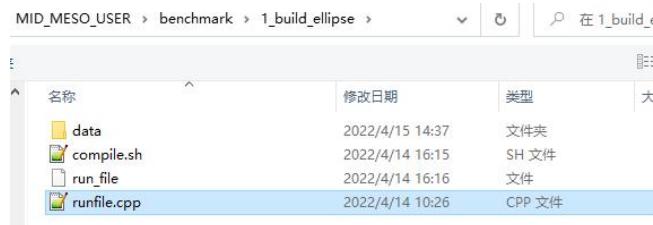
在软件界面右键项目 - 属性 - 优化 - 全程序优化，选择“否”（一般情况不会有
问题，可选“是”）



MID-MESO 配置完成，接着可以通过在 VS 界面上添加现有项导入现有的
cpp 运行文件



以 benchmark 中的 1_build_ellipse 案例为例，选择该文件夹内的 runfile.cpp



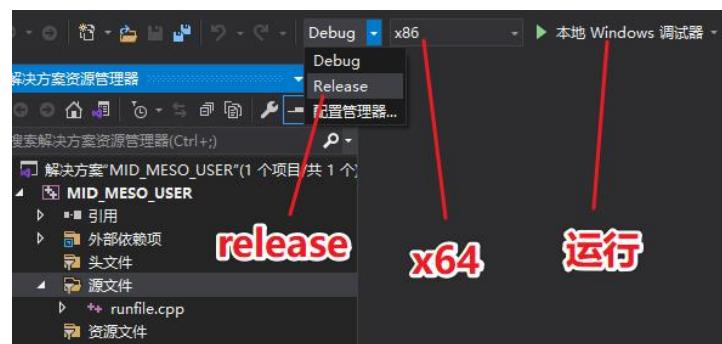
双击界面中的 runfile.cpp，可以对 runfile.cpp 进行编辑

```

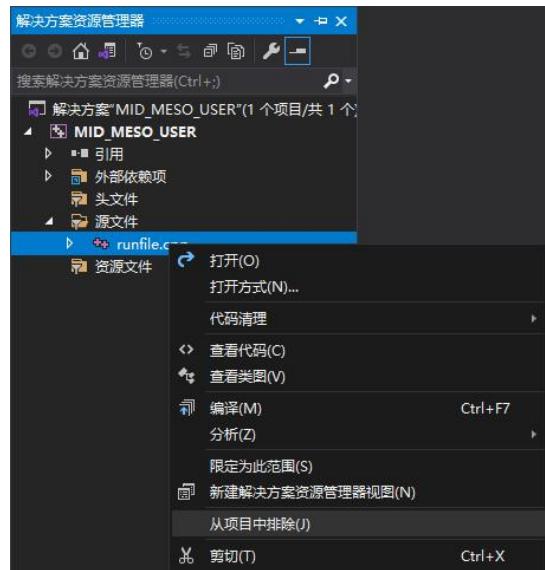
runfile.cpp + X
MID_MESO_USER
1 #define _CRT_SECURE_NO_WARNINGS
2 #include "../../include/MPF.h"
3 #using namespace pf;
4 #using namespace std;
5 #ifndef _WIN32
6 #define CPP_FILE_PATH string("benchmark\\1_build_ellipse\\")
7 #else
8 #define CPP_FILE_PATH string("")
9 #endif;
10 pf::Information settings();
11 int main(int argc, char* argv[])
12 {
13     // init simulation by settings
14     MPF simulation;
15     simulation.init_Modules(settings());
16     simulation.init_SimulationMesh();
17     pf::Set_Disperse& set = simulation.information.settings.disperse_settings;
18
19     simulation.output_before_loop();
20     for (int istep = set.begin_step; istep <= set.end_step; istep++) {
21         simulation.information.dynamicCollection.init_each_timeStep(istep, set.dt);
22
23         simulation.nucleations.nucleation(istep);
24     }
}

```

然后调整运行模式为 release 模式，同时选择编译器为 x64 位，最后点击右侧的运行，即可得结果。



当想要切换其他案例的 runfile.cpp 时，可右键（从项目中）排除当前加载的 runfile，然后重复上面导入其他 runfile 操作



另外，如果电脑没有 64 位，需要选择 x86 编译器的，也请将 lib\x86 中的 3 个 dll 文件 (fftw3 的动态链接库，如下图)

名称	修改日期	类型	大小
debug	2022/4/14 16:50	文件夹	
release	2022/4/14 16:50	文件夹	
fftw3.h	2016/7/30 23:10	H 文件	19 KB
libfftw3-3.dll	2016/7/30 23:05	应用程序扩展	2,257 KB
libfftw3-3.lib	2021/4/28 4:29	Object File Library	242 KB
libfftw3f-3.dll	2016/7/30 23:08	应用程序扩展	2,336 KB
libfftw3f-3.lib	2021/4/28 4:29	Object File Library	247 KB
libfftw3l-3.dll	2016/7/30 23:10	应用程序扩展	1,105 KB
libfftw3l-3.lib	2021/4/28 4:29	Object File Library	149 KB

复制黏贴，替换项目文件中的：

名称	修改日期	类型	大小
benchmark	2022/4/12 14:20	文件夹	
Debug	2022/4/19 23:51	文件夹	
examples	2022/4/12 10:13	文件夹	
include	2022/4/23 21:40	文件夹	
lib	2022/4/14 16:49	文件夹	
Release	2022/4/19 23:51	文件夹	
src	2022/4/23 19:57	文件夹	
x64	2022/4/14 16:55	文件夹	
compile_trunk.sh	2022/4/23 21:23	SH 文件	2 KB
libfftw3-3.dll	2016/7/30 23:38	应用程序扩展	2,650 KB
libfftw3f-3.dll	2016/7/30 23:42	应用程序扩展	2,708 KB
libfftw3l-3.dll	2016/7/30 23:44	应用程序扩展	1,219 KB
MID_MESO.vcxproj	2022/4/23 17:20	VC++ Project	10 KB
MID_MESO.vcxproj.filters	2022/4/23 17:20	VC++ Project Fil...	6 KB
MID_MESO.vcxproj.user	2022/4/12 21:54	Per-User Project...	1 KB

然后就可以正常使用 visual studio 软件中的 x86 编译器编译并运行该程序。

3.2. MID-MESO 在 linux 下的安装

将 MID-MESO 软件包解压到文件夹中，在命令行中定位到该文件夹（注：仅解压出的文件是必须的，其他都不是必须的，这里以 windows10 的子系统 ubuntu 为例）

```
huangqi@DESKTOP-2U2T7R9:/mnt/d/program/MID_MESO_USER$ cd MID_MESO_USER/
huangqi@DESKTOP-2U2T7R9:/mnt/d/program/MID_MESO_USER/MID_MESO_USER$ ll
总用量 6596
drwxrwxrwx 1 huangqi huangqi 512 4月 15 21:45 /
drwxrwxrwx 1 huangqi huangqi 512 4月 14 21:30 /
drwxrwxrwx 1 huangqi huangqi 512 4月 15 14:37 benchmark/
-rwxrwxrwx 1 huangqi huangqi 969 4月 14 16:16 compile_trunk.sh*
drwxrwxrwx 1 huangqi huangqi 512 4月 15 14:37 examples/
drwxrwxrwx 1 huangqi huangqi 512 4月 15 14:37 include/
drwxrwxrwx 1 huangqi huangqi 512 4月 15 14:37 lib/
-rwxrwxrwx 1 huangqi huangqi 2712765 7月 30 2016 libfftw3-3.dll*
-rwxrwxrwx 1 huangqi huangqi 2772692 7月 30 2016 libfftw3f-3.dll*
-rwxrwxrwx 1 huangqi huangqi 1247967 7月 30 2016 libfftw3l-3.dll*
-rwxrwxrwx 1 huangqi huangqi 7223 4月 15 21:45 MID_MESO_USER.vcxproj*
-rwxrwxrwx 1 huangqi huangqi 995 4月 15 21:45 MID_MESO_USER.vcxproj.filters*
-rwxrwxrwx 1 huangqi huangqi 168 4月 14 17:13 MID_MESO_USER.vcxproj.user*
drwxrwxrwx 1 huangqi huangqi 512 4月 15 21:45 val/
huangqi@DESKTOP-2U2T7R9:/mnt/d/program/MID_MESO_USER/MID_MESO_USER$ -
```

随意访问一个 runfile.cpp 文件，以 benchmark\\1_build_ellipse 为例

```
huangqi@DESKTOP-2U2T7R9:/mnt/d/program/MID_MESO_USER/MID_MESO_USER/benchmark/1_build_ellipse$ ll
总用量 4
drwxrwxrwx 1 huangqi huangqi 512 4月 15 21:57 /
drwxrwxrwx 1 huangqi huangqi 512 4月 15 14:37 /
-rwxrwxrwx 1 huangqi huangqi 276 4月 14 16:15 compile.sh*
drwxrwxrwx 1 huangqi huangqi 512 4月 15 21:45 data/
-rwxrwxrwx 1 huangqi huangqi 2152 4月 14 10:26 runfile.cpp*
huangqi@DESKTOP-2U2T7R9:/mnt/d/program/MID_MESO_USER/MID_MESO_USER/benchmark/1_build_ellipse$ -
```

运行 compile.sh 实现对 runfile.cpp 的编译

```
huangqi@DESKTOP-2U2T7R9:/mnt/d/program/MID_MESO_USER/MID_MESO_USER/benchmark/1_build_ellipse$ ./compile.sh
> The program is compiling... Please wait...
> The runfile has been builded
huangqi@DESKTOP-2U2T7R9:/mnt/d/program/MID_MESO_USER/MID_MESO_USER/benchmark/1_build_ellipse$ ll
总用量 2744
drwxrwxrwx 1 huangqi huangqi 512 4月 15 21:58 /
drwxrwxrwx 1 huangqi huangqi 512 4月 15 14:37 /
-rwxrwxrwx 1 huangqi huangqi 276 4月 14 16:15 compile.sh*
drwxrwxrwx 1 huangqi huangqi 512 4月 15 21:45 data/
-rwxrwxrwx 1 huangqi huangqi 2302504 4月 15 21:58 run_file*
-rwxrwxrwx 1 huangqi huangqi 2152 4月 14 10:26 runfile.cpp*
huangqi@DESKTOP-2U2T7R9:/mnt/d/program/MID_MESO_USER/MID_MESO_USER/benchmark/1_build_ellipse$ -
```

生成的 run_file 文件为编译好的运行文件，可以直接运行或者提交到服务器的任务列表运行。

3.3. 开始一个简单的模拟

对于一个基本的模拟，仅需要编写 runfile.cpp 文件（暂不考虑数据库文件

的编写)。runfile.cpp 文件基本可以分为两部分内容：information 类的基本设置、main 函数编写。在 setting() 函数中实例化的 information 类的变量会在函数 main 函数中被读取，使用者设置的参数也因此能够被 MID-MESO 所调用。

Information 类参数众多，详细可自己打开 include\baseTools\InfoTools.h 和 include\modules\Information.h 查看内部的参数、默认值及 information 类中的一些辅助函数。

3.3.1. 基本设置

为了使代码更清晰，我们在 pf::Information settings(); 函数中初始化一个 information 类的变量，并将它返回以待进一步使用。因此首先我们在 settings() 函数中实例化一个 information 类变量 inf

pf::Information inf;

然后我们就可以修改它内部各属性的值

<1> 模拟域的参数

inf.settings.disperse_settings.Nx = 1;

inf.settings.disperse_settings.Ny = 1;

inf.settings.disperse_settings.Nz = 1;

这里的 Nx、Ny、Nz 分别表示 x、y、z 三个方向上网格点的数量。一般而言，一维的模拟需要将 Ny、Nz 定义为 1，二维模拟需要将 Nz 定义为 1，而三维不做限制，这暗合一些程序功能函数的逻辑判断。

inf.settings.disperse_settings.dx = 1.0;

这里的 dx 为模拟域内单个网格在 x、y、z 方向上的大小，MID-MESO 中采

用正方体网格，也因此模拟域的在 x、y、z 方向上的大小分别为 $N_x \cdot dx$ 、 $N_y \cdot dx$ 、 $N_z \cdot dx$ 。

<2> 模拟的时间参数

```
inf.settings.disperse_settings.begin_step = 0;
```

```
inf.settings.disperse_settings.end_step = 0;
```

```
inf.settings.disperse_settings.dt = 1.0;
```

这里的 begin_step 与 end_step 分别为模拟的开始时间步和模拟的结束时间步，而 dt 为时间步的步长。

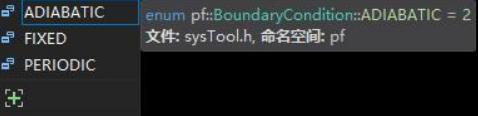
<3> 网格的边界条件

```
inf.settings.disperse_settings.x_bc = BoundaryCondition::PERIODIC;
```

```
inf.settings.disperse_settings.y_bc = BoundaryCondition::PERIODIC;
```

```
inf.settings.disperse_settings.z_bc = BoundaryCondition::PERIODIC;
```

这里三个参数分别控制 x、y、z 三个方向上的边界情况。其中 BoundaryCondition 这个 enum 类型含有三个可选的 flag (ADIABATIC、FIXED、PERIODIC)



```
pf::Information inf;
// 数值离散时空
inf.settings.disperse_settings.begin_step = 0;
inf.settings.disperse_settings.end_step = 0;
inf.settings.disperse_settings.Nx = 50;
inf.settings.disperse_settings.Ny = 50;
inf.settings.disperse_settings.Nz = 1;
inf.settings.disperse_settings.dt = 1e-2;
inf.settings.disperse_settings.x_bc = BoundaryCondition::  
// 其他功能
// 文件输入输出
inf.settings.file_settings.isPhiOutput = true;
inf.settings.file_settings.working_folder_path = CPP FILE
```

分别代表了绝热边界、固定边界、周期边界。其中绝热边界、周期边界直接就可以定义，无需其他设置补充；而固定边界条件，需进一步定义边界上各个点固定数值

```
inf.settings.details settings.MPF_FixedBoundaryCondition.push(XXXX);
```

<4> .dat 文件的生成和程序的再启动

MID_MESO 软件通过将模拟域中各相的相分数、相浓度进行输出为二进制 data 文件，然后通过读取 data 文件复现相分数与相浓度的方式来保存一个模拟，和在已有模拟的基础上启动一个新的模拟。

```
inf.settings.file_settings.is_write_datafile = true;
```

```
inf.settings.file_settings.dataFile_output_step = 1;
```

```
inf.settings.file_settings.data_file_name = "DATA";
```

一般而言程序会在退出时输出一个结束的 data 文件，is_write_datafile 设置则启动在程序运行过程中输出 data 文件，dataFile_output_step 设置为程序每隔多少时间步输出一个 data 文件，data_file_name 设置为输出的 data 文件公有的文件名。

```
inf.settings.file_settings.is_init_byDatafile = false;
```

```
inf.settings.file_settings.is_Datafile_init_by_path = false;
```

```
inf.settings.file_settings.restart_datafile_path = "";
```

以上三行代码控制 data 文件的读取，其中 is_init_byDatafile 设置是否以 data 文件启动一个模拟，is_Datafile_init_by_path 为是否通过写 data 文件地址的方式来启动一个模拟，这里在 windows 下可以选 false，从而在程序初始化过程中会弹出选框，可以可视化地在 win 系统提供的文件资源管理器选择目标 data 文件，而在 linux 下只能选择 true，然后进一步在 restart_datafile_path 中定义 data 文件的路径。

<5> 输出的设置

```
inf.settings.file_settings.working_folder_path = CPP_FILE_PATH + "data";
```

```
inf.settings.file_settings.file_output_step = 1;
```

```
inf.settings.file_settings.screen_output_step = 1;
```

```
inf.settings.file_settings.log_file_name = "log";
```

以上 4 个参数控制程序的基本输出， working_folder_path 为工作文件夹，是所有输出文件的存放地， file_output_step 控制每隔多少时间步输出 vts 文件， screen_output_step 控制每隔多少时间步统计一次相、浓度、能量等信息，并在控制台上输出日志， log_file_name 为日志文件的文件名。

<6> .vts 文件的输出内容的控制

```
inf.settings.file_settings.isPhiOutput = false;
```

```
inf.settings.file_settings.isPhaseIndexesOutput = false;
```

```
inf.settings.file_settings.isPhaseFlagsOutput = false;
```

```
inf.settings.file_settings.isPhaseGradientOutput = false;
```

```
inf.settings.file_settings.isPhaseConOutput = false;
```

