

S. Bulent Biner

Programming Phase-Field Modeling

EXTRAS ONLINE

 Springer

Programming Phase-Field Modeling

S. Bulent Biner

Programming Phase-Field Modeling



Springer

S. Bulent Biner
Idaho National Laboratory
Idaho Falls, ID, USA

ISBN 978-3-319-41194-1 ISBN 978-3-319-41196-5 (eBook)
DOI 10.1007/978-3-319-41196-5

Library of Congress Control Number: 2016951933

© Springer International Publishing Switzerland 2017

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made.

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To my wife Züllal

Preface

This book aims to introduce students and scientists to the programming of the phase-field method. The codes provided in the book will serve as a foundation and template for developing other phase-field models with more complexity as demanded by their underlying physics. The Matlab/Octave programming language was chosen for the codes presented in the book. This approach provides a very efficient and compact connection with the mathematical formulism and its numerical implementations; moreover, they can be easily expanded to higher level programming languages (e.g., Fortran, C/C++, and MPI). Therefore, this book provides a fast track to numerical implementation of phase-field modeling, the numerical details of which are usually omitted in literature. Particular attention was devoted to computational efficiency and clarity during development of the codes, with the latter most often being given a higher preference. Therefore, if it is desired, they can be further improved, with a little effort, and can be turned into production codes due to the extremely well-optimized nature of Matlab/Octave.

This book is not intended to provide an extensive survey of phase-field modeling or to be an exercise book for Matlab/Octave programming. There are plenty of references and textbooks that cover both of these subjects. It is hoped that this book will serve as a cookbook for programming of phase-field modeling. However, with this in mind, the book starts with the historical background and fundamental formulism of the phase-field method.

In the following chapters, three numerical algorithms, namely finite differencing, spectral methods, and finite element methods, are developed with increasing complexity and the capability to solve the equations of the phase-field method. For each algorithm, the presented case studies start with a description of the model and are followed by its formulism, numerical implementation, results obtained from the solution, discussion, and the source codes. The same two case studies are repeated with each of the three algorithms. Any numerical algorithm can be characterized in terms of its properties, which are accuracy, flexibility to handle many different problems, robustness, and computational efficiency. Very often, it is difficult to achieve all of these properties in one particular algorithm. The aim of this book is that the reader, after hands-on experience, can understand the fine details and strengths and weaknesses of each given solution's methods sufficiently well and can adopt the suitable algorithm needed in their models.

This book facilitates a comparative study with a collection of algorithms. Chapter 7 is devoted to a more complex and new class of phase-field models termed ‘‘Phase-Field Crystal Method.’’ This method bridges the traditional phase-field theory to the atomistic scale, enabling simulations on diffusive time scales that are not achievable with current atomistic simulations.

The computer programs and background materials discussed in each section of this book also provide a forum for undergraduate level modeling–simulation courses as part of their curriculum. Even though there are no specific exercises provided, if desired, exercises can be easily devised in variety of ways (for example, by modifying either material-specific properties or simulation-specific properties).

Of course, writing this book is the product of many stimulating discussions with colleagues, graduate students, and research associates over the years, and all their valuable contributions are greatly appreciated. Special thanks to Michael Luby, senior editor of Springer, for his patience, skills, and recommendations; it was a great pleasure working with him.

Idaho Falls, ID

S. Bulent Biner

Contents

1	An Overview of the Phase-Field Method and Its Formalisms	1
	References	5
2	Introduction to Numerical Solution of Partial Differential Equations	9
2.1	Introduction	9
2.2	Basic Numerical Methods	9
	References	11
3	Preliminaries About the Codes	13
	References	15
4	Solving Phase-Field Models with Finite Difference Algorithms	17
4.1	Introduction	17
4.2	One-Dimensional Transient Heat Conduction: A Solution with Finite Difference Algorithm	19
4.3	Source Codes	19
	References	21
4.4	Case Study-I Simulation of the spinodal decomposition of a binary alloy with explicit Euler finite difference algorithm	21
4.4.1	Background	21
4.4.2	Phase-Field Model	22
4.4.3	Numerical Implementation	22
4.4.4	Results and Discussion	23
4.4.5	Source Codes	26
	References	34
4.5	Case Study-II Phase-field modeling of grain growth with finite-difference algorithm	35
4.5.1	Background	35
4.5.2	Phase-Field Model	36
4.5.3	Numerical Implementation	36

4.5.4	Results and Discussion	37
4.5.5	Source Codes	39
	References	50
4.6	Case Study-III Phase-field modeling of solid-state sintering	51
4.6.1	Background	51
4.6.2	Phase-Field Model	51
4.6.3	Numerical Implementation	52
4.6.4	Results and Discussion	52
4.6.5	Source Codes	53
	References	68
4.7	Case Study-IV Phase-field modeling of dendritic solidification	69
4.7.1	Background	69
4.7.2	Phase-Field Model	70
4.7.3	Numerical Implementation	71
4.7.4	Results and Discussion	71
4.7.5	Source Codes	72
	References	81
4.8	Case Study-V Multicellular systems, the role of elastic mismatch and cell motility	81
4.8.1	Background	82
4.8.2	Phase-Field Model	82
4.8.3	Numerical Implementation	83
4.8.4	Results and Discussion	83
4.8.5	Source Codes	85
	References	97
5	Solving Phase-Field Models with Fourier	
	Spectral Methods	99
5.1	Introduction	99
5.2	One-Dimensional Transient Heat Conduction: A Solution with Fourier Spectral Algorithm	99
5.3	Source Code	101
	References	102
5.4	Case Study-VI Simulation of spinodal decomposition of a binary alloy with semi-implicit Fourier spectral algorithm	102
5.4.1	Background	102
5.4.2	Phase-Field Model	102
5.4.3	Numerical Implementation	103
5.4.4	Results and Discussion	103
5.4.5	Source Code	104
	References	110
5.5	Case Study-VII Phase-field modeling of grain growth with semi-implicit Fourier spectral algorithm	110
5.5.1	Background	110

5.5.2	Phase-Field Model	110
5.5.3	Numerical Implementation	111
5.5.4	Results and Discussion	111
5.5.5	Source Codes	113
	References	121
5.6	Case Study-VIII Phase-field simulation of precipitation behavior of a Fe-Cu-Mn-Ni alloy	121
5.6.1	Background	121
5.6.2	Phase-Field Model	122
5.6.3	Numerical Implementation	124
5.6.4	Results and Discussion	124
5.6.5	Source Codes	124
	References	137
5.7	Case Study-IX The role of elastic inhomogeneities and applied stress on the phase-separation behavior of a binary alloy	137
5.7.1	Background	137
5.7.2	Phase-Field Model	138
5.7.3	Solving the Equation of Mechanical Equilibrium	138
5.7.4	Numerical Implementation	140
5.7.5	Results and Discussion	140
5.7.6	Source Codes	144
	References	155
5.8	Case Study-X The role of lattice defects on spinodal decomposition of a Fe-Cr alloy	156
5.8.1	Background	156
5.8.2	Phase-Field Model	156
5.8.3	Numerical Implementation	157
5.8.4	Results and Discussion	157
5.8.5	Source Codes	161
	References	168
6	Solving Phase-Field Equations with Finite Elements	169
6.1	Introduction	169
6.2	Isoparametric Representation and Numerical Integration	169
6.3	Introduction to Strong and Weak Forms of FEM Formulation	171
6.3.1	FEM Discretization of Weak Form	172
6.3.2	Discretization in Time for Transient Heat Transfer	173
6.4	FEM Formulation of Linear Elasticity Based on the Principles of Virtual Work	174
6.4.1	A Numerical Example for FEM Solution of Linear Elasticity	176