以上参数控制一些基本数据的输出， Phi 即为各相的相分数， PhaseIndexes 为各个晶粒按编码输出， PhaseFlags 即输出各个晶粒的晶界， PhaseGradient 为输出各晶粒界面上的法向方向， PhaseCon 为输出各晶粒的相浓度。

```
inf.settings.file_settings.isInterfaceEnergyOutput = false;
```

```
inf.settings.file_settings.isDrivingForceOutput = false;
```

```
inf.settings.file_settings.isChemicalPotentialDivisionOutput = false;
```

```
inf.settings.file_settings.isEnergyDensityOutput = false;
```

以上参数控制一些能量数据的输出， InterfaceEnergy 对应

$$\left(\frac{\delta}{\delta \phi^\beta(\mathbf{r}, t)} - \frac{\delta}{\delta \phi^\alpha(\mathbf{r}, t)} \right) \Gamma^{int}$$

DrivingForce 对应

$$\left(\frac{\delta}{\delta \phi^\beta(\mathbf{r}, t)} - \frac{\delta}{\delta \phi^\alpha(\mathbf{r}, t)} \right) \Gamma^{bulk}$$

ChemicalPotentialDivision 对应

$$\frac{\delta}{\delta x_i^\alpha(\mathbf{r}, t)} \Gamma^{thermo}(\mathbf{r}, t) \text{ and } \frac{\delta}{\delta x_i^\alpha(\mathbf{r}, t)} \Gamma^{elas}(\mathbf{r}, t) \text{ and } \frac{\delta}{\delta x_i^\alpha(\mathbf{r}, t)} \Gamma^{elec}(\mathbf{r}, t) \dots$$

EnergyDensity 对应

$$\frac{\delta}{\delta \phi^\alpha(\mathbf{r}, t)} f^{thermo}(\mathbf{r}, t) \text{ and } \frac{\delta}{\delta \phi^\alpha(\mathbf{r}, t)} f^{elas}(\mathbf{r}, t) \text{ and } \frac{\delta}{\delta \phi^\alpha(\mathbf{r}, t)} f^{elec}(\mathbf{r}, t) \dots$$

inf.settings.file settings.isTemperatureOutput = false;

inf.settings.file settings.isElectricFieldOutput = false;

inf.settings.file settings.isMagneticFieldOutput = false;

inf.settings.file settings.isMechanicalFieldOutput = false;

inf.settings.file settings.isFluidFieldOutput = false;

以上参数控制外场数据的输出，分别对应温度场、电场、磁场、力场、流场。

3.3.2. main 函数编写

一般情况下，启动一个相场模拟都需要初始化模拟网格和界面能模块等，因此这里给出一个通用的 main 函数框架：

```
MPF simulation; //实例化一个多相场模拟
simulation.init_Modules(settings()); //初始化各个模块
simulation.init_SimulationMesh(); //初始化模拟网格及内存申请
pf::Set_Disperse& set = simulation.information.settings.disperse_settings;
```

```

//调用设置信息，用于制造循环

simulation.output_before_loop(); //模拟前预输出

for (int istep = set.begin_step; istep <= set.end_step; istep++) {

    //程序主循环

    simulation.information.dynamicCollection.init_each_timeStep(istep,
set.dt); //动态收集数据的初始化，用于辅助各个函数计算

    simulation.init_mesh_data(istep); //初始化模拟中调用的中间量

    .... //程序主体

    simulation.output_in_loop(istep); //循环中动态输出

    simulation.data_file_write_in_loop(istep); //循环中生成 data 文件

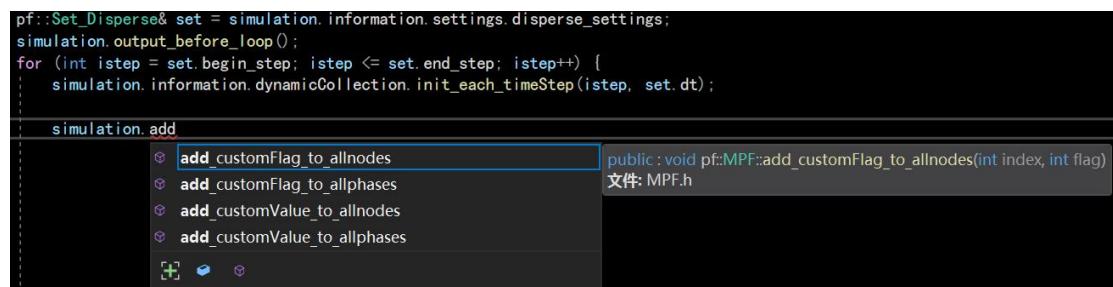
}

simulation.output_after_loop(); //程序结束的总结输出

simulation.exit_MPf(); //结束时 data 文件输出及所有内存的释放

```

以上 main 函数的框架基本为固定框架，在大多数情况下都可用。而程序主体部分可以填入 MPF 类或其附带模块中的功能函数（此过程推荐在 window 系统，使用 vs 软件辅助完成）：



如上图，vs 除了方便的 debug 功能纠错外，可以辅助函数及变量的编写，而且将鼠标放于函数名上有关于该函数的声明介绍，方便使用者查找该函数和输入适合的参数。下面介绍主循环中可用的功能函数：

<1>核心物理参数计算函数，计算扩散系数、热扩散系数、各相能量密度、相组分扩散势等。

simulation.prepare_physical_parameters_in_mesh(istep);

<2>演化方程部分，相浓度演化方程、相分数演化方程、温度演化方程

simulation.evolve_phase_concentration_evolution_equation(istep);

simulation.evolve_phase_evolution_equation(istep, false);

simulation.evolve_temperature_evolution_equation(istep);

<3>赋值功能函数，相分数赋值、相分数+相浓度赋值(组合)、温度赋值

simulation.phaseFraction_assignment(istep, false);

simulation.phaseFraction_assignment_and_prepare_cFlag(istep, false);

simulation.phaseConcentration_assignment_with_cFlag(istep);

simulation.temperature_assignment(istep);

<4>形核函数，调用形核模块中的形核函数，在每个时间步都对核盒进行侦测，生成符合条件的核（含生成新相的内存申请）

simulation.nucleations.nucleation(istep);

<5>弛豫界面函数，自动排除驱动力，仅在界面能作用下弛豫界面，需要注意的是相浓度不会演化所以需结合<6>内的相浓度 fill 函数使用，弛豫界面后再次对各个晶粒中的相浓度初始化

simulation.relaxation_interface(2000, 100, true);

<6>填充函数，强制填充某个晶粒的浓度或温度，需注意该函数不能直接调用，需以 if 判断时间步后调用，否则会导致相浓度和温度场无法演化；同时无法保证该晶粒内的组分归一，需人为控制所填充的各个相组分之和归一。

```
simulation.fill_phase_con_with_value(0, 0, 0, 0.0);
```

```
simulation.fill_phase_temperature_with_value(0, 0, 0.0);
```

<7>自动调整时间步长函数，以确保数值演化稳定，且兼顾长时间迭代效率；其控制参数为设置内的 `phi_incre_limit` 及 `con_incre_limit`，该函数会自动限制相分数和相浓度的增量在该限制以下。

```
simulation.automatically_adjust_dt(istep, 500, 500, 100.0, 1e-3, true);
```

<8>物质守恒辅助函数，在对物质守恒要求高的模拟过程中，可以调用该函数，它会读取程序启动之初读取的组分含量配比（中间也可修改该配比标准），每隔一段时间步，对模拟域内的组分 `comps` 进行放缩，弥补离散计算或数值问题带来的物质损失

```
simulation.conserved_the_aim_substances(vector<int> comp, istep, 500,  
false);
```

<9>采用局部均匀化的方法，对驱动力带来的相分数增量进行优化，每个时间步处理 `times` 次，均匀化范围为设置中的 `bulk_incre_average_range`，该过程可以缓解因界面内外驱动力差异过大所带来的界面不稳定

```
simulation.optimizationAlgorithm.optimize_bulk_increment_on_interface(iste  
p, times);
```

<10>采用界面法向方向局部均匀化的方法对相浓度场赋值，该函数可替换上面给出的相浓度赋值函数，设置中的 `con_average_range` 参数控制沿界面均匀处理的长度，该函数极大回避相浓度的数值问题，且能解决因浓度演化不稳定带来的界面问题，其缺点为界面上的浓度场效果不佳，完全失去晶界扩散功能

```
simulation.optimizationAlgorithm.optimize_phaseCon_assignment with_cFla  
g(istep);
```

<11>与上面函数的处理思路相同，但不具有赋值功能，且仅缓和因晶界演化过快所带来的相浓度问题，没有优化相浓度场中的扩散项和反应项，理论上保留了晶界扩散的功能

```
simulation.optimizationAlgorithm.optimize_phaseCon_phase_transition_term  
_on_interface(istep);
```

<13>在模拟域上加入自定义场变量，需在函数主循环外调用

```
simulation.add  
    ⌂ add_customFlag_to_allnodes  
    ⌂ add_customFlag_to_allphases  
    ⌂ add_customValue_to_allnodes  
    ⌂ add_customValue_to_allphases
```

<14>自定义场变量的输出函数，输出单独的 vts 文件，需配合 if 判断时间步使用

```
simulation.write_custom  
    ⌂ write_customFlag_in_node  
    ⌂ write_customFlag_in_phase  
    ⌂ write_customValue_in_node  
    ⌂ write_customValue_in_phase  
    ⌂ write_customVec3_in_node  
    ⌂ write_customVec3_in_phase
```

4. 初始化一个微观结构

本节主要讲解初始化微观结构的辅助工具：`information::nucleationBox`，该工具包含 `nucleus_box`、`geometricRegion_box`、`condition_check_phase_box`、`pointSet_box` 四个盒子，分别用于定义形核点、**几何体形核**、条件形核、**点集形核**四种类型的数据结构，用于构建微观结构。

能够在程序初始化阶段起作用的盒子分别是 `geometricRegion_box` 和 `pointSet_box`。使用者可直接实例化 `GeometricRegion` 和 `PointSet` 类型的变量，并将其 `push` 到对应 `box` 中。然后可通过在 `main` 函数的主循环中调用 `Nucleation::nucleation(istep, is_normalize);` 函数来实时检测 4 个形核盒，并执行形核步骤或者动态的条件判断生成新的核。

形核的顺序是有规律的，各个盒子中的 `generate_step` 是第一层判断，其负责在第 `generate_step` 时间步上形核与主循环 `istep` 对应，`generate_step` 小的核会率先生成。在 `generation_step` 相同的情况下，`geometricRegion_box` 中的核会优先于 `pointSet_box` 中的核生成。而分别在 `geometricRegion_box` 内或者 `pointSet_box` 内的核则按其 `push` 的顺序判断，先 `push` 的核会优先生成。优先生成的核，如与后面生成的核在空间上有重叠则会被后续生成的核覆盖重叠部分，基于此原理可以构建更加复杂的微观组织结构，如同绘图作画一般。

以上 `GeometricRegion` 和 `PointSet` 类型的变量的通用属性有：负责标记晶粒的 `phaseIndex`，进一步负责区分各晶粒的 `phaseProperty`，控制在第几时间步形核的 `generate_step`，以及期望修改的形核区域内的物质成分 `x` 和温度 `temperature`。

后续程序中存在的可使用的类会以蓝色标名，而代码部分，则以下划线标注。

4.1. 填充椭圆

椭圆属于空间几何体，故此采用 `GeometricRegion` 类实例化一个变量：

`pf::GeometricRegion geo;`

然后可以调用该类中的 `init` 函数来初始化这个变量的部分属性 `init`（几何类型, `generate_step`, `phaseIndex`, `phaseProperty`, `temperature`）：

`geo.init(Geometry::Geo_Ellipsoid, 0, 1, 0, 0.0);`

接着需要定义一个椭圆的基本属性，圆心位置 (x, y, z) ：

`geo.ellipSolid.set_core(25, 25, 0);`

已经椭圆在各方向上的半径 (x, y, z) ：

`geo.ellipSolid.set_radius(20, 10, 0);`

除了上面定义的温度，也可定义组分，以及自定义的一些变量，都会自动在椭圆范围内初始化（这一部分用法在 X.X.X 可见，以下代码亦可省略）：

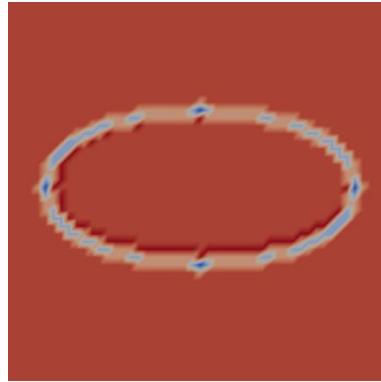
`geo.x;`

`geo.customFlags;`

`geo.customValues;`

`geo.customVec3s;`

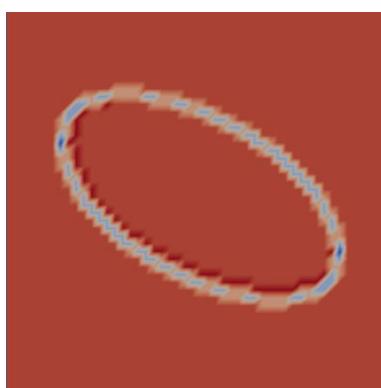
根据以上的设置可以构建的结构如下：



通过函数 `set_rotation_radian_and_rotation_gauge` 可以控制几何体的旋转，
`double _radian[3]` 为三次旋转的旋转弧度，`RotationGauge _rotationGauge` 为每
次旋转所绕的坐标轴（定义为旋转遵守的标准），需注意这里采用的坐标系为几
何体上的相对坐标，而不是模拟域的绝对坐标 (`double _radian[3]`,
`RotationGauge _rotationGauge`)：

```
double radian[] = {0.0, AngleToRadians(30.0), 0.0};  
geo.ellipSolid.set_rotation_radian_and_rotation_gauge(radian,  
RotationGauge::RG_XZX);
```

以上含义为绕椭圆圆心的 X 轴 0 度、Z 轴 30 度、X 轴 0 度旋转，结果如下：



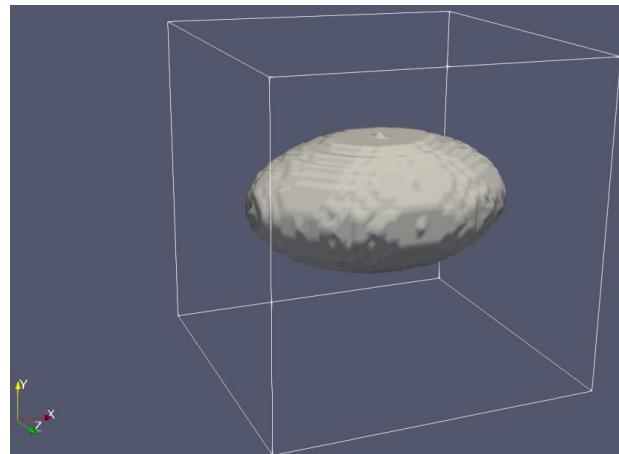
4.2. 填充椭球

三维椭球的初始化与椭圆类同，而在定义几何体时需要定义三维几何体：

```
geo.ellipSolid.set_core(25, 25, 25);
```

```
geo.ellipSolid.set_radius(20, 10, 15);
```

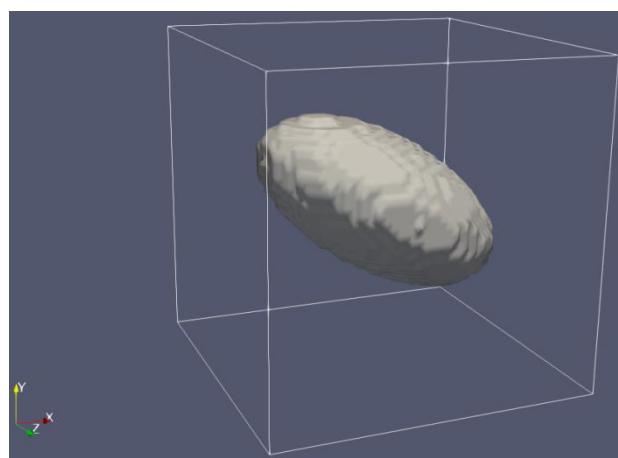
可以构建的椭球如下：



同理，可通过函数 `set_rotation_radian_and_rotation_gauge` 可以控制几何体的旋转（旋转中心为椭圆圆心）：

```
double radian[] = { AngleToRadians(30.0), 0.0, 0.0};  
geo.ellipSolid.set_rotation_radian_and_rotation_gauge(radian,  
RotationGauge::RG_ZXZ);
```

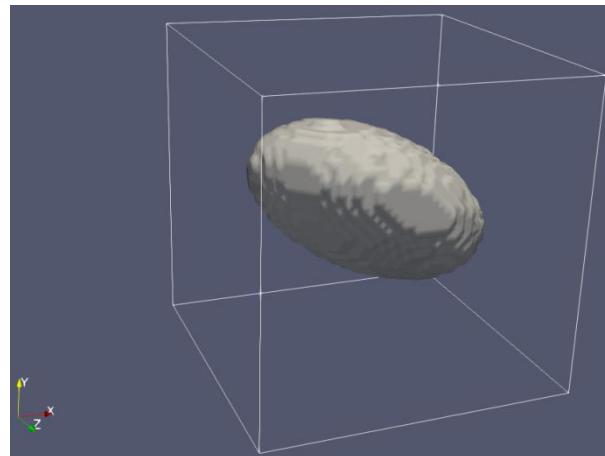
以上代码含义为，绕 Z 轴旋转 30 度：



```
double radian[] = { AngleToRadians(30.0), AngleToRadians(30.0), 0.0};  
geo.ellipSolid.set_rotation_radian_and_rotation_gauge(radian,
```

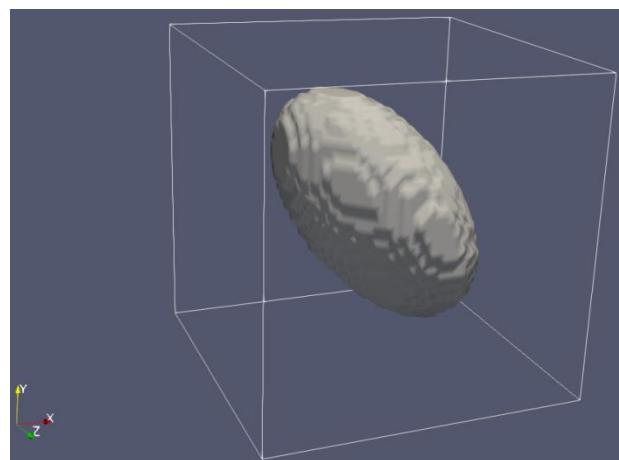
```
RotationGauge::RG_ZXZ);
```

以上代码含义为，绕 Z 轴旋转 30 度，再绕 X 轴旋转 30 度：



```
double radian[] = { AngleToRadians(30.0), AngleToRadians(30.0),  
AngleToRadians(30.0) };  
  
geo.ellipSolid.set_rotation_radian_and_rotation_gauge(radian,  
  
RotationGauge::RG_ZXZ);
```

以上代码含义为，绕 Z 轴旋转 30 度，绕 X 轴旋转 30 度，再绕 Z 轴旋转 30 度：

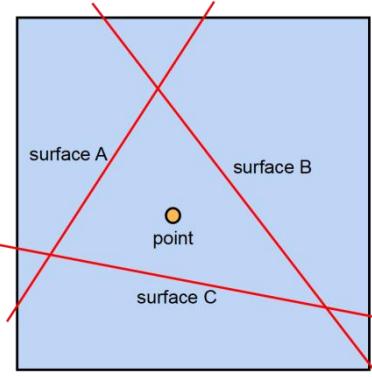


4.3. 填充多边形

多边形的初始化与椭圆、椭球类同，在几何类型处声明为多面体：

```
geo.init(Geometry::Geo_Polyhedron, 0, 1, 0, 0.0);
```

在定义该多边形的几何结构时需要几个切面及一个空间上的点，可以根据下图理解：



如图，我们通过三个面 ABC 切割了模拟域，然后我们通过定义一个域内的点来申明想要的区域（一般选择中间的多边形，但也可选择周围的区域），需注意的是为避免一些逻辑问题，该构图法无周期性。但是相同的思路可直接拓展到 1 维和 3 维，具有很强的普适性。

可以通过三个点定义一个面，在二维情况下，前两个点落在平面上，第三个点可简单定义为前两个点其中一个在 z 轴上的腾空：

```
geo.polyhedron.add_surf(Point(25, 45, 0), Point(10, 25, 0), Point(10, 25, 1));
```

我们可以把平面上的点选择为几何体的顶点：

```
geo.polyhedron.add_surf(Point(10, 25, 0), Point(25, 5, 0), Point(25, 5, 1));
```

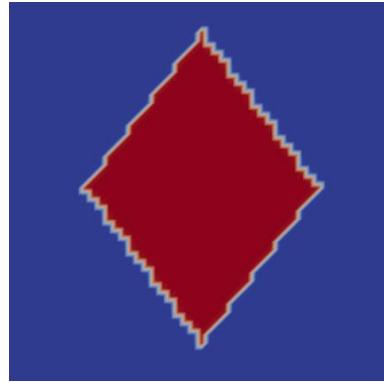
```
geo.polyhedron.add_surf(Point(25, 5, 0), Point(40, 25, 0), Point(40, 25, 1));
```

```
geo.polyhedron.add_surf(Point(40, 25, 0), Point(25, 45, 0), Point(25, 45, 1));
```

由于我们想要一个常规菱形，因此选择这四个切面中间的点：

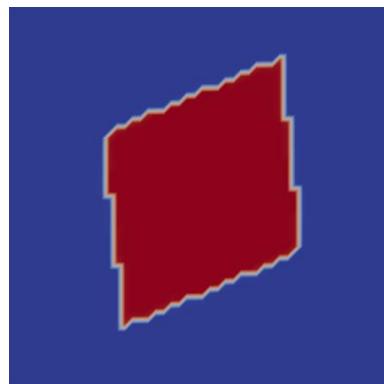
```
geo.polyhedron.set_a_point_inside_polyhedron(25, 25, 0);
```

构建的多面体如下：



当然，我们可以通过函数 `set_rotation_radian_and_rotation_gauge` 控制几何体的旋转（旋转中心为上面定义的多边形内的点）：

```
double radian[] = {0.0, AngleToRadians(30.0), 0.0};  
  
geo.ellipSolid.set_rotation_radian_and_rotation_gauge(radian,  
RotationGauge::RG_XZX);
```



4.4. 填充多面体

多面体的构建与多边形的构建逻辑完全自治，仅需在设置切面时注意多面体的封闭性，同时设定的点不要设在多面体之外。我们可以通过多面体的各个顶点设置各个切面，下例为八面体菱形的各个切面：

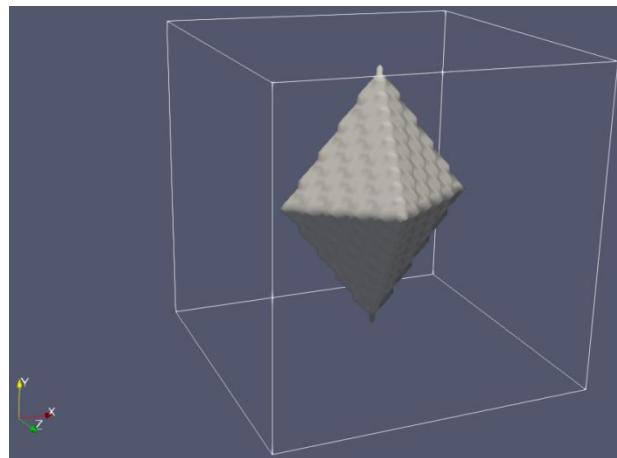
```
geo.polyhedron.add_surf(Point(25,45,25), Point(10,25,25), Point(25,25,15));  
  
geo.polyhedron.add_surf(Point(25,45,25), Point(10,25,25), Point(25,25,35));
```

```
geo.polyhedron.add_surf(Point(25,5,25), Point(10,25,25), Point(25,25,15));  
geo.polyhedron.add_surf(Point(25,5,25), Point(10,25,25), Point(25,25,35));  
geo.polyhedron.add_surf(Point(25,45,25), Point(25,25,15), Point(40,25,25));  
geo.polyhedron.add_surf(Point(25,45,25), Point(25,25,35), Point(40,25,25));  
geo.polyhedron.add_surf(Point(25,5,25), Point(25,25,15), Point(40,25,25));  
geo.polyhedron.add_surf(Point(25,5,25), Point(25,25,35), Point(40,25,25));
```

同时我们设置菱形中心的点：

```
geo.polyhedron.set_a_point_inside_polyhedron(25, 25, 25);
```

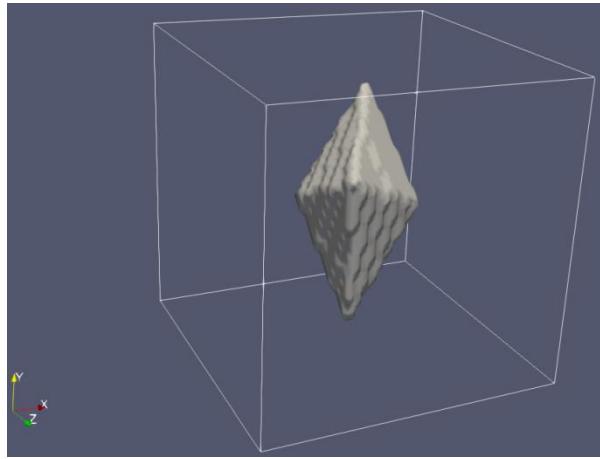
可以构建如下菱形：



同理，我们可以对该菱形执行旋转操作（旋转中心同为设定的多面体内的点）：

```
double radian[] = {0.0, AngleToRadians(-60.0), AngleToRadians(30.0)};  
geo.ellipSolid.set_rotation_radian_and_rotation_gauge(radian,  
RotationGauge::RG_XYZ);
```

旋转后的结果为：



4.5. 构建二维下的 voronoi 结构

Voronoi 结构的构建是通过 `information` 类的辅助函数：

```
Information::generate_voronoi_structure(Vector3    box_position,    Vector3  
box_size, int grain_number, int generate_step, vector<int> phases_properties,  
vector<double> phases_weight, XNode x, double temperature) 来控制形核系统  
中的 geometricRegion_box, 在给定参数的情况下自动生成有关联的多边形, 然  
后由 Nucleation::nucleation(istep, is_normalize); 函数检测生成。
```

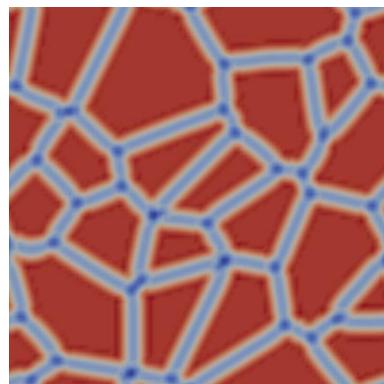
其中参数 `box_position` 为需要生成 voronoi 结构的空间起始点位置, 一般选
择为 0 点; 参数 `box_size` 为需要生成 voronoi 结构的空间的 x、y、z 三个方向的
长度; 参数 `grain_number` 为晶粒数量, 编号由 0~(number -1); `generate_step`、
`x`、`temperature` 与上面几何结构类同; `phases_properties` 为想要赋予晶粒的
property 属性, 用于一定程度上区分各个晶粒, 分配方式属于随机分配, 使用
时将想分配的 property 属性 push 到 `vector<int>` 的实例化变量中, 并在函数此
处作为输入参数输入; `phases_weight` 为与 `phases_properties` 对应的各个
property 属性分配时的权重, 权重高的分配量会更多。

简单定义一个不含组分且不区分各晶粒的 voronoi 结构:

```
vector<int> phase_property;  
phase_property.push_back(0);  
vector<double> phase_weight;  
phase_weight.push_back(1);  
pf::XNode x;  
inf.generate_voronoi_structure(Vector3(0, 0, 0), Vector3(100, 100, 0), 20, 0,  
phase_property, phase_weight, x, 0.0);
```

以上函数含义为生成 20 个晶粒，晶粒属性皆为 0，不含组分，温度为 0，生成的空间为以 0 点为基的 x、y 方向长 100 的正方形 voronoi 周期性结构。注意，如果模拟域小于该设定空间，最终获得的 voronoi 结构将不完整；而如果模拟域略大于该设定空间模拟域会被自动填充，如果过大则会异常。

得到的结果如下：



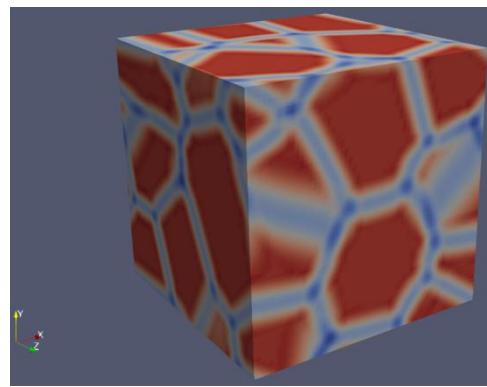
4.6. 构建三维下的 voronoi 结构

Voronoi 三维结构与二维结构类同，定义：

```
vector<int> phase_property;  
phase_property.push_back(0);
```

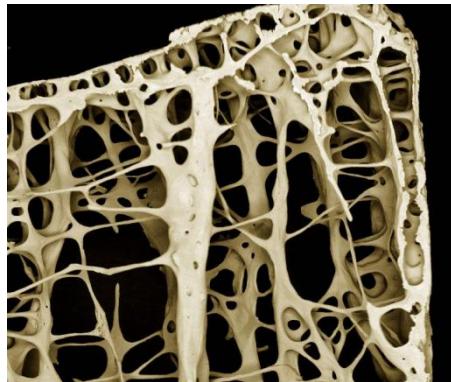
```
vector<double> phase_weight;  
phase_weight.push_back(1);  
pf::XNode x;  
inf.generate_voronoi_structure(Vector3(0, 0, 0), Vector3(60, 60, 60), 12, 0,  
phase_property, phase_weight, x, 0.0);
```

即需要将生成 voronoi 的空间区域定义为三维。结果如下



4.7. 从 BMP 图片上读取结构

为了适应更加普遍的情况，我们有构建更加复杂的二维组织结构的需求。我们通过在草稿上绘制结构图或者通过扫描电镜、相机等拍摄了自己想要模拟的图片，并且期望能够在相场模拟之初建立。当然再复杂的结构都有通过上面几何形核拼装成的可能，但如果可以直接受打印机打印一张图片，没有人愿意去一笔一画作画。因此对于图片形式的微观组织结构，程序提供了对 BMP24 图片的读取、转化、输出类([BMP24reader](#))，使用者需预加工自己的图片（至 bmp24 位）以适应程序的需求，如对于下例复杂的骨骼结构图片

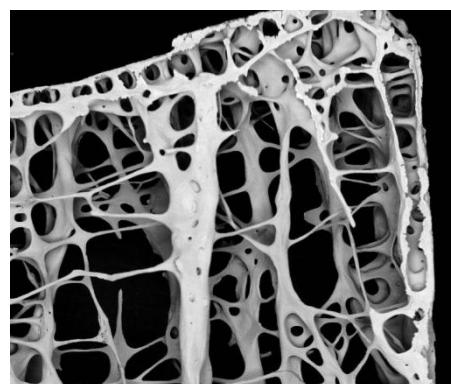


可直接调用 `information` 类的辅助函数：

```
Information::generate_structure_from_BMP_pic(int      phaselIndex,      int  
phase_property, int generate_step, XNode x, double threshold[2], double  
temperature, string fileName) 来控制形核系统中的 pointSet_box，在给定参数  
的情况下自动生成点的集合，然后由 Nucleation::nucleation(istep, is_normalize);  
函数检测生成初始结构。
```

其中参数 `phaselIndex`、`property`、`generate_step`、`temperature`、`x` 的设置与几何体参数一致，分别为晶粒的编号、晶粒的性质、生成的步数、该结构的温度和组分。`fileName` 为读取图片的地址，`threshold[2]` 为取灰度值的一个区间来生成该晶粒。

该函数会自动将上面的图片转化为灰度图片：



白色为 1，黑色为 0。通过设置合理的 `threshold[2]` 可以读取自己想要的区

域。比如通过代码：

```
pf::XNode x;  
double threshold1[] = { 0.6, 1.0 };  
inf.generate_structure_from_BMP_pic(2, 0, 0, x, threshold1, 0.0,  
CPP_FILE_PATH + "bone.bmp");  
double threshold2[] = { 0.1, 0.6 };  
inf.generate_structure_from_BMP_pic(1, 0, 0, x, threshold2, 0.0,  
CPP_FILE_PATH + "bone.bmp");
```

分别读取 [0.6, 1.0] 区间的白色骨骼部分为晶粒 2，读取 [0.1, 0.6] 区间的带阴影的骨骼部分为晶粒 1，可以得到如下组织结构（输出各晶粒的 Index）：