6.5	Source Codes	176
	References	196
6.6	Case Study-XI Simulation of spinodal decomposition of a binary alloy with finite element method	196
6.6.1	Background	197
6.6.2	Phase-Field Model	197
6.6.3	FEM Implementation	197
6.6.4	Numerical Implementation	197
6.6.5	Results and Discussion	199
6.6.6	Source Codes	199
	References	218
6.7	Case Study-XII Phase-field modeling of grain growth with finite element method	218
6.7.1	Background	218
6.7.2	Phase-Field Model	219
6.7.3	FEM Formulation	219
6.7.4	Numerical Implementation	219
6.7.5	Results and Discussion	220
6.7.6	Source Codes	222
	References	237
6.8	Case Study-XIII Phase-field modeling of instabilities in layered thin films	237
6.8.1	Background	238
6.8.2	Phase-Field Model	238
6.8.3	FEM Formulation	238
6.8.4	Numerical Implementation	239
6.8.5	Results and Discussion	240
6.8.6	Source Codes	243
	References	272
6.9	Case Study-XIV Phase-field modeling of multi-variant martensitic transformations	272
6.9.1	Background	273
6.9.2	Phase-Field Model	273
6.9.3	FEM Formulation	275
6.9.4	Numerical Implementation	276
6.9.5	Results and Discussion	277
6.9.6	Source Codes	279
	References	302
6.10	Case Study-XV Phase-field modeling of brittle fracture	303
6.10.1	Background	303
6.10.2	Phase-Field Model	303
6.10.3	FEM Formulation	305

6.10.4	Numerical Implementation	306
6.10.5	Results and Discussion	307
6.10.6	Source Codes	310
	References	335
7	Phase-Field Crystal Modeling of Material Behavior	337
7.1	Introduction	337
	References	339
7.2	Case Study-XVI Numerical Implementations of Phase-Field Crystal Method	339
7.2.1	Phase-Field Crystal Model	340
7.2.2	Numerical Implementation	340
7.2.3	Results and Discussion	341
7.2.4	Source Codes	341
	References	355
7.3	Case Study-XVII Phase-Field Crystal Modeling of Grain Growth	355
7.3.1	Background	355
7.3.2	Phase-Field Crystal Model	355
7.3.3	Numerical Implementation	355
7.3.4	Results and Discussion	356
7.3.5	Source Codes	356
	References	361
7.4	Case Study-XVIII Phase-Field Crystal Modeling of Deformation Behavior of a Bicrystal	361
7.4.1	Background	361
7.4.2	Numerical Implementation	362
7.4.3	Results and Discussion	362
7.4.4	Source Codes	364
	References	368
8	Concluding Remarks	369
	Errata to: Programming Phase-Field Modeling	E1
	Appendix A	371
	Appendix B	377
	Appendix C	381
	Appendix D	391
	Index	397

List of Programs and Functions

Many functions are repeatedly used in many programs given in the text. This list gives their first appearances in the text.

<i>Program/Function Name</i>	<i>Chapter/ Case Study</i>	<i>Page Number</i>
Chapter 4		
heat_1d_fd.m		19
Chapter 4/Case Study-I		
fd_ch_v1.m		26
fd_ch_v2.m		28
micro_ch_pre.m		31
calculate_energ.m		31
free_energ_ch_v1.m		32
free_energ_ch_v2.m		32
write_vtk_grid_values.m		32
laplacian.m		34
Chapter 4/Case Study-II		
fd_ca_v1.m		39
fd_ca_v2.m		43
init_grain_micro.m		46
free_energ_fd_ca_v1.m		49
free_energ_fd_ca_v2.m		50
Chapter 4/Case Study-III		
fd_sint_v1.m		53
fd_sint_v2.m		61
micro_sint_pre.m		64
free_energ_sint_v1.m		66
free_energ_sint_v2.m		67
Chapter 4/Case Study-IV		
fd_den_v1.m		72
fd_den_v2.m		77
nucleus.m		79
gradient_mat.m		80
vect2matx.m		80

<i>Program/Function Name</i>	<i>Chapter/ Case Study</i>	<i>Page Number</i>
Chapter 4/Case Study-V		
fd_cell_dyna_v1.m		85
fd_cell_dyna_v2.m		90
micro_poly_cell.m		93
free_energ_v1.m		96
free_energ_v2.m		96
Chapter 5		
heat_1d_fft.m		101
Chapter 5/Case Study-VI		
fft_ch_v1.m		104
fft_ch_v2.m		107
prepare_fft.m		109
Chapter 5/Case Study-VII		
fft_ca_v1.m		113
fft_ca_v2.m		117
free_energ_fft_ca_v1.m		120
free_energ_fft_ca_v2.m		120
Chapter 5/Case Study-VIII		
fft_FeCuNiMn_v1.m		125
fft_FeCuNiMn_v2.m		128
init_FeCuNiMn_micro.m		131
Fe_Cu_Ni_Mn_free_energy.m		132
Chapter 5/Case Study-IX		
fft_raft_v1.m		144
fft_raft_v2.m		147
green_tensor.m		149
solve_elasticity_v1.m		150
solve_elasticity_v2.m		153
Chapter 5/Case Study-X		
fft_FeCr_v1.m		161
fft_FeCr_v2.m		164

(continued)

<i>Program/Function Name</i>	<i>Chapter/ Case Study</i>	<i>Page Number</i>
dislo_strain.m		166
FeCr_chem_potent_v1.m		167
FeCr_chem_potent_v2.m		167
Chapter 6		
fem_elast_v1.m		176
input_fem_elast.m		178
sfr2.m		179
gauss.m		180
jacob2.m		182
modps.m		183
bmats.m		183
dbe.m		184
stiffness.m		184
loads.m		186
boundary_cond.m		189
stress.m		189
output.m		191
write_vtk_fem.m		193
Chapter 6/Case Study-XI		
fem_ch_v1_2.m		199
init_micro_ch_fem.m		204
apply_periodic_bc.m		204
cart_deriv.m		205
chem_stiff_v1.m		206
chem_stiff_v2.m		211
free_energ_fem_v1.m		214
free_energ_fem_v2.m		215
input_fem_pf.m		215
jacob3.m		216
periodic_boundary.m		217
recover_slave_dof.m		218
Chapter 6/Case Study-XII		
fem_ca_v1_2.m		222
apply_periodic_bc2a.m		226
apply_periodic_bc2b.m		227
gg_free_energ_v1.m		227
gg_free_energ_v2.m		228
gg_rhs_1.m		228
gg_rhs_2.m		230
gg_stiff_1.m		232
gg_stiff_2.m		234
init_micro_grain_fem.m		235
Chapter 6/Case Study-XIII		
fem_thin_film.m		243
bmats2.m		246

<i>Program/Function Name</i>	<i>Chapter/ Case Study</i>	<i>Page Number</i>
boundary_cond_v1.m		246
boundary_cond_v2.m		247
chem_stiff_v3.m		248
chem_stiff_v4.m		252
dbe2.m		256
elastic_energ_v1.m		257
elastic_energ_v2.m		258
stiffness2.m		259
stiffness3.m		262
stress2.m		265
stress3.m		268
init_micro_thin_film.m		271
Chapter 6/Case Study-XIV		
fem_mart_v1_2.m		279
bmats1.m		283
boundary_cond2_v1.m		283
boundary_cond2_v2.m		284
init_mart_micro.m		285
marten_free_energ_v1.m		286
marten_free_energ_v2.m		286
marten_stiff_v1.m		287
marten_stiff_v2.m		293
material_parameters.m		298
stress_strain_v1.m		299
stress_strain_v2.m		301
Chapter 6/Case Study-XV		
fem_frac_v1_2.m		310
fract_stiff_v1.m		315
fract_stiff_v2.m		320
initialize.m		324
residual_v1.m		326
residual_v2.m		329
stress_fract_v1.m		332
stress_fract_v2.m		334
Chapter 7/Case Study-XVI		
pfc_2D_v1.m		343
pfc_2D_v2.m		346
pfc_3D_v1.m		348
pfc_3D_v2.m		351
prepare_fft_3d.m		353
write_vtk_grid_values_3D.m		354
Chapter 7/Case Study-XVII		
pfc_poly_v1.m		356
Chapter 7/Case Study-XVIII		
pfc_def_v2.m		364

An Overview of the Phase-Field Method and Its Formalisms

1

A microstructure can be described as the spatial arrangement of the phases and possible defects that have different compositional and/or structural character (for example, the regions composed of different crystal structures and having different chemical compositions, grains of different orientations, domains of different structural variants, and domains of different electric or magnetic polarizations). The size, shape, volume fraction, and spatial arrangement of these microstructural features determine the overall properties of any type of multiphase and/or multicomponent materials.

Although, phase-field modeling is relatively new paradigm, it has almost become the method of choice for modeling and simulation of microstructure evolution under different driving forces (e.g., compositional gradients, temperature, stress/strain, and electric and magnetic fields) in materials science and physics. Phase-field modeling is also now finding ever-expanding applications in biology, medicine, and earth sciences, as well as filling the much-needed gap between the atomistic scale and continuum level of modeling and simulations.