其中红色部分为 index=2 的晶粒，白色部分为 index=1 的晶粒，蓝色部分为 index=0 的背景。

需要注意的是，该过程用到了 windows 系统的函数，因此无法在 linux 系统下使用该函数，使用者可通过 windows 产生.dat 文件，然后在 linux 下初始化该结构。

5. 界面研究

界面能是相场的总能中最基础的部分，它主导形成了各晶粒间“diffuse interface”的形成，对于界面能及界面演化的理解，也是理解整个相场法的根基。在排除驱动力、插值函数及其他耦合的演化方程的影响后，自由能函数表现为如下形式

$$F(\phi, \nabla\phi) = \int_V f^{int} dV = \int_V f^{grad}(\phi, \nabla\phi) + f^{pot}(\phi) dV$$

针对多晶粒的相场模拟，目前已存在有非常多的界面能的模型。比较经典的梯度模型有 Steinbach 1996 和 Steinbach 1999，势能模型有 Nestler 改进的势垒和势井

$$\begin{aligned} f_{S.96}^{grad}(\phi, \nabla\phi) &= \eta \sum_{\alpha} \sum_{\beta>\alpha} \xi^{\alpha\beta} |\phi^{\alpha} \nabla\phi^{\beta} - \phi^{\beta} \nabla\phi^{\alpha}|^2 \\ f_{S.99}^{grad}(\phi, \nabla\phi) &= -\eta \sum_{\alpha} \sum_{\beta>\alpha} \xi^{\alpha\beta} \nabla\phi^{\alpha} \nabla\phi^{\beta} \\ f_{N.well}^{pot}(\phi) &= \frac{9}{\eta} \sum_{\alpha} \sum_{\beta>\alpha} \xi^{\alpha\beta} (\phi^{\alpha} \phi^{\beta})^2 + \frac{1}{\eta} \sum_{\alpha} \sum_{\beta>\alpha} \sum_{\gamma>\beta} \xi^{\alpha\beta\gamma} (\phi^{\alpha} \phi^{\beta} \phi^{\gamma})^2 \\ f_{N.obstacle}^{pot}(\phi) &= \frac{16}{\eta\pi^2} \sum_{\alpha} \sum_{\beta>\alpha} \xi^{\alpha\beta} \phi^{\alpha} \phi^{\beta} + \frac{1}{\eta} \sum_{\alpha} \sum_{\beta>\alpha} \sum_{\gamma>\beta} \xi^{\alpha\beta\gamma} \phi^{\alpha} \phi^{\beta} \phi^{\gamma} \end{aligned}$$

并且，针对界面上的一些问题，Steinbach 在 2009 年进一步给出了反对称形式的界面能演化方程

$$\left(\frac{\delta}{\delta\phi^{\beta}} - \frac{\delta}{\delta\phi^{\alpha}} \right) \Gamma^{int} = \xi^{\alpha\beta} \left(\eta (\phi^{\beta} \nabla^2 \phi^{\alpha} - \phi^{\alpha} \nabla^2 \phi^{\beta}) + \frac{\pi^2}{2\eta} (\phi^{\alpha} - \phi^{\beta}) \right)$$

以上多晶形式的界面能演化方程都被写到界面能模块之中，案例写于 benchmark 的 2_interface_study1D 和 2_interface_study2D 中，使用者可以在 runfile.cpp 的设置函数中，修改 information 类的参数以选择期望使用的梯度能及势能（以 information 类实例化参数 inf）

```
inf.settings.details.settings.int_grad = Int_Gradient::Steinbach_1999;
```

```
inf.settings.details settings.int_pot = Int_Potential::Nestler_Obstacle;
```

使用者可以通过以下参数调控界面宽度

```
inf.settings.disperse_settings.int_width = 10.0;
```

使用者可以通过复写界面能函数 Xi_{ab} 和 Xi_{abc} 来替代程序中的默认函数,

以实现对界面能的控制

```
inf.materialSystem.functions.Xi_ab = Xi_ab;
```

```
inf.materialSystem.functions.Xi_abc = Xi_abc;
```

使用者也可以通过设置梯度能及势能的变分为自定义格式，并复写界面能对相的变分来实现界面能模型的个人设计，详见 example 的 dendrite 案例

```
inf.settings.details settings.int_grad = Int_Gradient::Int_GCustom;
```

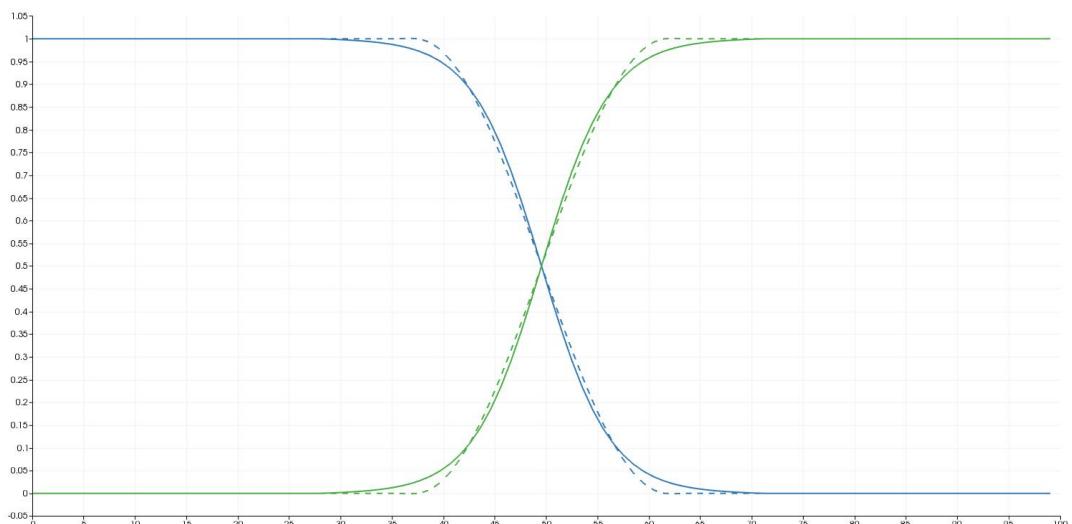
```
inf.settings.details settings.int_pot = Int_Potential::Int_PCustom;
```

```
inf.materialSystem.functions.dfint_dphi = dfint_dphi;
```

5.1. 一维界面研究

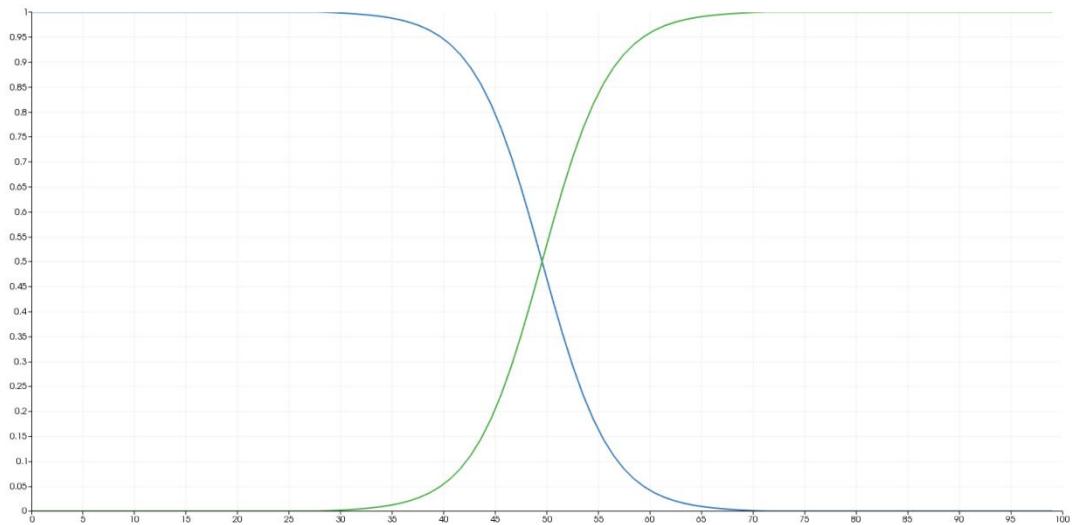
案例见 benchmark。

首先采用 Steinbach 1996 的梯度能，对比 Nestler 改进的势能模型

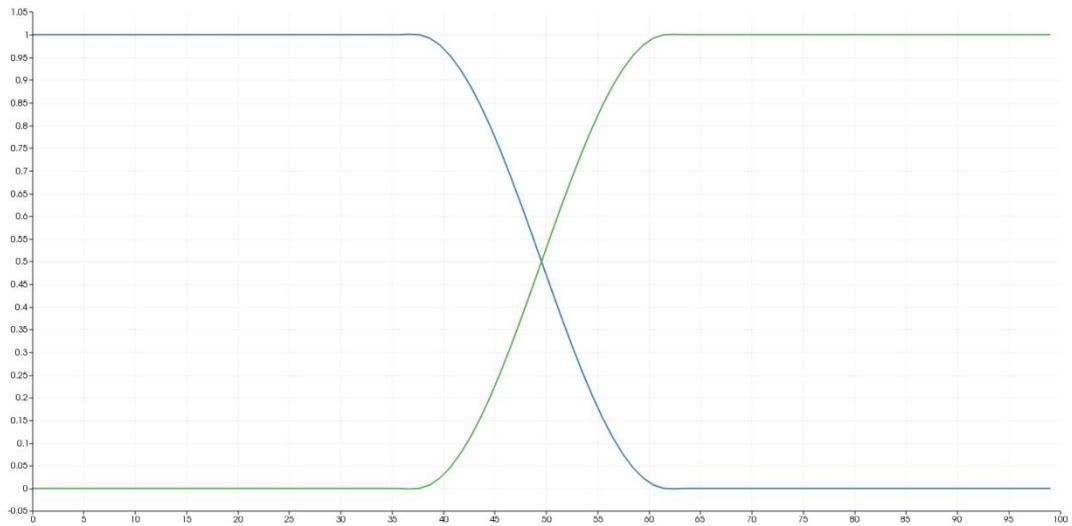


实线是 well 模型的计算结果，虚线是 obstacle 模型的。显然 obstacle 模型具有超出 0~1 的范围的特性，而 well 模型的界面会延展的更宽。

采用 well 模型，切换 Steinbach1996 和 1999 模型，得到结果如下

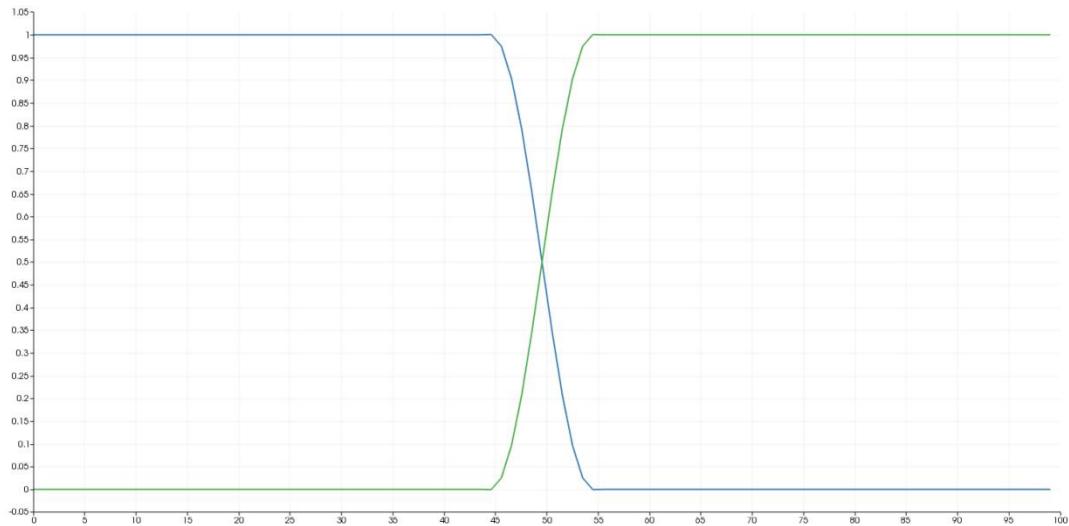


采用 obstacle 模型，切换 Steinbach1996 和 1999 模型，得到结果如下



显然，在一维情况下，Steinbach 在 1996 年与 1999 年给出的模型算出的结果几乎没有差别。

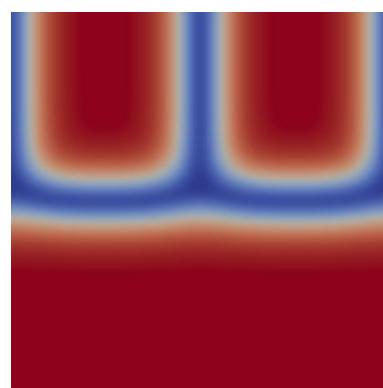
接下来选择 Steinbach2009 年改进的模型



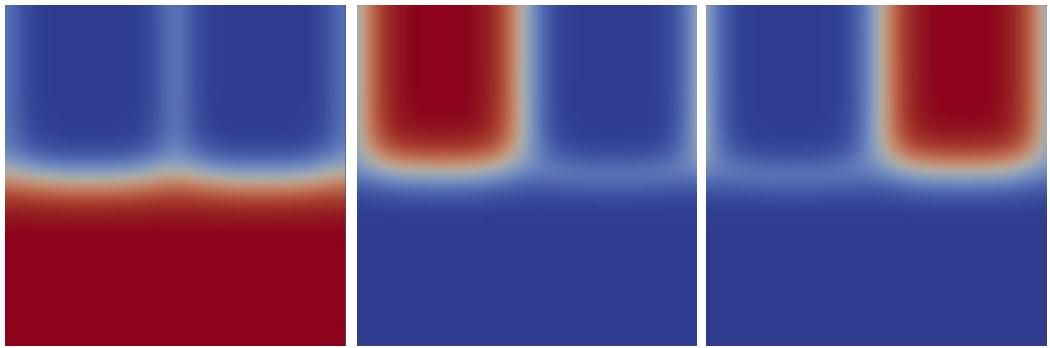
在界面宽度都设置为 10 的情况下，可以看到 Steinbach2009 改进后的模型更加窄，同时界面宽度也在 10 左右。

5.2. 二维界面研究

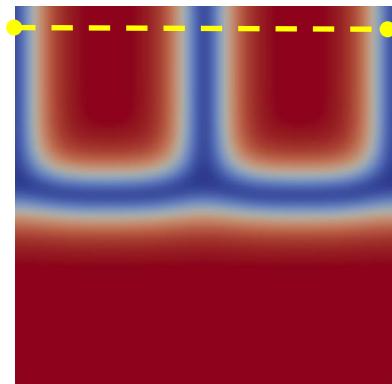
在多晶模拟中，较为常见的现象是某一晶粒的相分数值出现在它不该出现的地方，比如存在如下的三晶粒模拟，案例见 benchmark



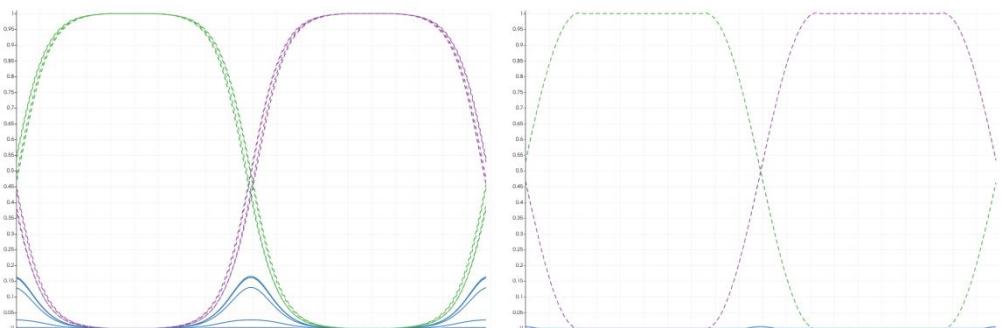
我们分别取出这三个晶粒的相分数值，可以发现三个晶粒的相分数并不局限于其晶粒存在区域的周围，而是延展到了其他晶粒的晶界上。这种现象在 well 和 obstacle 势中都有存在，尤以在 well 势中表现的十分明显。



Nestler 在常规界面势中引入三相共存势能项 $\xi^{\alpha\beta\gamma}\phi^\alpha\phi^\beta\phi^\gamma$ 和 $\xi^{\alpha\beta\gamma}(\phi^\alpha\phi^\beta\phi^\gamma)^2$ 即为针对该问题的解决方案，同时 Steinbach 在 2009 年对界面能演化方程的优化对处理该问题也有很好的效果。