The motion of phase boundaries has attracted interest since the early nineteenth century. The problem is named after Josef Stefan, a Slovene physicist, who introduced solutions to this general class of problems around the 1890s. These solutions, which are known as sharp interface

approaches, have been used to explain the kinetics of diffusional phase transformations. The boundary conditions for a solution set of differential equations are explicitly defined at the phase interfaces in these approaches. However, solving these sets of nonlinear expressions turns out to be extremely difficult, stemming from the interface interactions with various complex processes (such as merging, dissolution, and breakup) during the course of phase transformations.

Phase-field modeling overcomes these difficulties through inclusion into its formalism of a different interface description. A phase-field model describes a microstructure (both the compositional and/or structural domains) with a set of field variables. The field variables are assumed to be continuous across the interfacial regions, which is opposite of that in sharp interface models where they are discontinuous. There are two types of field variables: conserved and non-conserved. The temporal evolution of the conserved field variables that is governed by the Cahn–Hilliard [1, 2] nonlinear diffusion equation (Eq. 1.1) and the non-conserved field variables evolve with the Allen–Cahn [3, 4] relaxation equation or time-dependent Ginzburg–Landau equation [5] (Eq. 1.2). These two kinetic equations underlie the basic physics of many common phase-field models in the literature.

$$\frac{\partial c_i}{\partial t} = \nabla M_{ij} \nabla \frac{\delta F}{\delta c_j(r, t)} \quad (1.1)$$

and

$$\frac{\partial \eta_p}{\partial t} = -L_{pq} \frac{\delta F}{\delta \eta_q(r,t)} \quad (1.2)$$

where r is the position, t is the time, M_{ij} is the diffusivities of the species, and c_1, c_2, \dots, c_n are conserved field variables. L_{pq} is the mobility of the non-conserved field variables, $\eta_1, \eta_2, \dots, \eta_m$. Subscripts are used to distinguish different chemical species and phases/domains. F is the free energy of the system.

For an inhomogeneous binary system, a free energy expression was derived by Cahn and Hilliard [1, 2]. They assumed that the free energy of an infinitesimal volume in a nonuniform system depends on both its composition and the composition of its nearby environment. The total free energy of the system cannot depend only on the local composition because different spatial configurations with the same volume fraction are not energetically equivalent. Therefore, they assumed that free energy depends on both composition and its derivatives for an inhomogeneous system. Cahn and Hilliard started with the homogeneous free energy density for binary system $f(c)$ and performed a Taylor expansion on $f(c)$ in terms of the derivatives of compositions to approximate $f(c, \nabla c, \nabla^2 c, \dots)$. Because the concentrations of a binary system must obey the relationship $c_1 + c_2 = 1$, only one concentration fraction and one chemical potential are independent. For an isotropic material, the free energy simplifies to an equation with constant coefficients and even powers of ∇c as:

$$f = f_0(c) + \kappa_1 \nabla^2 c + \frac{1}{2} \kappa_2 (\nabla c)^2 + \frac{1}{2} \kappa_3 (\nabla^2 c)^2 + \kappa_4 \nabla^4 c + \dots \quad (1.3)$$

They also argued that the derivative terms with even powers $\nabla^2 c, \nabla^4 c, \nabla^6 c$ should vanish. Keeping only the terms up to second order and discarding even power derivatives, they arrived at the free energy functional as:

$$F(c) = \int_V f_0(c) + \kappa (\nabla c)^2 dv \quad (1.4)$$

where the first term in the integrand is the chemical energy and k is a gradient energy coefficient that penalizes the formation of sharp interfaces.

Later, the free energy functional, developed by Wheeler et al. [6–8] to simulate isothermal phase transitions in binary alloys, was dependent on both composition and the non-conserved order parameter for the phase gradients:

$$F(f.c, \eta) = \int_V \left[f(c, \eta) + \frac{\kappa_c}{2} (\nabla c)^2 + \frac{\kappa_\eta}{2} (\nabla \eta)^2 dv \right] \quad (1.5)$$

where κ_c and κ_η are the composition and phase gradient energy coefficients, respectively. The chemical/homogeneous energy density $f(c, \eta)$ promotes phase separation in the absence of interface/gradient energies. In this approach, it is assumed that the composition of the phases at the interfaces is equal and interpolated from the compositions of the adjacent phases with:

$$f(c, \eta) = h(\eta) f_\alpha(c) + [1 - h(\eta)] f_\beta(c) \quad (1.6)$$

where $h(\eta)$ is the nondimensional interpolation function, has a minimum at $\eta = 0$ and $\eta = 1$, and provides a barrier for transition from one phase to the other. The form of the interpolation function is chosen for numerical convenience rather than any physical reasoning. A discussion on different interpolation functions can be found in Moelans work [9]. However, although the model treats the interface in a thermodynamic way, it is limited to binary systems with two free energy curves.

Early phase-field models investigated the dendritic solidification of pure and binary materials. Langer [10] and Fix [11] were the first to introduce a phase-field model for the first-order phase transitions. A similar diffusive interface model for solidification was independently developed by Collins and Levin [12]. Caginalp et al. [12–16] investigated the effects of anisotropy and the convergence of the phase-field equations to the conventional sharp interface models in the limit of vanishing phase interface thickness. Penrose and Fife [17, 18] provided a framework for deriving the phase-field equations in a thermodynamically consistent way from a single entropy

functional. However, the simulations of growth for two- and three-dimensional dendrites in the works of Kobayashi [19–21] demonstrated the potential of the phase-field modeling approach.

In the early phase-field models given above, only two phases were considered and they were suitable to study processes as solid–liquid transformations and phase separations in the solids states in binary alloys. The introduction of multiphases and multicomponents led to development of another class of formulism, including a free energy functional for many non-conserved order parameters, which are simultaneously evolving, given by Chen and Yang [22] as:

$$F = \int_V \left[f(\eta_1, \eta_2, \dots, \eta_N) + \sum_i^N \frac{\kappa_i}{2} (\nabla \eta_i)^2 \right] \quad (1.7)$$

in which

$$\begin{aligned} f(\eta_1, \eta_2, \dots, \eta_N) = & \sum_i^N \left[-\frac{A}{2} \eta_i^2 + \frac{B}{2} \eta_i^4 \right] \\ & + \sum_i^N \sum_j^N \eta_i^2 \eta_j^2 \end{aligned} \quad (1.8)$$

where A , B , and κ_i are the coefficients. Similar models were also developed by Morin et al. [23] and Kobayashi et al. [24].

Of course, in all these models, the width of the interface is the same as the physical width, which can be as small as a few nanometers. The models are thermodynamically consisted for the interface description; however, they caused severe limitations when performing simulations that are spatially representative of realistic microstructures in which the size of the phases may approach to several microns.

The free energy formulism suggested by Steinbach and his coworkers [25, 26] inspired development of series of models for multicomponent and multiphase systems that have produced quantitative simulations on realistic-length scales. Their original model did not include solute diffusion and only considered pairwise interactions between phases using double-well interpolation functions and Allen–

Cahn dynamics (Eq. 1.2). Later, they improved on their original model with the order parameters that overlap between pairs of phases at the interfaces.

Tiaden et al. [27] included solute diffusion to Steinbach’s multiphase model. The interfacial region was modeled as a mixture of phases, each with different composition, but also with a constant composition ratio. In the model, the concentration c for the whole system is a weighted sum of all c_α , the concentration of c in phase α :

$$c(r, t) = \sum_\alpha \delta_\alpha c_\alpha \quad (1.9)$$

where δ_α is the fraction of phase α . The diffusion of a single component was addressed by partitioning the diffusing species amongst the different phases and solving separate diffusion equations in each phase. In the model, ∇c_α was defined as the driving force for component diffusion and standard Fickian diffusion equations were solved for each phase. The diffusion equations were coupled with phase evolutions driven by a difference in free energy between phases, determined from a local linearization of the phase diagram. This approach with the introduction of partition coefficients enabled the phases with different solute solubilities to be modeled. Although the model was important because it demonstrated the feasibility of modeling diffusional transport in a multiphase model, it was limited to dilute solute concentrations. The use of partitioning coefficients permitted phases to have different equilibrium concentrations; however, the coefficients could not easily be related to free energy density. As a result, incorporation of experimentally measured thermodynamic and kinetic data into the model has been difficult.

Later, the dilute solution limitation of Tiaden’s model was eliminated in the works of Kim et al. [28]. Again, in their work, only single component diffusion was considered; however, they utilized an interpolating function to link the free energy curves. Their model introduced a more sophisticated condition to distribute solute

amongst the phases at diffuse interfaces. A two-phase multicomponent model with an anti-trapping current was also introduced by Kim [29], but it has not been extended to an arbitrary number of phases and components.

The first multicomponent extension of Tiaden's model was given by Gafe et al. [30]. In their model, c_i^α is the concentration of component i in phase α and the fluxes of each component are weighted by the phase fractions δ_α :

$$c_i(r, t) = \sum_{\alpha} \delta_\alpha c_i^\alpha \quad (1.10)$$

The driving force for diffusion was again chosen to be ∇c_i^α , the concentration gradient of component i in the phase α , which is dilute solution approximation. In order for the phases to exchange solute, the model assumes the components are able to instantaneously partition themselves amongst the phases as dictated by partition coefficients. This, again, lead to use of an extrapolation scheme; however, the coefficients were assumed to be a function of composition and temperature and were calculated with Thermo-Calc (a thermodynamic database).

Several other models that build on Steinbach's models have been proposed. Nestler and Wheeler extended the Steinbach multiphase model to study eutectic and peritectic binary alloys [31]. They modeled solute with a nonlinear diffusion equation based on free energy formulation, but they did not include a composition gradient energy and assumed an ideal solution. Later on, Nestler, Garcke, and Stinner proposed a non-isothermal multicomponent extension governed by an entropy functional [32, 33]. A complicated phase barrier function was found to be necessary to prevent the appearance of a foreign third phase at a two-phase boundary. Recently, Eiken et al. [34] developed a multi-component extension to the Tiaden dilute solution multiphase model that removed the dilute solution limitation and allowed for easier inclusion of thermodynamic data. Qin and Wallach developed a two-phase multicomponent solidification model [35] and multiphase, multicomponent model [36] that were linked to the

MTDATA thermodynamic database. Steinbach et al. [37] also reported obtaining thermodynamic data for their multiphase multicomponent model with CALPHAD methods and using NIST mobility database for diffusion data.

From these studies, for the multiphase and multicomponent systems, the free energy functional can be written in simplified form as [38]:

$$F[\{c\}, \{\eta\}] = \int_V \left[f_0 + \sum_{\alpha, \beta=1}^{N-1} \frac{1}{2} \lambda_{\alpha\beta} \nabla \eta_\alpha \cdot \nabla \eta_\beta + \sum_{i, j=1}^{M-1} \frac{1}{2} \kappa_{ij} \nabla c_i \cdot \nabla c_j \right] dv \quad (1.11)$$

where f_0 , the homogenous/bulk free energy, is:

$$f_0 = \sum_{\alpha=1}^N \eta_\alpha f_\alpha([c], T) + \sum_{\alpha \neq \beta}^M W_{\alpha\beta} \eta_\alpha \eta_\beta \quad (1.12)$$

where $W_{\alpha\beta} > 0$ facilitates the mean-field interactions between the phases and is similar to a positive enthalpy of mixing for phases. The free energy curves are the driving force for the separation of phases, and the gradient energy coefficient $\lambda_{\alpha\beta}$ and κ_{ij} penalize gradients that develop, creating surface energy. The κ_{ij} penalizes the phases for differing in composition and $\lambda_{\alpha\beta}$ introduces additional energy that is not captured by the composition gradients. Eq. 1.11 is a first-order approximation of a system with an inhomogeneous distribution of phases and components. As can be seen, it reduces to the Cahn–Hilliard free energy functional (Eq. 1.4) for two-component systems [38].

Phase transformations in solids usually produce coherent microstructures at their early stages. In a coherent microstructure, the lattice planes and directions are continuous across the interfaces and the lattice mismatch between phases/domains is accommodated by elastic displacements. The elastic energy contribution, resulting from elastic displacements, to the total free energy in a phase-field model can be introduced directly by expressing the elastic strain energy as a function of field variables or by including coupling terms between the field

variables and the displacement gradients in the local free energy functions as [39–42]:

$$f_{mechanical} = \frac{1}{2} \sum_{\alpha=1}^N \eta_\alpha [\varepsilon_{ij}^\alpha - \varepsilon_{ij}^{o\alpha}] C_{ijkl}^\alpha [\varepsilon_{kl}^\alpha - \varepsilon_{kl}^{0\alpha}] \quad (1.13)$$

in which $\varepsilon_{ij}^{o\alpha}$ and C_{ijkl}^α are the eigenstrain and elasticity tensor of phases, respectively. If phase transformations involve charged species or electrical or magnetic dipoles, the electrostatic or magnetic energy contributions to the total free energy of a microstructure usually are incorporated using approaches to that of elastic energy. If all four mentioned contributions are considered, the free energy functional (in very generic terms) becomes:

$$F = \int_V [f_{chemical} + f_{gradient} + f_{mechanical} + f_{external}] \times dv \quad (1.14)$$

Finally, Langevin noise terms are usually added to the right-hand side of the evolution equations (Eqs. 1.1 and 1.2) to satisfy the dissipation theorem [43] and to account for thermal fluctuations. This approach works well when the initial state is not too far away from instability with respect to its transformation to the new state by nucleation. The explicit nucleation method, based on the classical nucleation theory [44] and Poisson seeding, has been suggested in Simmons et al. [45, 46]. For this method, the critical size and critical free energy of nucleus formation are determined using the classical nucleation theory, which assumes homogeneous properties within a critical nucleus and a sharp-interface between nucleus and the matrix. Whether or not a stable nucleus is introduced at a given location is determined by comparing a random number with the probability of nucleation. A more sophisticated, but more complex, model was recently given by Heo et al. [47] to treat nucleation in phase-field simulations, which combines diffuse-interface theory with Poisson seeding.

Of course, the overview presented above is a short summary of an immense amount of

literature available on phase-field modeling. However, excellent review articles [48–51] and textbook [52] cover both physical and mathematical aspects in far more in depth than given in here.

It appears that a general procedure for phase-field modeling involves development of a mathematical description for the system free energy that consists of any combination of conserved and non-conserved field variables to fully characterize the microstructure and its evolution. In addition, phase-field modeling must have suitable numerical methods to carry out simulations to elucidate the temporal and spatial evolutions that are relevant to the experimentally observed time and length scales.

References

1. Cahn JW, Hilliard JE (1958) Free energy of a non-uniform system. I. Interfacial energy. *J Chem Phys* 28:258
2. Cahn JW (1961) On spinodal decomposition. *Acta Metall* 9:795
3. Allen SM, Cahn JW (1972) Ground state structures in ordered binary alloys with second neighbor interactions. *Acta Metall* 20:423
4. Allen SM, Cahn JW (1973) A correction to the ground state of fcc binary ordered alloys with first and second neighbor pairwise interactions. *Scr Metall* 7:1261
5. Landau LD, Khalatikow IM (1963) The selected works of L.D. Landau (English translation). Pergamon, Oxford
6. Wheeler AA, Boettger WJ, McFadden GB (1992) Phase-field model for isothermal transformations in binary alloys. *Phys Rev A* 45:7424
7. Wheeler AA, Boettger WJ, McFadden GB (1993) Phase-field model of solute trapping during solidification. *Phys Rev E* 47:1893
8. Wheeler AA, Boettger WJ, McFadden GB (1996) Phase-field model of a eutectic alloy. *Proc R Soc Lond Ser A* 452:495
9. Moelans N (2011) A quantitative and thermodynamically consistent phase-field interpolation function for multi-phase systems. *Acta Mater* 59:1077
10. Langer JS (1986) Models of pattern formation in first-order phase transitions. In: Grinstein G, Mazenko G (eds) Directions in condensed matter physics. World Scientific, Singapore, p 165
11. Fix GJ (1983) Phase field models for free boundary problems. In: Fasano A, Primicerio A (eds) Free boundary problems: theory and applications, vol 2. Pitman, Boston, MA, p 580