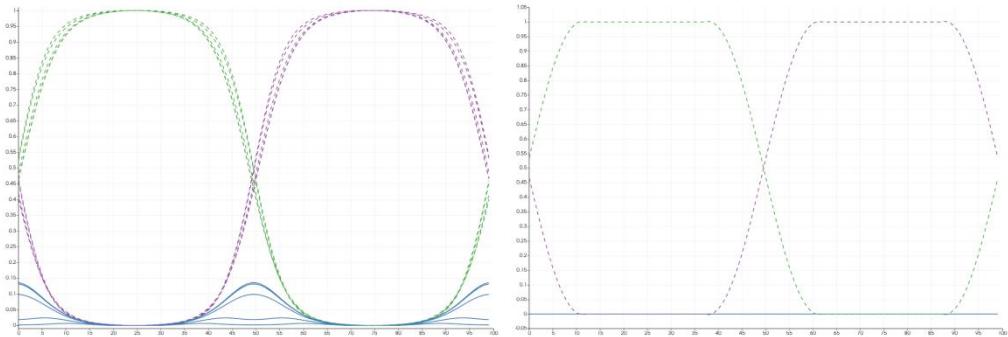


为了探究各模型在该问题上的作用，取黄色虚线上的三相分数的值，增大 $\xi^{\alpha\beta\gamma}$ （取值 $0*\xi^{\alpha\beta}$ 、 $10*\xi^{\alpha\beta}$ 、 $100*\xi^{\alpha\beta}$ 、 $1000*\xi^{\alpha\beta}$ 、 $10000*\xi^{\alpha\beta}$ ）直到蓝色曲线在界面上贴近 0 为止，每种情况都将相分数演化至平衡



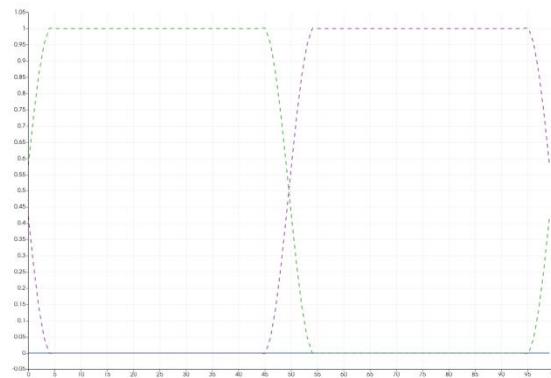
S.96+N.well

S.96+N.obstacle



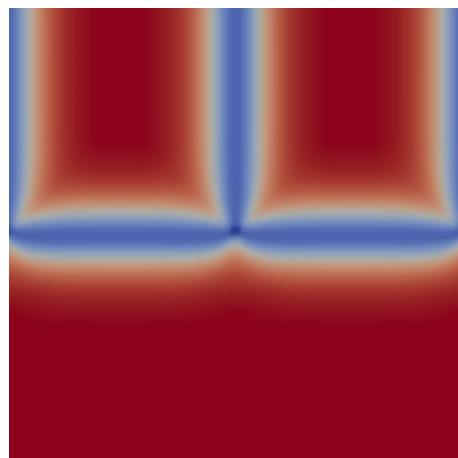
S.99+N.well

S.99+N.obstacle



S.2009

可以看见，在 $\xi^{\alpha\beta\gamma}$ 取值为 0 时，仅有 S.99+N.obstacle 和 S.2009 中第三相的相分数为 0，它没有在界面上与其他两相混合在一起，这说明了 Obstacle 所能起到的限制作用。而对于 Well 势，Nestler 所给出的三相共存势能确实能够很好地限制第三相的相分数，但 $\xi^{\alpha\beta\gamma}$ 也不是越高越好，当 $\xi^{\alpha\beta\gamma}$ 取到 $\xi^{\alpha\beta}$ 的 10000 倍时，界面变形严重

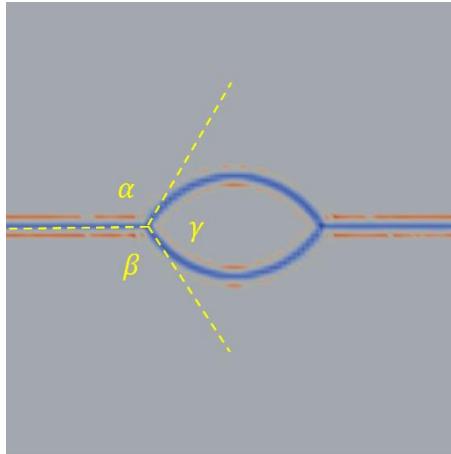


$$\xi^{\alpha\beta\gamma} = 10000 * \xi^{\alpha\beta}$$

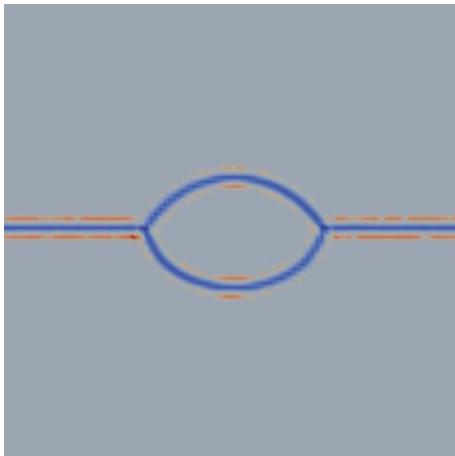
5.3. 多相结

在两相交界处生成一个椭圆，假设三相的界面能相互相等（都为 1.0），进行一定时长的演化，可得稳定角度，其角度由杨氏方程可描述，案例见 benchmark

$$\xi^{\alpha\beta} + \xi^{\alpha\gamma} \cos(\alpha) + \xi^{\beta\gamma} \cos(\beta) = 0$$



在上述基础上修改 $\xi^{\beta\gamma} = 0.3$ ，得稳定角度



5.4. 给一个驱动

在上面一维界面研究的基础上给界面一个驱动力，让界面动起来，当然该方法多维通用，案例见 benchmark。

首先在 runfile 中定义两个概念相，这里假设分别为 Alpha 与 Beta：

```
enum PHASE{Alpha, Beta};
```

接着，告述程序我们已经定义了两个相，当然是通过 information 类的设置

```
inf.materialSystem.phases.add_Phase(PHASE::Alpha, "alpha");
```

```
inf.materialSystem.phases.add_Phase(PHASE::Beta, "beta");
```

在设置 inf 中给定基体相时声明基体相的 property 为 Alpha，当然基体相的晶粒编号依旧默认为 0

```
inf.materialSystem.matrix_phase.set(0, PHASE::Alpha);
```

在生成第二个晶粒时，将其的 phaseProperty 参数定义为 Beta，其晶粒编号为 1

```
geo.init(pf::Geometry::Geo_Polyhedron, 0, 1, PHASE::Beta, 0.0);
```

接下来根据变分公式，详见 2.3，在不存在组分的情况下自动简化为

$$\frac{\delta}{\delta \phi^\alpha(\mathbf{r}, t)} \Gamma(\mathbf{r}, t) = \frac{\delta}{\delta \phi^\alpha(\mathbf{r}, t)} f^{total}(\mathbf{r}, t)$$

因此两相的驱动力简化为两相的总能密度之差

$$\Delta G^{\alpha\beta} = \frac{\delta}{\delta \phi^\beta(\mathbf{r}, t)} f^{total}(\mathbf{r}, t) - \frac{\delta}{\delta \phi^\alpha(\mathbf{r}, t)} f^{total}(\mathbf{r}, t)$$

重写 MID_MESO 软件中的能量密度方程

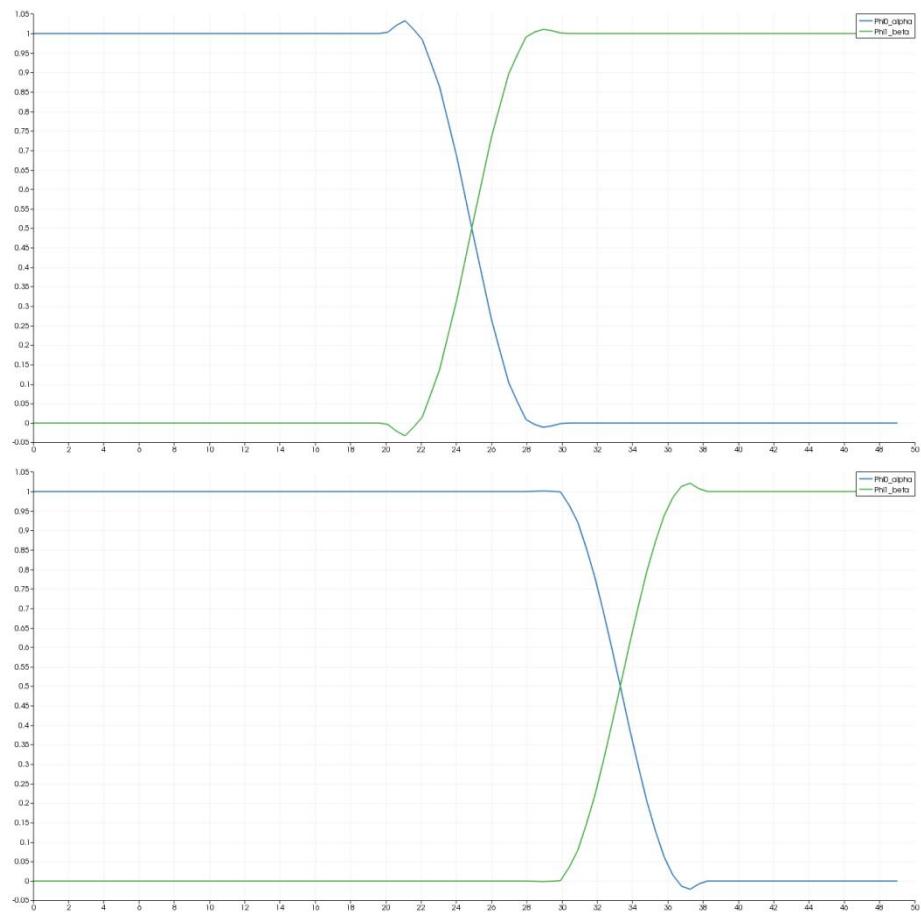
```
static void energy(pf::PhaseNode& n, pf::PhaseEntry& p, pf::Info_DynamicCollection& inf)
{
    if (p.phaseProperty == PHASE::Alpha)
        p.chemEnergyDensity = -0.1;
    else if (p.phaseProperty == PHASE::Beta)
        p.chemEnergyDensity = 0;
    return;
}

inf.materialSystem.functions.Energy = energy;
```

这里写的无论是哪个能量密度（化学能、弹性能 or 其他）都能起到相同效

果，程序会自动叠加这些能量为总能密度，并应用于驱动力计算之中。因此，在这里我们创建了一个 Alpha 类晶粒长大，Beta 类晶粒缩小的驱动力，驱动力大小为 0.1。当然 Alpha 类的晶粒个数不限，Beta 类晶粒个数也不限，且能量密度大小可以按使用者定义为任意函数，当然可以跟热力学吉布斯自由能函数衔接。

然后我们可以得到一个推进的界面，可见由于蓝色基体相能量密度更低（-0.1），其在长大



5.5. 界面上相组分研究

案例详见 benchmark。

在多相模型与相浓度模型的耦合中，存在三类基本理论：

- 根据物质守恒，晶粒的收缩与膨胀会导致晶粒内溶质总摩尔分数的变化（当溶剂不存在时，即所有组分）

$$\phi^\alpha(\mathbf{r}, t) \frac{\partial x_i^\alpha(\mathbf{r}, t)}{\partial t} + \frac{\partial \phi^\alpha(\mathbf{r}, t)}{\partial t} x_i^\alpha(\mathbf{r}, t) = 0$$

- 当系统处于远离热力学平衡的状态时，在一定外界条件下，由于系统内部非线性相互作用，可以经过突变而形成新的有序结构——耗散结构。这里的耗散指的是系统维持这种新型结构需要从外界输入能量或物质。**耗散结构理论**是比利时科学家普里高津在研究非平衡热力学过程中提出。假设不同的晶粒属于不同的耗散结构，每个晶粒通过彼此间的物质交换、能量交换、负熵交换来维持和发展有序结构，并完成整个模拟体系的演化。多相场的能量竞争，其实质可以理解为能量的交换，低能晶粒生长而总能增加、高能晶粒收缩而总能降低；相浓度界面上的耗散行为则属于物质的交换，其同样可由物质的势驱动，由高势晶粒输入低势晶粒

$$P_i^{\alpha\beta}(\mathbf{r}, t) = \frac{1}{\phi^\alpha} |\nabla \phi^{\alpha\beta}| \kappa_i^{\alpha\beta} V_m \left[\frac{\delta}{\delta x_i^\beta(\mathbf{r}, t)} \Gamma(\mathbf{r}, t) - \frac{\delta}{\delta x_i^\alpha(\mathbf{r}, t)} \Gamma(\mathbf{r}, t) \right]$$

- 而晶粒内物质的扩散现象是一个基于分子热运动的输运现象，是分子通过布朗运动从高浓度区域(或高化势)向低浓度区域(或低化势)的运输的过程。它是趋向于热平衡态的驰豫过程，是熵驱动的过程

$$\phi^\alpha(\mathbf{r}, t) \frac{\partial x_i^\alpha(\mathbf{r}, t)}{\partial t} = V_m \nabla \cdot \left\{ \phi^\alpha(\mathbf{r}, t) V_m \sum_{j=1}^{n-1} \left[{}^\alpha M_{ij}(\mathbf{r}, t) \nabla \frac{\delta}{\delta x_j^\alpha(\mathbf{r}, t)} \Gamma(\mathbf{r}, t) \right] \right\}$$

在本节中同样可基于上面一维界面研究来测试这一过程，假设这两个晶粒属于同一类相且仅含物质 i，物质 i 有从高浓度流向低浓度的趋势，以上方程皆可简化为浓度差驱动

$$P_i^{\alpha\beta}(\mathbf{r}, t) = \frac{|\nabla \phi^{\alpha\beta}|}{\phi^\alpha} \kappa_i^{\alpha\beta} [x_i^\beta(\mathbf{r}, t) - x_i^\alpha(\mathbf{r}, t)]$$

$$\begin{aligned}\phi^\alpha(\mathbf{r}, t) \frac{\partial x_i^\alpha(\mathbf{r}, t)}{\partial t} &= V_m \nabla \cdot [\phi^\alpha(\mathbf{r}, t) V_m^\alpha M_{ii}(\mathbf{r}, t) \nabla x_i^\alpha(\mathbf{r}, t)] \\ &= \nabla \cdot [\phi^\alpha(\mathbf{r}, t)^\alpha D_{ii}(\mathbf{r}, t) \nabla x_i^\alpha(\mathbf{r}, t)]\end{aligned}$$

设置晶粒内物质扩散遵守浓度梯度

```
inf.settings.details settings.flux model = PhaseFluxModel::IntDiff_ConGrad;
```

假设存在晶粒类编号为 0, 存在组分编号为 0 (仅方便 MID-MESO 识别)

```
const int phase_property = 0, component = 0;
```

告述 MID-MESO 存在相 0 (名为“Grains”), 该相含有物质 0 (名为“X”), 整个模拟域含有物质 0

```
inf.materialSystem.phases.add Phase(phase_property, "Grains");
```

```
inf.materialSystem.phases[phase_property].x.add_nodeEntry(component,  
"X");
```

```
inf.materialSystem.sys x.add_nodeEntry(component);
```

定义基体相时, 同时为基体相的组分 0 赋值, 为低浓度晶粒

```
inf.materialSystem.matrix_phase.set(0, phase_property);
```

```
inf.materialSystem.matrix_phase.x.add_con(component, 0.1);
```

定义第二晶粒时, 同时为该晶粒的组分 0 赋值, 为高浓度晶粒

```
pf::GeometricRegion geo;
```

```
geo.init(pf::Geometry::Geo_Polyhedron, 0, 1, phase_property, 0.0);
```

```
geo.x.add_x(component, 0.9);
```

由于涉及扩散过程, 需定义扩散系数

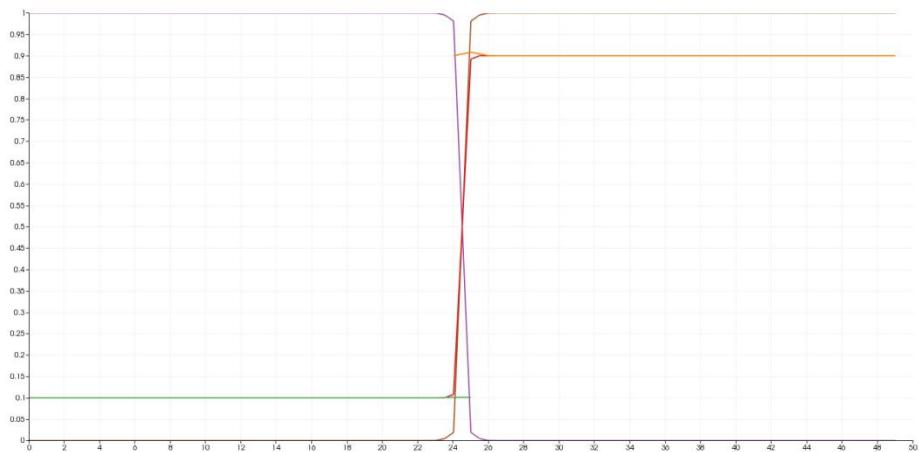
```
static void chemicalDiffusivity(pf::PhaseNode& n, pf::PhaseEntry& p,  
pf::Info_DynamicCollection& inf) {  
    p.kinetics_coeff.set(component, component, 1.0);  
    return;  
}
```

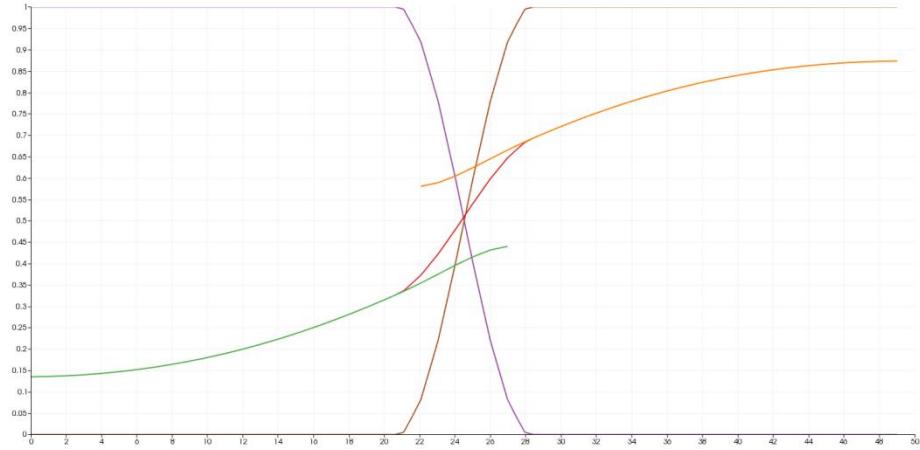
```
inf.materialSystem.functions.ChemicalDiffusivity = chemicalDiffusivity;
```

由于涉及界面上的耗散，需定义界面上的耗散反应，它同样属于广义界面反应的一种，定义反应速率为 0.1，其驱动力为两相组分浓度差

```
static void intphaseReaction(pf::PhaseNode& n, pf::PhaseEntry& alpha, pf::PhaseEntry& beta,  
pf::Info_DynamicCollection& inf) {  
    double reaction_rate = 0.1;  
    Vector3 norm = pf::OuterFunctions::normals(alpha, beta, inf);  
    for (auto comp1 = alpha.x.begin(); comp1 < alpha.x.end(); comp1++)  
        for (auto comp2 = beta.x.begin(); comp2 < beta.x.end(); comp2++)  
            if (comp1->index == comp2->index){  
                double pre = reaction_rate * norm.abs() * (comp2->value - comp1->value);  
                if (pre > 0 && comp2->value > X_EPSILON && comp1->value < (1.0 - X_EPSILON)){  
                    comp1->ChemicalReactionFlux += pre / alpha.phaseFraction;  
                    comp2->ChemicalReactionFlux -= pre / beta.phaseFraction;  
                }  
                else if (pre < 0 && comp1->value > X_EPSILON && comp2->value < (1.0 - X_EPSILON)){  
                    comp1->ChemicalReactionFlux += pre / alpha.phaseFraction;  
                    comp2->ChemicalReactionFlux -= pre / beta.phaseFraction;  
                }  
            }  
    return;  
}  
inf.materialSystem.functions.InterPhasesReaction = intphaseReaction;
```

基于该 runfile 的模拟结果为



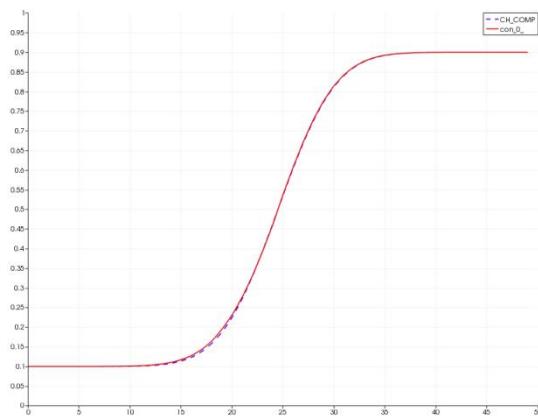


上方图中，紫线为基体分布，土线为第二晶粒分布，绿线为基体内组分 0 分布，黄线为第二晶粒内组分 0 分布，红色线为组分 0 在全域的分布（关于晶粒的相分数加权求和）。可见两晶粒的组分 0 在模拟之初相差很大，分别是 0.1 和 0.9；而在耗散作用下，第二晶粒内的组分 0 逐渐耗散至基体内，因此第二晶粒界面处组分 0 含量降低，而基体界面处组分 0 含量升高；且在各晶粒皆存在扩散过程，因此在扩散作用下基体界面处的组分 0 向基体内部扩散，而第二晶粒内部的组分 0 向其界面处扩散。

进一步，调用 Cahn-Hilliard 求解器，求解相同浓度场。定义 CH 方程扩散速率为 0.1。同时，定义相浓度场扩散速率为 0.1，耗散速率遵守关系

$$\kappa_c^{\alpha\beta} = \frac{\omega}{dr} M_{cc}$$

其中， ω 拟合为 4.0。对比结果（虚线：CH，实线：相浓度场）：



6. 其他场的简单调用

6.1. 温度场

温度演化公式为

$$\frac{\partial T(\mathbf{r}, t)}{\partial t} = \nabla \cdot \sum_{\alpha} \phi^{\alpha} D_{temp}^{\alpha} \nabla T(\mathbf{r}, t) + H(\mathbf{r}, t)$$

其中温度场初始化可以通过定义全局初始温度，及借助形核工具帮助区域
初始温度

```
inf.materialSystem.init_temperature = 0.0;  
  
pf::GeometricRegion geo;  
  
geo.init(pf::Geometry::Geo_Polyhedron, 0, 0, 0, 0.0);  
  
geo.temperature = 1.0;
```

其中各相的热扩散系数可以在告述 MID-MESO 相的信息时，一同给出

```
inf.materialSystem.phases.add_Phase(0, "grain");  
  
inf.materialSystem.phases[0].heat_diffusivity = 1.0;
```

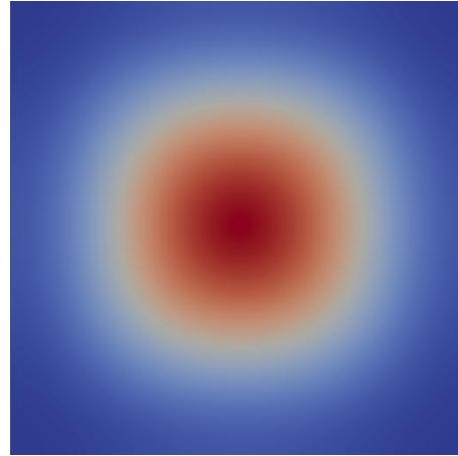
热源一般不存在时可以忽略，如果存在热源可以重写热源函数

```
static double heatSource(pf::PhaseNode& n, pf::Info_DynamicCollection& inf){  
    return 0.0;  
}  
  
inf.materialSystem.functions.HeatSource = heatSource;
```

更详细的热源函数可以参考 example 中的 dendrite 案例。同时我们需要在
main 函数的主循环中引用温度场的演化方程和赋值方程

```
simulation.evolve_temperature_evolution_equation(istep);  
  
simulation.temperature_assignment(istep);
```

可得结果如下



6.2. 弹性场

弹性能密度为

$$\begin{aligned} f^{elas}(\mathbf{r}, t) &= \frac{1}{2} \sigma^{el}(\mathbf{r}, t) \epsilon^{el}(\mathbf{r}, t) \\ &= \frac{1}{2} (\epsilon_{mn}(\mathbf{r}, t) - \epsilon_{mn}^*(\mathbf{r}, t)) C_{mnop}(\mathbf{r}, t) (\epsilon_{op}(\mathbf{r}, t) - \epsilon_{op}^*(\mathbf{r}, t)) \\ \epsilon_{mn}^*(\mathbf{r}, t) &= \sum_{\alpha} \phi^{\alpha}(\mathbf{r}, t) \epsilon_{mn}^{*,\alpha}(\mathbf{r}, t) \\ C_{mnop}(\mathbf{r}, t) &= \sum_{\alpha} \phi^{\alpha}(\mathbf{r}, t) C_{mnop}^{\alpha}(\mathbf{r}, t) \end{aligned}$$

相场微弹性理论 (Armen G. Khachaturyan)

$$\frac{\partial \varepsilon_{ij}^0(\mathbf{r}, t)}{\partial t} = LC_{ijkl}^0 \left\{ \begin{array}{l} \frac{1}{2} \int_{|\mathbf{k}| \neq 0} [n_k \Omega_{lm}(\mathbf{n}) + n_l \Omega_{km}(\mathbf{n})] \tilde{\sigma}_{mn}^0(\mathbf{k}) n_n e^{i\mathbf{k} \cdot \mathbf{r}} \frac{d^3 k}{(2\pi)^3} \\ - \Delta S_{klmn}(\mathbf{r}) C_{mnpq}^0 [\varepsilon_{pq}^0(\mathbf{r}) - \varepsilon_{pq}^*(\mathbf{r})] - \varepsilon_{kl}^*(\mathbf{r}) - \bar{\varepsilon}_{kl}^0 + S_{klmn}^0 \sigma_{mn}^{ex} \end{array} \right\}$$

弹性场有两个输入参数，分别为 $\epsilon_{mn}^{*,\alpha}(\mathbf{r}, t)$ 和 $C_{mnop}(\mathbf{r}, t)$ ，其中本征应变需要

重写设置内的函数

```
static vStrain effectivePhaseEigenStrains(pf::PhaseNode& n, pf::PhaseEntry& p,
pf::Info DynamicCollection& inf) {
    vStrain strain;
    if (p.index == 1) {
        strain[0] = 0.01; strain[1] = 0.01; strain[2] = 0.01;
    }
    else if (p.index == 0) {
        strain[0] = 0.0; strain[1] = 0.0; strain[2] = 0.0;
    }
}
```

```
    return strain;
}
inf.materialSystem.functions.EffectivePhaseEigenStrains = effectivePhaseEigenStrains;
```

本征应变可以是一个与浓度、温度等场变量相关的复杂函数，而弹性常数可以需要选择弹性常数模型：Khachaturyan、IngoSteinbach、Custom

```
inf.materialSystem.mechanics.effectiveElasticConstantsModel =
EffectiveElasticConstantsModel::EEC_Khachaturyan;
```

Khachaturyan 模型的有效弹性常数为各相弹性常数的加权求和，IngoSteinbach 模型的有效弹性常数为各相弹性常数的逆的加权求和再求逆，自定义弹性常数模型需要重写有效弹性常数函数。

对于 Khachaturyan 和 IngoSteinbach 模型来说 MID-MESO 简化了使用方式，仅需在告述 MID-MESO 相的信息同时，定义该相的弹性常数

```
inf.materialSystem.phases.add_Phase(0, "grain");
double Gm = 1.0, v = 0.3, Az = 3.0, C12 = v * Gm / (1 - 2 * v), C11 = 2 * Gm / Az +
C12;
inf.materialSystem.phases[0].elastic_constants(0, 0) = C11;
inf.materialSystem.phases[0].elastic_constants(1, 1) = C11;
inf.materialSystem.phases[0].elastic_constants(2, 2) = C11;
inf.materialSystem.phases[0].elastic_constants(1, 0) = C12;
inf.materialSystem.phases[0].elastic_constants(2, 0) = C12;
inf.materialSystem.phases[0].elastic_constants(2, 1) = C12;
inf.materialSystem.phases[0].elastic_constants(0, 1) = C12;
inf.materialSystem.phases[0].elastic_constants(0, 2) = C12;
inf.materialSystem.phases[0].elastic_constants(1, 2) = C12;
inf.materialSystem.phases[0].elastic_constants(3, 3) = Gm;
inf.materialSystem.phases[0].elastic_constants(4, 4) = Gm;
inf.materialSystem.phases[0].elastic_constants(5, 5) = Gm;
```

而考虑到即便同一类相的晶粒也有不同的旋转方向，因此每个晶粒的旋转都可以被定义（旋转矩阵生成原理与前面形核类同）

```
inf.materialSystem.mechanics.rotation_gauge = RotationGauge::RG_XYX;
double radien[] = {AngleToRadians(0.0), AngleToRadians(0.0), AngleToRadians(0.0)};
inf.materialSystem.mechanics.grainRotation.add_matrix3(0,
RotationMatrix::rotationMatrix(radien, RotationGauge::RG_XYX));
```

```
inf.materialSystem.mechanics.grainRotation.add_matrix3(1,  
RotationMatrix::rotationMatrix(radien, RotationGauge::RG_XYX));
```

需注意的是 Khachaturyan 和 IngoSteinbach 模型定义的各相弹性常数会被自动旋转，而重写的本征应变及弹性常数函数都需要使用者自己调用旋转函数旋转

```
double radien[] = { AngleToRadians(0.0), AngleToRadians(0.0) , AngleToRadians(0.0) };  
eigenstrain.do_rotate(RotationMatrix::rotationMatrix(radien, RotationGauge::RG_XYX));
```

另外，相要弹性场正常运转，还需要设置启动弹性场（为了避免内存的无端消耗）

```
inf.materialSystem.is_mechanics_on = true;
```

目前程序中集成了两种弹性场的求解函数：

1、 Armen G. Khachaturyan 的相场微弹性理论，其调用函数为

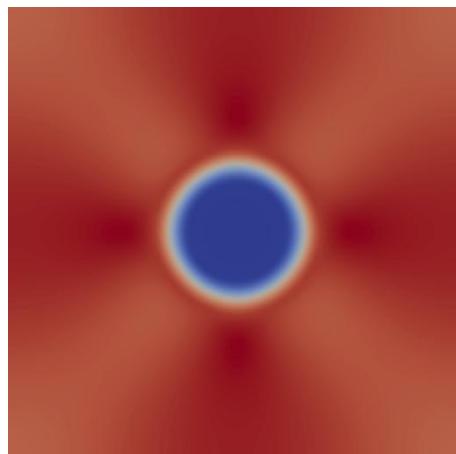
```
simulation.mechanics.SetEffectiveEigenStrains();  
simulation.mechanics.SetEffectiveElasticConstants();  
simulation.mechanics.initVirtualEigenstrain();  
simulation.mechanics.Solve2(istep, 1e-6, 1000, 1.0, true); //参数设置仅参考
```

2、 Steinbach 的求解法

```
simulation.mechanics.SetEffectiveEigenStrains();  
simulation.mechanics.SetEffectiveElasticConstants();  
simulation.mechanics.Solve(1e-4, 1e-2, 1000, false); //参数设置仅参考
```

弹性模拟结果包含本征应变分量、应变分量、应力分量、J1 主应力、vMises

屈服准则，J1 主应力展示



6.3. 电场

电场的电势电荷平衡求解遵守泊松方程

$$\nabla \cdot \sum_{\alpha} {}^e \varepsilon^{\alpha} \nabla^e \varphi(\mathbf{r}, t) = - {}^e \rho(\mathbf{r}, t)$$

输入参数为各相的电导率及各组分所带的电荷，同样在告述 MID-MESO 相的信息时给出

```
inf.materialSystem.phases.add Phase(0, "grain");
```

```
inf.materialSystem.phases[0].conductivity = 1.0;
```

同时，我们需要定义电场的初始状态及边界条件，如下定义初始电势分布都是 0，而求解过程中晶粒 1 的电势始终为 1.0

```
inf.materialSystem.electricFieldMask settings.init electric potential = 0.0;
```

```
inf.materialSystem.electricFieldMask settings.electric potential phase index.  
add_double(1, 1.0);
```