12. Collins JB, Levin H (1985) Diffuse interface model of diffusion-limited crystal growth. *Phys Rev B* 31:6119
13. Caginalp G, Fife P (1986) Phase-field methods for interfacial boundaries. *Phys Rev B* 33:7792
14. Caginalp G, Fife P (1987) Higher order phase-field models and detailed anisotropy. *Phys Rev B* 34:4940
15. Caginalp G, Jones J (1995) A derivation and analysis of phase field models of thermal alloys. *Ann Phys* 237:66
16. Caginalp G, Socolovsky EA (1991) Computation of sharp phase boundaries by spreading: the planar and spherically symmetric cases. *J Comput Phys* 95:85
17. Penrose O, Fife PC (1990) Thermodynamically consistent models of phase-field type for the kinetics of phase transitions. *Physica D* 43:44
18. Penrose O, Fife PC (1993) On the relation between the standard phase-field model and a ‘thermodynamically consistent’ phase-field model. *Physica D* 69:107
19. Kobayashi R (1992) Simulations of three dimensional dendrites. In: Kai S (ed) *Pattern formation in complex dissipative system*. World Scientific, Singapore, p 121
20. Kobayashi R (1993) Modeling and numerical simulations of dendritic crystal growth. *Physica D* 63:410
21. Kobayashi R (1994) Numerical approach to three-dimensional dendritic solidification. *Exp Math* 3:59
22. Chen LQ, Yang W (1994) Computer simulation of the domain dynamics of quenched system with a large number of non-conserved order parameters: the grain-growth kinetic. *Phys Rev B* 50:15752
23. Morin B, Elder KR, Sutton M, Grant M (1995) Model of the kinetics of polymorphous crystallization. *Phys Rev Lett* 75:2156
24. Kobayashi R, Warren JA, Carter WC (2000) A continuum model of grain boundaries. *Phys D: Nonlinear Phenom* 140:141
25. Steinbach I, Pezzolla F, Nestler B, Seebelberg M, Prieler R, Schmitz GJ, Rezende JLL (1996) A phase-field concept for multiphase systems. *Physica D* 94:135
26. Steinbach I, Pezzolla F (1999) A generalized field method for multiple transformations using interface fields. *Physica D* 134:385
27. Tiaden J, Nestler B, Diepers HJ, Steinbach I (1998) The multiphase-field model with an integrated concept for modeling solute diffusion. *Physica D* 115:73
28. Kim SG, Kim WT, Suzuki T (1999) Phase-field model for binary alloys. *Phys Rev E* 60:7186
29. Kim SG (2007) A phase-field model with antitrapping current for multicomponent alloys with arbitrary thermodynamic properties. *Acta Mater* 55:4391
30. Grafe U, Bottger B, Tiaden J, Fries SG (2000) Coupling of multicomponent thermodynamic databases to a phase-field model: application to solidification and solid state transformations of super alloys. *Scr Mater* 42:1179
31. Nestler B, Wheeler AA (2000) A multiphase-field model of eutectic and peritectic alloys: numerical simulations of growth structures. *Physica D* 138:114
32. Nestler B, Garcke H, Stinner B (2005) Multicomponent alloy solidification: phase-field modeling and simulations. *Phys Rev E* 71:041609
33. Garcke H, Nestler B, Stinner B (2004) A diffuse interface model for alloys with multiple components and phases. *J SIAM Appl Math* 64:775
34. Eiken J, Boettger B, Steinbach I (2006) Multiphase-field approach for multicomponent alloys with extrapolation scheme for numerical application. *Phys Rev E* 73:0066122
35. Qin RS, Wallach ER (2003) A phase-field model coupled with thermodynamic database. *Acta Mater* 51:6199
36. Qin RS, Wallach ER, Thomson RC (2005) A phase-field model for the solidification of multicomponent and multiphase alloys. *J Crystal Growth* 279:163
37. Steinbach I, Boettger B, Eiken J, Warnken N, Fries SG (2007) Calphad and phase-field modeling: a successful liaison. *J Phase Equilib Diffus* 28:101
38. Cogswell DA, Carter WC (2011) Thermodynamic phase-field model for microstructure with multiple components and phases: the possibility of metastable phases. *Phys Rev E* 83:061602
39. Chen LQ, Hu SY (2004) Phase-field method applied to strain-dominated microstructure evolution during solid-state phase transformations. In: Raabe D, Roters F, Barlat F, Chen LQ (eds) *Continuum scale simulation of engineering materials. Fundamentals-microstructures-process applications*, Wiley
40. Hu SY, Chen LQ (2001) A phase-field model for evolving microstructures with strong elastic inhomogeneity. *Acta Mater* 49:1879
41. Biner SB, Hu SY (2009) Simulation of damage evolution in composites: A phase-field model. *Acta Mater* 57:2088
42. Gururajan MP, Abinandanan TA (2007) Phase-field study of precipitation rafting under uniaxial stress. *Acta Mater* 55:5015
43. Lifshitz EM, Pitaevskii LP (1980) *Statistical physics. Part I, Landau and Lifshitz course of theoretical physics*. Pergamon Press, Oxford
44. Aaronson HI, Lee JK (1975) *Lectures on the theory of phase transformations*. TMS, New York
45. Simmons JP, Shen C, Wang Y (2000) Phase-field modeling of simultaneous nucleation and growth by explicitly incorporating nucleation events. *Scr Mater* 43:935
46. Simmons JP, Wen Y, Shen C, Wang Y (2004) Microstructural development involving nucleation and growth phenomena with the phase-field method. *Mater Sci Eng, A* 365:136
47. Heo TW, Zhang L, Du Q, Chen LQ (2010) Incorporating diffuse interface nuclei phase-field simulation. *Scr Mater* 63:8

48. Chen LQ (2002) Phase-field models for microstructure evolution. *Annu Rev Mater Res* 32:113
49. Loginova IS, Singer HM (2008) The phase-field technique for modeling multi materials. *Rep Prog Phys* 71:106501
50. Stainbach I (2009) Topical review: phase-field models in material science. *Model Simul Mater Sci Eng* 17:073001
51. Wang Y, Li J (2010) Overview: 150, phase-field modeling of defects and deformation. *Acta Mater* 58:1212
52. Provatas N, Elder K (2010) Phase-Field methods in materials science and engineering. Wiley

Introduction to Numerical Solution of Partial Differential Equations

2

2.1 Introduction

Many of the fundamental theories of physics and engineering, including the phase-field models, are expressed by means of systems of partial differential equations, PDEs. A PDE is an equation which contains partial derivatives, such as

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \quad (2.1)$$

in which u is regarded as function of length x and time t . There is no real unified theory for PDEs. They exhibit their own characteristics to express the underlying physical phenomena as accurately as possible. Since PDEs can be hardly solved analytically, their solutions rely on the numerical approaches. A brief of summary of the numerical techniques involving their spatial and temporal discretization is given below. These techniques will be applied to solving the equations of the various phase-field models throughout the book and their detailed descriptions and implementations are given in relevant chapters. There are numerous textbooks also available on the subjects, of which some of them are listed in the references [1–5].

2.2 Basic Numerical Methods

We will start first with numerical techniques used in the spatial discretization of PDEs. As a simple

example, assume, an equation for some vector function $u(x)$, in a domain Ω

$$Lu = f, \quad x \in \Omega \quad (2.2)$$

with boundary condition

$$Bu = 0, \quad x \in \partial\Omega \quad (2.3)$$

where L and B are some linear operators. Then, the problem can be restated as to finding the best approximation to the unknown function $u(x)$. The numerical scheme known as *the method of weighted residuals* is widely applied for this purpose. The method is based on the approximation of the unknown function $u(x)$ by a sum of the so-called *trial functions* $\phi_n(x)$.

$$\tilde{u}(x) = \sum_{n=0}^N a_n \phi_n(x) \quad (2.4)$$

in which $\tilde{u}(x)$ represents the approximate solution of Eq. 2.2 with the unknown coefficients a_n which need to be determined.

The residual R (i.e., error) from this approximation can be calculated as

$$R = L\tilde{u} - f \quad (2.5)$$

Owing to the fact that \tilde{u} is different from the exact solution u , the residual R does not vanish for all $x \in \Omega$. The next step is the selection of the unknown coefficients a_n so that the chosen function approximates the exact solution as closely as

possible. To achieve this, *test or weighting functions* $\psi(x)$ are selected so that the residual function R is minimized over the domain of interest.

$$\int_{\Omega} \psi_n(x) R dx = 0 \quad (2.6)$$

The numerical methods given below and utilized in the next chapters mainly differ in the choice of *trail and test functions* and their minimization approaches. However, for all cases, the trail and tests functions should have a set of desired properties which are: (1) easy to compute, (2) rapid convergence, and (3) completeness.