同样，为避免内存的浪费，我们需要打开电场求解的开关

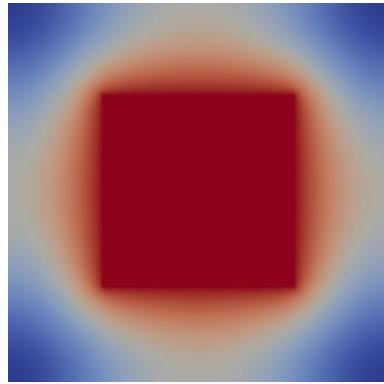
```
inf.materialSystem.is_electrics_on = true;
```

而在主函数中，我们需要在主循环中调用如下两函数，来实现每个时间步都求解电势电荷的平衡

```
simulation.electricField.setElectricChargeDensity();
```

```
simulation.electricField.solve(1e-4, 10000);
```

求解的电势分布结果如下



6.4. 磁场

本部分理论部分由组内曾银平博士提供，同样，磁场的磁势磁荷平衡求解遵守泊松方程

$$\nabla^2 m \varphi(\mathbf{r}, t) = -m \rho(\mathbf{r}, t) = \nabla \cdot \mathbf{M}(\phi, \mathbf{r}, t)$$

使用磁场需重写函数

```
static void magnetizationIntensity(pf::PhaseNode& n, pf::Info_DynamicCollection& inf)
{
    n.magneticValues.mag_intensity[1] = n[0].phaseFraction;
}
inf.materialSystem.functions.MagnetizationIntensity = magnetizationIntensity;
```

及其对相分数的变分

```
static void dMagnetizationIntensity_dPhi(pf::PhaseNode& n, pf::PhaseEntry& p,
                                         pf::Info_DynamicCollection& inf) {
    double vec[] = { 0.0, 0.0, 0.0 };
    if (p.index == 0) {
        vec[1] = 1.0;
        n.magneticValues.dmag_intensity_dphi.add_vec(p.index, vec);
    }
    else {
        n.magneticValues.dmag_intensity_dphi.add_vec(p.index, vec);
    }
}
inf.materialSystem.functions.dMagnetizationIntensitydPhi =
dMagnetizationIntensity_dPhi;
```

磁势磁荷的求解同样需要给定初始状态和边界条件

```
inf.materialSystem.magneticFieldMask.settings.init_magnetic_potential = 0.0;
```

```
inf.materialSystem.magneticFieldMask.settings.is_averaged = true;
```

```
inf.materialSystem.magneticFieldMask.settings.average_value = 0.0;
```

为了避免内存的浪费，磁势磁荷的求解也需要打开开关

```
inf.materialSystem.is_magetics_on = true;
```

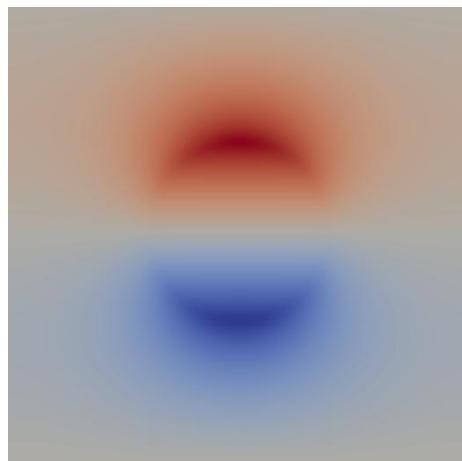
在主函数中，我们需要在主循环中调用如下三函数，来实现该理论框架下磁势磁荷的求解（参数仅参考）

```
simulation.magneticField.setMagneticParameters();
```

```
simulation.magneticField.solve_magnetizationIntensity(1e-4, 10000, true, 100);
```

```
simulation.magneticField.solve_dMagnetizationIntensity_dPhi(1e-4, 10000,  
true, 100);
```

能够得到的磁场分布结果如下



6.5. 流场

流场求解器被用于求解不可压粘性流动问题，其基本函数模型为不可压的纳维-斯托克（Navier-Stokes）方程：

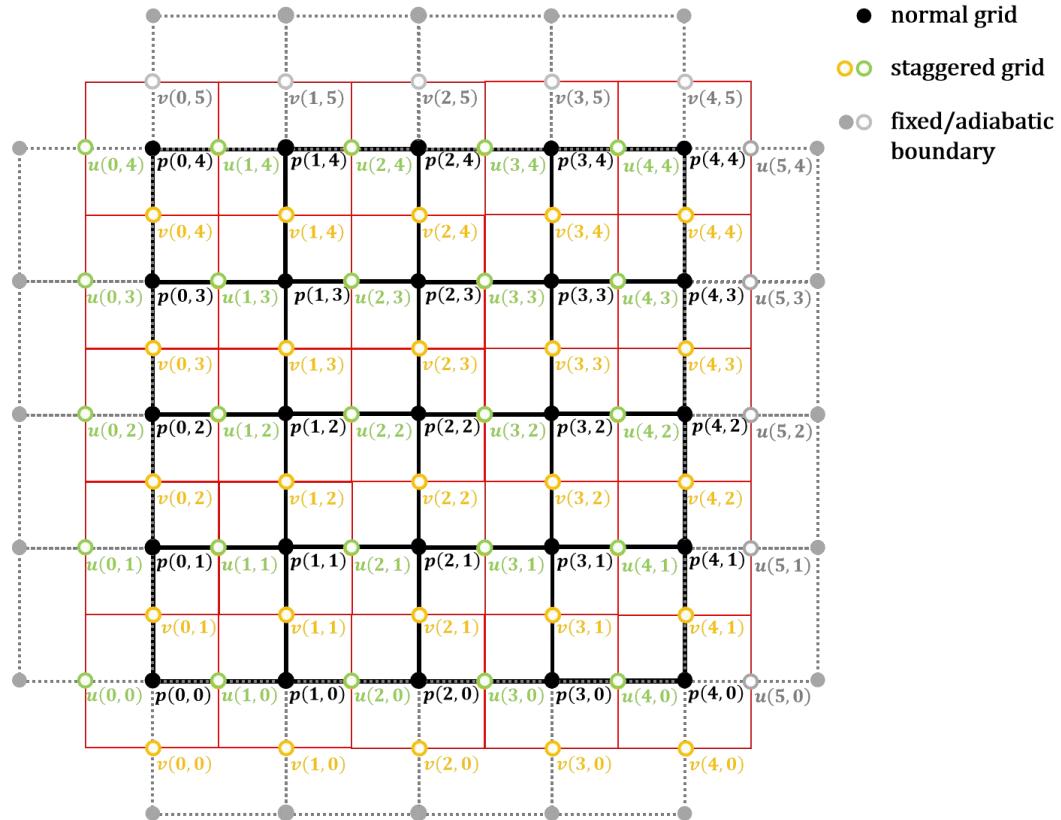
(1) 连续性方程

$$\nabla \cdot \mathbf{V}(\mathbf{r}, t) = 0$$

(2) 动量方程

$$\rho(\mathbf{r}, t) \frac{D(\mathbf{V}(\mathbf{r}, t))}{Dt} = -\nabla p(\mathbf{r}, t) + \nabla \cdot \mu(\mathbf{r}, t) \nabla \mathbf{V}(\mathbf{r}, t) + \rho(\mathbf{r}, t) \mathbf{f}(\mathbf{r}, t)$$

为使压强 p 和速度 \mathbf{V} 的求解更加稳定，采用交错网格法离散速度场空间：



在进行边界条件设置时，可参考上面的网格样例。其中压强所在网格（黑色网格）即为 MID-MESO 的主网格格点，包含包括相分数、浓度、外场等一系列信息数据。而速度场网格（红色网格），分别包含 u 、 v 、 w 三个方向的速度分量。

使用时需打开流场开关，并设置流相的密度、粘度和流相对应的相场相 index：

```
inf.materialSystem.is_fluid_on = true;
inf.materialSystem.fluidFieldMask.settings.density = 1.0;
inf.materialSystem.fluidFieldMask.settings.viscosity = 1.0;
```

```
inf.materialSystem.fluidFieldMask.settings.fluid_phase_box.push_back(0);
```

边界条件的设置在流场求解中，**至关重要！**

```
static void boundary U(pf::VectorNode& u, pf::PhaseNode& down_node, pf::PhaseNode&
up_node, int Nx, int Ny, int Nz) {
    if ((u.x == Nx - 1 || u.x == 0) && u.y > 4.0 && u.y < 16.0) {
        u.vals[0] = 1.0;
    }
    if (u.y == 0) {
        u.vals[0] = 0.0;
    }
    if (u.y == Ny - 1) {
        u.vals[0] = 0.0;
    }
    if(down_node[1].phaseFraction > 0.999 || up_node[1].phaseFraction > 0.999)
        u.vals[0] = 0.0;
}

static void boundary V(pf::VectorNode& v, pf::PhaseNode& down_node, pf::PhaseNode&
up_node, int Nx, int Ny, int Nz) {
    if (v.y == 0) {
        v.vals[0] = 0.0;
    }
    if (v.y == Ny - 1) {
        v.vals[0] = 0.0;
    }
    if (down_node[1].phaseFraction > 0.999 || up_node[1].phaseFraction > 0.999)
        v.vals[0] = 0.0;
}

static void boundary W(pf::VectorNode& w, pf::PhaseNode& down_node, pf::PhaseNode&
up_node, int Nx, int Ny, int Nz) {
    w.vals[0] = 0.0;
}

static void boundary main(pf::PhaseNode& node, int Nx, int Ny, int Nz) {
    if (node.x == 0) {
        node.velocityValues.pressure = 0.0;
    }
    else if (node.x == Nx - 1) {
        node.velocityValues.pressure = 0.0;
    }
    node.velocityValues.volume_force.set_to_zero();
}

simulation.fluidField.set_boundary_condition_for_domain_U(boundary U);
simulation.fluidField.set_boundary_condition_for_domain_V(boundary V);
```

```
simulation.fluidField.set_boundary_condition_for_domain_W(boundary_W);  
simulation.fluidField.set boundary condition for main domain(boundary_main);
```

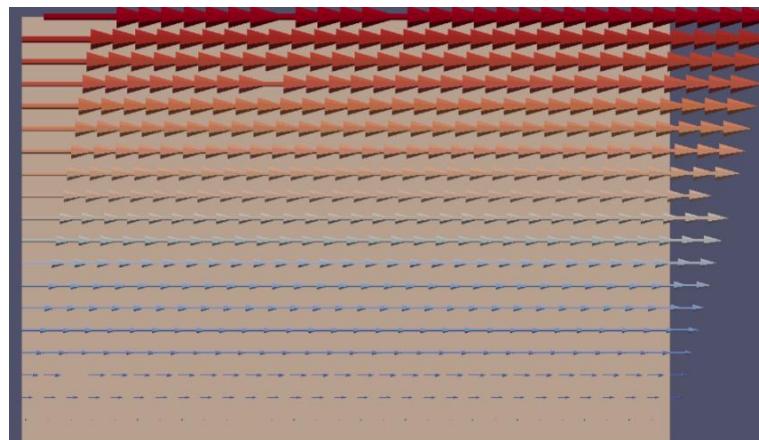
调用压力修正法求解流场：

```
simulation.fluidField.evolve_momentum_equation(fluid_dt);  
simulation.fluidField.do_pressure_correction(1e-4, 0.8, false, 100);  
simulation.fluidField.correcting_velocity_field(fluid_dt);  
simulation.fluidField.do_boundary_condition();  
simulation.fluidField.assign_velocity_to_main_domain();
```

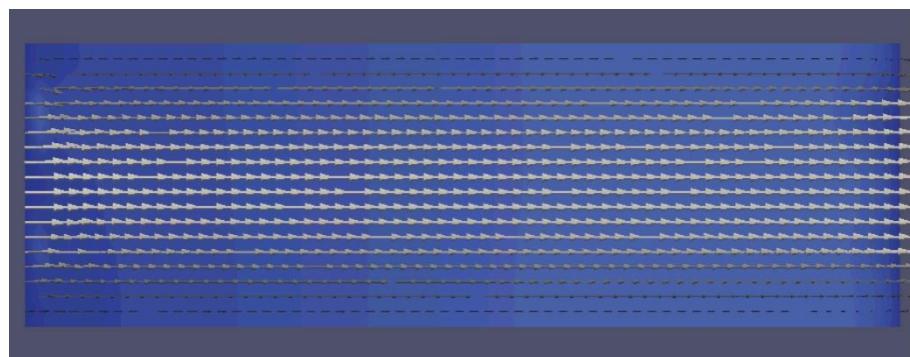
输出速度场和压强：

```
inf.settings.file_settings.isFluidFieldOutput = true;
```

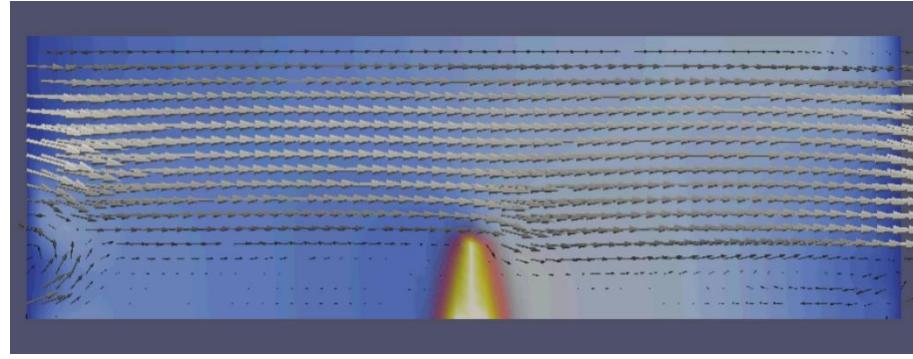
在 benchmark 中有流场的基本案例，couette 流案例如下：



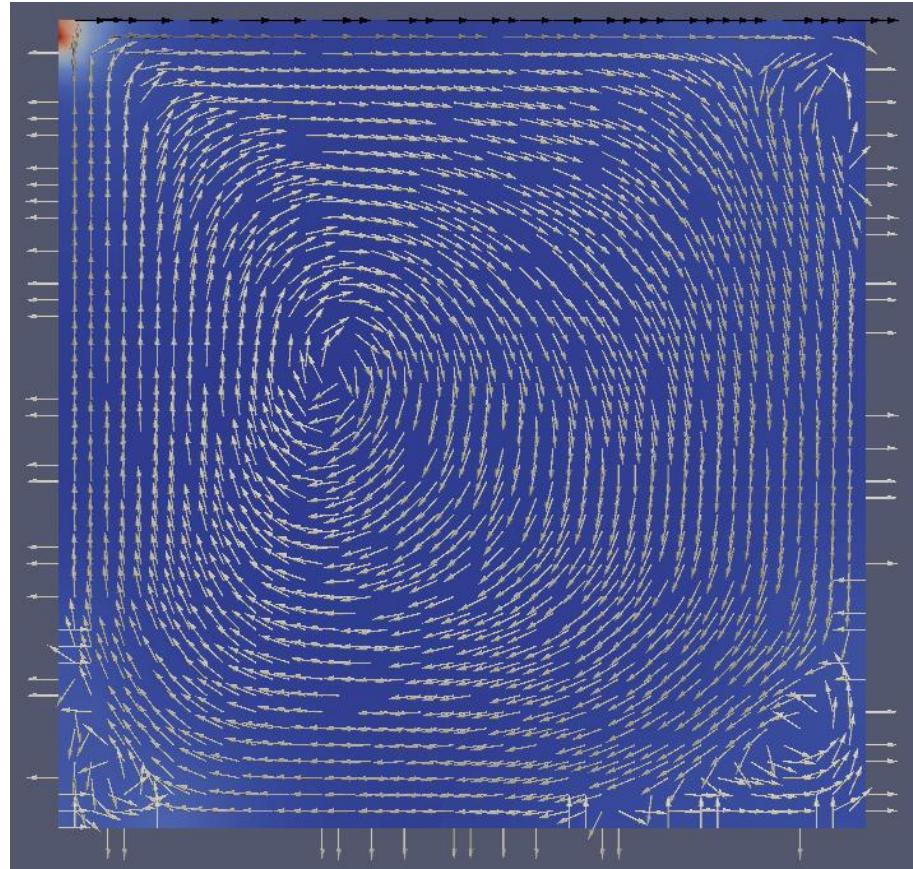
正常管道流如下：



管道中加入障碍物（另一个相）：



在 example 中有流场的进阶案例，方腔流案例如下：



6.6. 泊松方程求解器

泊松方程求解器适用于求解以下类型的函数模型， R 为待求解变量， LHS 为左侧输入量， RHS 为右侧输入量

$$\nabla \cdot LHS(\mathbf{r}, t) \nabla R(\mathbf{r}, t) = RHS(\mathbf{r}, t)$$

比如使用该求解器求解电场方程，定义 R 为电势、 LHS 为与电导率、 RHS 为与电荷有关的函数，首先需定义 LHS 、 R 与 RHS 的概念符号

```
enum ElectricField{ElectricPotential, ElecChargeDensity_Conduction};
```