- *Finite difference methods:*

The domain Ω is divided into small intervals. The unknown function $u(x)$ is approximated by a sequence of local low-order trial polynomials. The polynomials interpolate the solution at given discretization points and the solution results are expressed in the form of weighted sum of values of $u(x)$ at these discretization points.

- *Spectral methods:*

In this case, the trail functions are global smooth functions. The particular selection of the trail functions is usually dictated by the geometry. For periodic structures or intervals, the Fourier series which naturally satisfy the boundary conditions is chosen. On the other hand, Chebyshev or Legendre polynomials are preferred for the non-periodic problems.

- *Finite element methods:*

Similar to finite difference method, the finite element methods have also local character. The domain Ω is divided into small subregions and the trail functions $\phi_n(x)$ are local polynomials of fixed degree and only defined over these subregions.

On the other hand, the choice of test functions $\psi_n(x)$ determines the algorithm for the minimization of Eq. 2.6, also to the solution of Eq. 2.2.

- *Galerkin method:*

For this case, the test functions $\psi_n(x)$ and the trial functions $\phi_n(x)$ are chosen to be the same, both satisfying the boundary condition. Therefore, Eq. 2.2 can be solved as:

$$\int_{\Omega} \psi_n R = \int_{\Omega} \phi_n R = 0 \quad (2.7)$$

By making use of Eq. 2.5

$$\int_{\Omega} \phi_n (L\tilde{u} - f) = 0 \quad \text{and} \quad \int_{\Omega} \phi_n \sum_{n=0}^N a_n \phi_n = \int_{\Omega} \phi_n f \quad (2.8)$$

- *Tau method:*

Again, in this case, the test functions $\psi_n(x)$ are the same as the trail functions $\phi_n(x)$, but the trail functions do no need to satisfy Eq. 2.3, and the boundary conditions are enforced with additional set of equations.

- *Collocation method:*

In this case, the test functions are represented by delta functions, $\psi_n = \delta(x - x_n)$ at specific points x_n termed collocation points, requiring Eq. 2.2 to be satisfied exactly at this locations. In this case, the residual equation,

$$\int_{\Omega} \psi_n R = \int_{\Omega} \delta(x - x_n) R = 0 \quad (2.9)$$

Then, Eq. 2.2 takes the form

$$\sum_k^N a_k L \phi_k(x_n) = f(x_n) \quad (2.10)$$

in which the boundary conditions again enforced with additional set of equations as in Tau method.

Next, we will consider the time discretization of Eq. 2.1, by recasting it to a simpler form for the sake of notation as

$$\frac{u^{n+1} - u^n}{\Delta t} = f(u, t) \quad (2.11)$$

in which Δt is the period between the time steps $n + 1$ and n .

- *The simple forward Euler method*

$$u^{n+1} = u^n + \Delta t f^n \quad (2.12)$$

- *The backward Euler method*

$$u^{n+1} = u^n + \Delta t f^{n+1} \quad (2.13)$$

- *The Crank–Nicholson Method*

$$u^{n+1} = u^n + \frac{1}{2} \Delta t (f^{n+1} + f^n) \quad (2.14)$$

All these three time discretization schemes can be expressed in a unified manner with the so-called θ -methods as:

$$u^{n+1} = u^n + \Delta t [\theta f^{n+1} + (1 - \theta) f^n] \quad (2.15)$$

and as can be seen they are all special cases of Eq. 2.15. Except forward Euler method, the other two are implicit and may require iterative solutions. In addition, the first two Euler methods are first-order accurate and Crank–Nicholson method is second-order accurate.

In addition to θ -methods, the other time discretization schemes include popular second-order Adams–Bashforth method as forward Euler method

$$u^{n+1} = u^n + \frac{1}{2} \Delta t [3f^n - f^{n-1}] \quad (2.16)$$

and for implicit multistep Adams–Moulton third-order method is given as:

$$u^{n+1} = u^n + \frac{1}{12} \Delta t [5f^{n+1} + 8f^n - f^{n-1}] \quad (2.17)$$

On the other hand, Runge–Kutta methods are single-step, but multistage, time discretizations. The classical fourth-order Runge–Kutta method is

$$\begin{aligned} k_1 &= f(u^n, t^n) \\ k_2 &= f\left(u^n + \frac{1}{2}\Delta t k_1, t^n + \frac{1}{2}\Delta t\right) \\ k_3 &= f\left(u^n + \frac{1}{2}\Delta t k_2, t^n + \frac{1}{2}\Delta t\right) \\ k_4 &= f(u^n + \Delta t k_3, t^n + \Delta t) \\ u^{n+1} &= u^n + \frac{1}{6}\Delta t[k_1 + 2k_2 + 2k_3 + k_4] \end{aligned} \quad (2.18)$$

When the storage is not an issue, then the classical fourth-order Runge–Kutta method is commonly used. Otherwise, the low-storage versions are preferred owing to the fact that all Runge–Kutta methods have the same stability properties.

References

1. Larson S, Thomee V (2005) Partial differential equations with numerical methods. Springer text in applied mathematics, vol. 45
2. Morton KW, Mayers AF (2005) Numerical solution of partial differential equations, an introduction (2nd edn). Cambridge University Press
3. Evans G, Blackledge J, Yardley P (1998) Numerical methods for partial differential equations. Springer
4. Mazumdar S (2015) Numerical methods for partial differential equations: finite difference and finite volume methods. Elsevier Academic
5. Dormand JR (1996) Numerical methods for differential equations: a computational approach. CRC Press/Taylor & Francis

The Matlab/Octave programming language was chosen for the codes presented in the book. In the development of the codes, it is assumed that the reader has some degree of experience in computer programming. There are differences in syntax between Matlab/Octave programming and the other traditional programming languages such as Fortran, C, and C++. The readers who are not familiar with Matlab/Octave programming may find useful to consult the extensive documentations that are provided at their websites. In addition, there are plenty of books and online tutorials available, of which some of them are listed in reference section, [1–7].

Matlab is a high-level programming language and interactive environment enabling to perform computationally intensive tasks. The first intended usage of Matlab, also known as Matrix Laboratory, was to make LINPACK and EISPACK codes available to students without learning to use Fortran. The main features of Matlab include high-level language; 2D/3D graphics; mathematical functions for various fields; as well as functions for integrating Matlab-based algorithms with external applications and languages. In

The original version of this chapter was revised. An erratum to this chapter can be found at DOI [10.1007/978-3-319-41196-5_9](https://doi.org/10.1007/978-3-319-41196-5_9)

Electronic Supplementary Material: The online version of this chapter (doi: [10.1007/978-3-319-41196-5_3](https://doi.org/10.1007/978-3-319-41196-5_3)) contains supplementary material, which is available to authorized users.

addition, Matlab performs the numerical linear algebra computations using Basic Linear Algebra Subroutines (BLAS) and Linear Algebra Package (LAPACK). Matlab is licensed by Math Works Inc., the details of its licensing information can be found at www.mathworks.com/products/matlab/.

On the other hand, GNU-Octave is a free software available for everyone to use and redistribute with certain restrictions of GNU licensing. Similar to Matlab, Octave also uses the LAPACK and BLAS libraries. The first release of this package, primarily created by John W. Eaton, along with the contribution of other users, was in January 1993. Octave is written in C++ using the Standard Template Library and uses an interpreter to execute the scripting language. The syntax of Octave is almost identical to Matlab, which allows the codes developed to be portable and to be executable on both platforms. Octave is also available for operating systems Windows, Mac OS, and Linux. The downloading and installation instructions can be found at www.gnu.org/software/octave/.

Just as a side note, there are two other freely available software that are, in some degree, compatible with Matlab and Octave. These are:

FreeMat is another numerical computational package created by Samit Basu with the intent of constructing a free numerical computational package that is compatible with Matlab. More information on the capabilities of FreeMat,

including the downloading and installation instructions, can be found at www.freemat.sourceforge.net.

Scilab is again an open source, cross-platform numerical computational package developed and maintained by INRIA, the French National Research Institution. It was released in 1990. This software is also intended to allow Matlab users to smoothly utilize the package. In order to help in this process, there is a built-in code translator which assists the user in converting their existing Matlab codes into a Scilab code. The information about Scilab can be found at www.scilab.org.

There are many comparative studies, on the compatibility of the syntax between Matlab and Octave, including the availability and capability of the functions, and the computational efficiency as given in Refs. [8, 9]; almost all conclude that Octave is a noteworthy software for scientific and engineering applications and a free alternative to Matlab.

However, there are a few differences in the syntax of Matlab and of Octave. The most noticeable ones are listed below.

1. C-style auto-increment and assignment operator:

Octave supports C-style auto-increment and assignment operators, such as `i++`; `++i`; and `i+=1`. Matlab does not.

2. `fprintf` and `printf`:

Octave supports both `printf` and `fprintf` as a command for printing. Matlab only supports `fprintf`.

3. Whitespace:

Matlab does not allow whitespace before the transpose operator but Octave does:

`[0 1]'` works in Matlab and Octave

`[0 1] '` works only in Octave

4. Line continuation:

Matlab always requires `...` for line continuation. Octave also supports comma symbol, and backslash symbol `\` in addition to `...`

5. Comments:

Matlab uses the percent sign `%` to begin a comment. Octave uses both the hash symbol `#` and the percent sign `%`.

6. Exponentiation:

Octave can use `^` or `**`. Matlab requires `^`

7. String delimiters:

Octave can use `'` or `"`; Matlab requires `'`

8. To end blocks:

Octave can use `end` or specify the block with `endif`, `endfor`. Matlab requires `end`.

9. Logical operator NOT:

For not-equal comparison, Octave can use both `'~='` and `'!='`. Matlab requires `'~='`

All the codes provided in the book execute on both Matlab and Octave without requiring any modifications.

There were two possible alternatives while developing the codes. First one was to make a generic code for each three solutions algorithms and to solve the presented case studies with these three generic codes. The second one was to make stand-alone codes for each case study. For the sake of clarity, the second alternative was selected in the book. For each case study, the relevant program and function files are given in their respective subdirectories in the downloadable file. These directories also include the results obtained from the codes and simulations in the form of short movie files.

A modular approach is chosen for the programs and functions. In the text, their functionality is described first, followed by a variable and array list explaining their contents, followed by listing of the source codes with the line numbers as they appear in the downloadable file and followed by the annotations of line numbers in the source codes.

Particular attention was devoted to computational efficiency and clarity during development of the codes, with the lattermost often being given a higher preference. All codes in the text are presented in two formats. The ones in long-hand format are not optimized for Matlab/Octave. They are primarily intended to bring transparency and more closely follow the each phase-field model formulism without creating confusion with the short syntax of Matlab/Octave. It is hoped that this approach would be helpful for better understanding of the program structures, for both experienced and inexperienced users; in addition, they may also serve as

template for their implementation in other traditional computer programming languages. Since these longhand format versions do not take advantage of the highly optimized nature of Matlab/Octave, they require significantly longer execution times than their optimized counterparts. Also, there are still plenty of improvements that can be made to the optimized versions, so their performance can be increased further.

In the solution of phase-field models, as any grid-based numerical methods, the quality of the results increases with increasing spatial and temporal resolution. Of course, this directly translates into higher computational demand in terms of both memory and processor requirements. In spite of every effort that is made for the codes given in the text to produce acceptable results with minimum spatial and temporal resolutions, still they may require significant execution times, depending on the reader's computational resources. This is also the reason for avoiding to output the graphical results during the solution process.

Although both Matlab and Octave have impressive graphic routines, the graphical presentations of the results were made by using Paraview and Gnuplot in the text. Paraview has more powerful features for visualization and spatial and temporal analysis of the data. Again, Paraview is an open-source, multi-platform data analysis and visualization application. The details of downloading and installing instructions can be found at www.paraview.org. Gnuplot is also freely distributed, command-line driven graphing utility for Windows, Mac-OS, and Linux operating systems. The information about Gnuplot can be found at www.gnuplot.info. So, the installation of these two applications, along with the downloadable file, is also highly recommended.

Password protected, downloadable zip files, containing the source codes and the movie files resulting from the simulations, are available. Readers who have purchased the eBook or print

versions of this book can visit the Springer Link webpage for the book at the following link:

<https://link.springer.com/book/10.1007%2F978-3-319-41196-5>

On this page, there will be downloadable zip files for Chapters 4, 5, 6, and 7, that contain the necessary files to execute the main programs, per the case studies in those individual chapters, with movie files in the AVI format resulting from these simulations. There will also be additional materials related to the Appendices A, B, and C within the downloadable zip files for Chapters 4, 5, and 6.

The names of the downloadable zip files for the individual chapters and the required passwords to unzip them are given in the table below:

Chapters	Filenames	Passwords
Chapter-4	prog_pf_ch4.zip	sbb#4RFV
Chapter-5	prog_pf_ch5.zip	sbb#5TGB
Chapter-6	prog_pf_ch6.zip	sbb#6YHN
Chapter-7	prog_pf_ch7.zip	sbb#7UJM

References

- Quateroni A, Saleri F, Gervasio P (2014) Scientific computing with Matlab and Octave, 4th edn. Springer, Berlin
- Davista TA (2010) Matlab Premier, 8th edn. CRC Press
- Gilat A (2008) Matlab: an introduction with applications, 3rd edn. Wiley, New York
- Attaway S (2013) Matlab: a practical introduction to programming and problem solving, 3rd edn. Butterworth-Heinemann
- Moore H (2008) Matlab: for engineers, 2nd edn. Prentice Hall, Piscataway, NJ
- Hansen JS (2011) GNU-Octave beginner's guide. Packt Publishing
- Rogel-Salazer J (2014) Essential Matlab and Octave. CRC Press, Boca Raton, FL
- Leros AP, Andreatos A, Zgarianos A (2010) Matlab-Octave science and engineering benchmarking and comparison. *Latest Trends Comput* 2:746
- Sharma N, Gobbert MK (2010) A comparative evaluation of Matlab, Octave, FreeMat and Scilab for research and teaching. Technical report HPCF-2010-7 www.umbc.edu/hpcf/publications

Solving Phase-Field Models with Finite Difference Algorithms

4.1 Introduction

Finite difference algorithms offer a more direct approach to the numerical solution of partial differential equations than any other method. Finite difference algorithms are based on the replacement of each derivative by a difference quotient. Finite difference algorithms are simple to code, economic to compute, and easy to parallelize for the distributed computing environments. However, they also have disadvantages in terms of accuracy and imposing complex boundary conditions. For better understanding of the method, the solution of one-dimensional transient heat conduction is given as an example together with the source code in this section.

Numerous textbooks and lecture notes are available on solving ordinary and partial differential equations, ODEs and PDEs, with finite difference methods, and a few of them are also listed in the reference section, [1–6].

For a continuous and relatively smooth function, u , on a one-dimensional grid with grid spacing h at the grid points, the following equations are used for finite difference algorithm.

The original version of this chapter was revised. An erratum to this chapter can be found at DOI [10.1007/978-3-319-41196-5_9](https://doi.org/10.1007/978-3-319-41196-5_9)

The backward difference:

$$(\Delta u)_i^- = \frac{u_i - u_{i-1}}{h} \quad (4.1a)$$

The forward difference:

$$(\Delta u)_i^+ = \frac{u_{i+1} - u_i}{h} \quad (4.1b)$$

The centered difference:

$$(\Delta u)_i^\pm = \frac{u_{i+1} - u_{i-1}}{2h} \quad (4.1c)$$

The centered second difference:

$$(\Delta^2 u)_i = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} \quad (4.1d)$$

By using Taylor expansion, it is easy to prove that the error norms are as follows:

$$\begin{aligned} (\Delta u)_i^- - u'(x_i) &= O(h) \\ (\Delta u)_i^+ - u'(x_i) &= O(h) \\ (\Delta u)_i^\pm - u'(x_i) &= O(h^2) \\ (\Delta^2 u)_i - u''(x_i) &= O(h^2) \end{aligned} \quad (4.2)$$

These difference formulas, particularly the centered second difference to approximate the Laplace operator, will be used in our numerical solutions of the phase-field models. For a two-dimensional grid, at inside grid points (x_i, y_j) , the Laplace operator becomes:

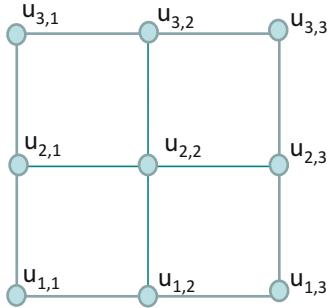


Fig. 4.1 A schematic node ordering representation for a two-dimensional finite difference grid

$$\begin{aligned} (\nabla^2 u)_{i,j} &= (\Delta_{xx}^2 u)_{i,j} + (\Delta_{yy}^2 u)_{i,j} \\ &= \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h_x^2} \\ &\quad + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h_y^2} \end{aligned} \quad (4.3)$$

This approximation is called a five-point stencil because it only involves five grid points, which can be inferred from the equation and from Fig. 4.1. The nine-point stencil reduces the discretization error to $O(h^4)$ and leads to usage of a larger h with a smaller error than for the five-point stencil; however, this occurs with increasing computational cost [1, 2]. Therefore, the five-point stencil is usually the choice for spatial discretization for solution of phase-field models.