然后，我们需要在 main 函数中，在主循环前初始化该求解器，且在模拟网格中生成 LHS 和 RHS 的储存空间

```
pf::PoissonEquationSolver electricfield_solver(simulation.phaseMesh,  
ElectricPotential, ElecChargeDensity_Conduction);
```

模拟前，我们需要初始化 LHS 和 RHS 的值分别都为 0.0

```
electricfield_solver.init_field(0.0, 0.0);
```

随后我们需要重写求解器中的边界条件函数和 RHS 函数

```
static void boundary(pf::PhaseNode& node, int lhs_index) {  
  
    if (node[1].phaseFraction > Simulation_Num_Cut_Off)  
  
        node.customValues[lhs_index] = 1.0;  
  
}  
  
static void rhs_cal(pf::PhaseNode& node, int rhs_index) {  
  
    node.customValues[rhs_index] = 0.0;  
  
}
```

并且用这两个自定义的函数覆盖求解器 electricfield_solver 中的默认函数

```
electricfield_solver.set_BoundaryCondition_calfunc(boundary);  
  
electricfield_solver.set_RHS_calfunc(rhs_cal);
```

最后，我们需要在主循环中调用求解器的求解函数即可完成电场的求解

```
electricfield_solver.solve(1e-4, 10000, true, 500); //参数仅参考
```

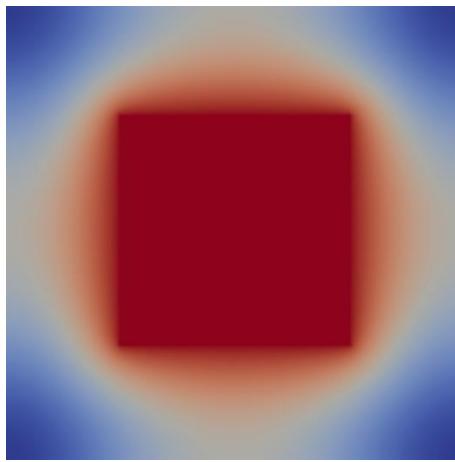
需要注意的是，泊松方程求解器借用了网格上自定义变量空间，该类型变量因其极大自主性无法自动输出，因此需手动调用函数输出

```

simulation.write_customValue_in_node("electric_potential", istep,
ElectricPotential);

```

如果是长时间的模拟，需要在上面函数前加上时间步判断，以确保不会每一步都输出电场分布导致电脑内存不足。基于上述步骤，利用泊松方程求解器来自定义电场求解器，我们能够获得如下电势分布结果



6.7. Allen-Cahn 方程求解器

Allen-Cahn 方程求解器适用于求解以下类型的函数模型， $A_i(\mathbf{r}, t)$ 为被解场变量， $L_{ij}(\mathbf{r}, t)$ 为关联变化速率， $\frac{\delta}{\delta A_j(\mathbf{r}, t)}$ 为自定义变分函数（界面能、体相能分离）， $S_{ij}(\mathbf{r}, t)$ 为关联源， $S_i(\mathbf{r}, t)$ 为独立源：

$$\frac{\partial A_i(\mathbf{r}, t)}{\partial t} = \sum_j \left[-L_{ij}(\mathbf{r}, t) \frac{\delta}{\delta A_j(\mathbf{r}, t)} (F_{int}(\mathbf{r}, t) + F_{bulk}(\mathbf{r}, t)) + S_{ij}(\mathbf{r}, t) \right] + S_i(\mathbf{r}, t) \#$$

例如使用 Allen-Cahn 方程求解一个双晶体系中的晶粒长大，首先我们需要定义晶粒的数量

```
static int grains_number = 2;
```

然后我们需要初始化该求解器，并在模拟网格的每个格点的自定义变量空间内给出存储 Allen-Cahn 求解器所需的空间

```
pf::AllenCahnSolver allenCahn(simulation.phaseMesh);
```

```
allenCahn.init field(grains number, pf::SOLVER ALLEN CAHN);
```

同时借助 MPF 的给自定义变量赋值的函数，为求解器中的基体晶粒赋初值

```
simulation.add_customValue_to_allnodes(allenCahn.grains_start_index, 1.0);
```

对于 Allen-Cahn 求解器我们需要重写三个函数

```
static double L(pf::PhaseNode& node, int grain, int grains_start_index){  
    double L = 5.0;  
    return L;  
}  
  
static double dF_dphi(pf::PhaseNode& node, int grain, int grains_start_index){  
    double A = 1.0, B = 1.0, kappa = 1.0  
    , dF_dphi = 0.0, phi = node.customValues[grain + grains_start_index],  
    drivingforce = 0.0, sum = 0.0;  
    for (int index = grains_start_index; index < grains_start_index + grains_number;  
        index++)  
        if (index != (grain + grains_start_index))  
            sum += node.customValues[index] * node.customValues[index];  
    if (grain == 1)  
        drivingforce = -1.0;  
    dF_dphi = -A * phi + B * phi * phi * phi + 2.0 * phi * sum - kappa *  
    node.cal_customValues_laplace(grain + grains_start_index, 1.0,  
    DifferenceMethod::FIVE POINT) + phi * (1 - phi) * drivingforce;  
    return dF_dphi;  
}  
  
static double Source(pf::PhaseNode& node, int grain, int grains_start_index){  
    double Source = 0.0;  
    return Source;  
}
```

同理，我们需要覆盖 Allen-Cahn 求解器中的默认函数

```
allenCahn.set_dF_dphi_func(dF_dphi);
```

```
allenCahn.set_L_func(L);
```

```
allenCahn.set_Source_func(Source);
```

然后，我们仅需要在主循环中调用求解函数即可

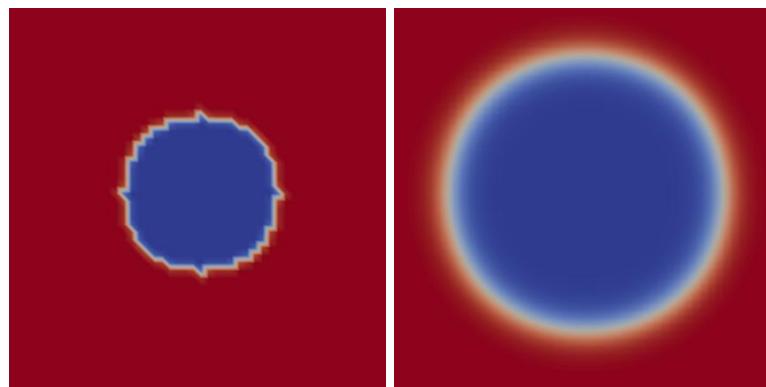
```
allenCahn.solve_one_step(istep, simulation.information.settings.disperse_settings.dt);
```

但同样也是因为借助的模拟网格中的自定义变量，我们也需要借助 MPF 类

中的辅助输出函数输出 vts 文件

```
if (istep % 50 == 0) {
    cout << "# allenCahn solver work: MAX VARIATION of step " <<
    to_string(istep) << " is " << to_string(max variation) << endl;
    for (int index = allenCahn.grains start index; index <
        allenCahn.grains start index + grains number; index++)
        simulation.write customValue in node("grain" + to_string(index -
            allenCahn.grains start index), istep, index);
}
```

结果如下所示



对于该求解器更加成熟的利用，可以参考 example 中的 multi_cellular 及 sintering 案例。

6.8. Cahn-Hilliard 方程求解器

Cahn-Hilliard 方程求解器适用于求解以下类型的函数模型， c 为组分， M 为组分迁移速率， $\frac{\delta}{\delta c_i} F$ 为自定义组分的势， $SOURCE_i$ 为该物质的源

$$\frac{\partial c_i}{\partial t} = \nabla M_i \nabla \frac{\delta}{\delta c_i} F + SOURCE_i$$

假设通过 Cahn-Hilliard 求解器求解一个物质扩散过程，首先我们需要定义物质的种类

```
static int comps_number = 1;
```

同样我们也需初始化 Cahn-Hilliard 求解器、模拟网格及对应组分的初始值

```
pf::CahnHilliardSolver cahnHilliard(simulation.phaseMesh);
```

```
cahnHilliard.init_field(comps_number, pf::SOLVER_CAHN_HILLIARD);
simulation.add_customValue_to_allnodes(cahnHilliard.comps_start_index, 0.001);
```

我们也需要自定义三个函数来替换默认函数

```
static double Mobility(pf::PhaseNode& node, int grain, int grains_start_index) {
    double m = 1.0;
    return m;
}

static double dF_dc(pf::PhaseNode& node, int grain, int grains_start_index) {
    double dF_dc = 0.0;
    dF_dc = node.customValues[grains_start_index + grain];
    return dF_dc;
}

static double Source(pf::PhaseNode& node, int grain, int grains_start_index) {
    double Source = 0.0;
    return Source;
}

cahnHilliard.set_dF_dc_func(dF_dc);

cahnHilliard.set_Mobility_func(Mobility);

cahnHilliard.set_Source_func(Source);
```

同时，我们需要在主循环中调用 Cahn-Hilliard 求解器的求解函数，已经半

手动定义输出

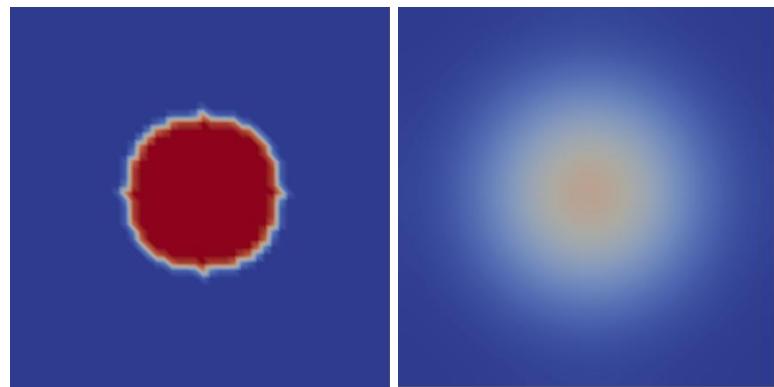
```
cahnHilliard.solve_one_step(istep,
simulation.information.settings.disperse_settings.dt, DifferenceMethod::FIVE_POINT);
if (istep % 100 == 0) {
    cout << "# allenCahn solver work: MAX_VARIATION of step " <<
    to_string(istep) << " is " << to_string(max_variation) << endl;
    for (int index = cahnHilliard.comps_start_index; index <
    cahnHilliard.comps_start_index + comps_number; index++)
        simulation.write_customValue_in_node("comp" + to_string(index) -
        cahnHilliard.comps_start_index), istep, index);
}
```

值得一提的是，与 Allen-Cahn 求解器的各个晶粒一样，Cahn-Hilliard 求解器的各个组分也可以借助形核函数中的几何形核来初始化，它可以将几何范围内的自定义变量初始化为相要的值

```
pf::GeometricRegion geo;
```

```
geo.init(pf::Geometry::Geo_Ellipsoid);  
geo.customValues.add_double(pf::SOLVER_CAHN_HILLIARD + 0, 0.999);
```

通过该模拟可以得到结果如下



对于该求解器更加成熟的利用，可以参考 example 中的 sintering 案例。

7. 搭建一个自己的数据库/函数库

通过上面的学习，你已经掌握了如何通过复写一个程序中自带的默认函数来自定义个人化的模拟，你可以构建属于自己的模型。此时，你已经是一个蓄势待发的有望在相场领域一展拳脚的研究者！

一个成熟的研究者需要有一套成熟的方法来管理自己的模拟体系的各种物理参数或者涉及到的模型，所以不能简单地把模型放在 runfile 文件中，因此你需要构建属于自己的数据库/函数库。这样每构建一个新的模拟，你仅需要复制黏贴自己的函数库，然后 include 该数据库/函数库，即可简单地展开计算。并且一个成熟的数据库/函数库可以实现不同函数的再次组装，避免了相同函数的反复定义。

比如对于上面给出的各种求解器，你可以在自己的数据库中构建一个电场的求解器，然后每次展开一个新的模拟就 include 该电场求解器，这样该求解器便成为了一个新的模块。以此类推，MID-MESO 软件便有了拓展的可能。

另外，存在于设置中的可被覆盖的物理参数/模型函数有

(详见 OuterFunction.h， 默认函数也大多存于此文件中)

```
double (*Ghser) (pf::PhaseNode&, int index);
void (*Energy) (pf::PhaseNode&, pf::PhaseEntry&, pf::Info_DynamicCollection&);
void (*Potential) (pf::PhaseNode&, pf::PhaseEntry&, pf::Info_DynamicCollection&);
double (*MolarVolume) (pf::PhaseNode&, pf::PhaseEntry&, pf::Info_DynamicCollection&);
double (*EnergyMinimizerIterator) (pf::PhaseNode&, pf::PhaseEntry&, pf::Info_DynamicCollection&);
void (*ChemicalMobility) (pf::PhaseNode&, pf::PhaseEntry&, pf::Info_DynamicCollection&);
void (*ChemicalDiffusivity) (pf::PhaseNode&, pf::PhaseEntry&, pf::Info_DynamicCollection&);
vStrain (*EffectivePhaseEigenStrains) (pf::PhaseNode&, pf::PhaseEntry&, pf::Info_DynamicCollection&);
Matrix6x6 (*EffectiveElasticConstants) (pf::PhaseNode&, pf::Info_DynamicCollection&);
void (*MagnetizationIntensity) (pf::PhaseNode&, pf::Info_DynamicCollection&);
void (*dMagnetizationIntensitydPhi) (pf::PhaseNode&, pf::PhaseEntry&, pf::Info_DynamicCollection&);
void (*InterPhasesReaction) (pf::PhaseNode&, pf::PhaseEntry&, pf::PhaseEntry&, pf::Info_DynamicCollection&);
void (*InnerPhaseReaction) (pf::PhaseNode&, pf::PhaseEntry&, pf::Info_DynamicCollection&);
double (*PhaseSource) (pf::PhaseNode&, pf::PhaseEntry&, pf::PhaseEntry&, pf::Info_DynamicCollection&);
double (*HeatSource) (pf::PhaseNode&, pf::Info_DynamicCollection&);
double (*Xi_ab) (pf::PhaseNode&, pf::PhaseEntry&, pf::PhaseEntry&, pf::Info_DynamicCollection&);
double (*Xi_abc) (pf::PhaseNode&, pf::PhaseEntry&, pf::PhaseEntry&, pf::PhaseEntry&, pf::Info_DynamicCollection&);
double (*Mobility) (pf::PhaseEntry&, pf::PhaseEntry&, pf::Info_DynamicCollection&);
Vector3 (*Normals) (pf::PhaseEntry&, pf::PhaseEntry&, pf::Info_DynamicCollection&);
void (*Nucleation) (pf::PhaseNode&, pf::NP_Node&, pf::Info_DynamicCollection&);
// double is int_width
double (*dfint_dphi) (pf::PhaseNode&, pf::PhaseEntry&, double, pf::Info_DynamicCollection&);
```

这些物理参数/模型函数因为接收了模拟网格格点上的场变量、自定义变量的值，并且都有修改场变量的权限，因此这些函数都具有无限的拓展的可能，它们既可以被使用为具有明确物理意义的函数，也可成为一个概念化的函数在模型框架的某一位置上来控制相场模型的演化。因此，熟练地掌握函数模型及模型的框架、拓展已有的模型、构建属于自己的新的模型，在此过程中不断扩张属于自己的数据库/函数库，这样不断地成长，你便能成为一个真正的强者。

关于如何构建一个完美的数据库，在数据量匮乏的情况下，谁也无法给出一个准确解，在此仅给出 Al-Cu 体系的案例作为参考，使用者可以通过命名空间来层层筛选自己相要的体系、相、函数等，对于个人使用而言该方法是一个简单且行之有效的方法。但，对一个课题组而言，如果期望构建一个庞大的数据库，该方法又无法兼顾数据保密的问题，因此本人在 benchmark 中给出的构建数据库方案仅做参考。

最后，我想说的是：

相场法不是程序黑盒，也不是参数玄学，它有简单而明确的理论基础——能量下降，它也有理论上无限可能的能量构建，它同时具有科学的严谨、数学的逻辑和图形的美妙。

以建模之法演化万象，这才是相场的真谛，愿共勉！