If the grid spacing in both the x and y spatial directions are equal to each other, $h = h_x = h_y$, then Eq. 4.3 becomes:

$$(\nabla^2 u)_{i,j} = \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}}{h^2} \quad (4.4)$$

For the equally spaced nine grid points shown in Fig. 4.1, without considering any boundary conditions, Eq. 4.4 can be obtained from the matrix-vector product as:

$$(\nabla^2 u)_{i,j} = \frac{1}{h^2} \begin{pmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{pmatrix} \begin{pmatrix} u_{1,1} \\ u_{2,1} \\ u_{3,1} \\ u_{1,2} \\ u_{2,2} \\ u_{3,2} \\ u_{1,3} \\ u_{2,3} \\ u_{3,3} \end{pmatrix} \quad (4.5)$$

Furthermore, as a general case, the matrix in Eq. 4.5 can be partitioned as:

$$M = \begin{pmatrix} A & I & O & O & O \\ I & A & I & \cdots & O \\ O & I & A & O & O \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ O & O & O & \cdots & I & A \end{pmatrix} \quad (4.6)$$

where I is the 3×3 identity matrix, O is the 3×3 zero matrix, and A is as follows:

$$A = \begin{pmatrix} -4 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & -4 \end{pmatrix} \quad (4.7)$$

It should be noted that matrix M has the following unique properties:

- It is sparse, with most five elements per row are non-zero.
- It is block tri-diagonal, with tri-diagonal and diagonal blocks.
- It is symmetric.

- It is diagonally dominant.
- Its diagonal elements are negative and others are nonnegative.
- It is negative definite.

4.2 One-Dimensional Transient Heat Conduction: A Solution with Finite Difference Algorithm

One-dimensional heat conduction equation, without heat generating sources, can be expressed as,

$$\rho c_p \frac{\partial T}{\partial t} = \frac{\partial}{\partial x} \left(\lambda \frac{\partial T}{\partial x} \right) \quad (4.8)$$

where ρ is the density, c_p is the heat capacity, λ is the thermal conductivity, T is the temperature, t is the time, and x is the distance. If the thermal conductivity, density, and heat capacity are constant over time and in the domain, the equation simplifies to

$$\frac{\partial T}{\partial t} = \mu \frac{\partial^2 T}{\partial x^2}, \quad \text{and} \quad \mu = \frac{\lambda}{\rho c_p} \quad (4.9)$$

By utilizing the forward difference, Eq. 4.1a, for the time derivative and the centered second finite difference, Eq. 4.1d, for the spatial derivative, Eq. 4.9 takes the form,

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} = \mu \left(\frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{\Delta x^2} \right) \quad (4.10)$$

in which Δt is the time between the time steps $n+1$ and n , and i is the grid number of finite difference stencil. With forward Euler time marching, the temperature at grid point i for the time step $n+1$ results in:

$$T_i^{n+1} = T_i^n + \mu \Delta t \left(\frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{\Delta x^2} \right) \quad (4.11)$$

Because the temperature field is known at the time step n , the explicit Euler time marching scheme enables the direct solution of Eq. 4.11.

The program, *heat_1d_fd.m* given below is used to generate the results by solving Eq. 4.11. The one-dimensional domain was discretized with 128 grid points with grid spacing of 1 mm in the program. The value of $\mu = 1.0 \text{ mm}^2 \text{ s}^{-1}$ is assumed. The time increment between the time steps Δt was taken as 0.2 s. The initial temperature at 40 mm region in the center was 1 °C and zero at the other places including two ends which constitute the boundary condition of the problem. The periodically produced graphical results during the solution are shown in Fig. 4.2. As can be seen, while the initial temperature at the center region decays overtime, the temperature rises at the other regions resulting from the thermal conductivity and heat flux.

The results seen in Fig. 4.2 agree well with the analytical solutions. However, this is true for only range of Δx and Δt values. The use of larger values results in instabilities, oscillations, and unacceptable results owing to the coarse spatial and temporal discretization.

4.3 Source Codes

Program

heat_1d_fd.m

This program solves one-dimensional transient heat conductivity problem with finite difference algorithm and explicit Euler time marching scheme.

Listing:

```

1 %%%%%%%%%%%%%%%%
2 % 1D transient heat conduction %
3 % space discretization: finite- %
4 % difference time integration: %
5 % Explicit Euler %
6 %%%%%%
7 %%-- get initial wall time:
8 time0=clock();
9
10 %%-- Simulation cell parameters:
11
12 Nx=128;

```

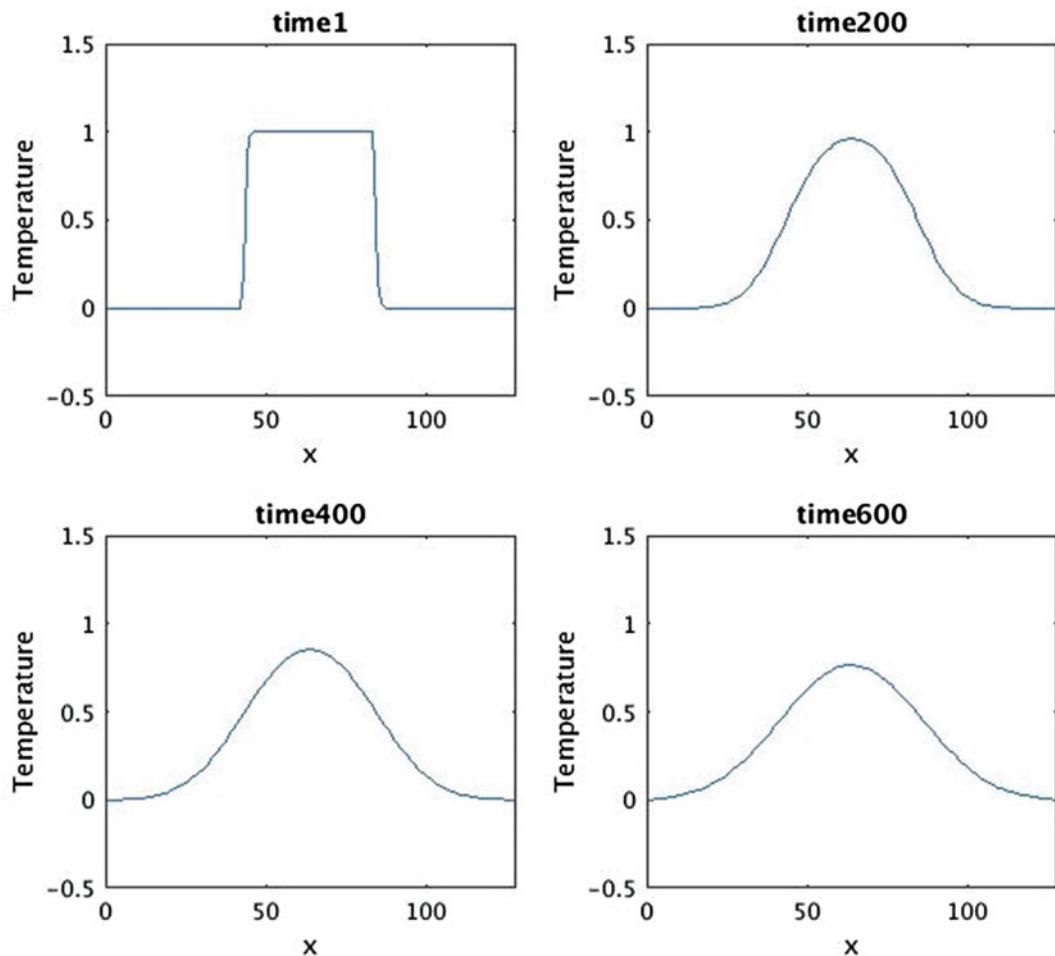


Fig. 4.2 Graphical outputs from the program *heat_1d_ft.m* during the solution of one-dimensional heat conductivity problem

```

13 dx = 1.0;
14
15 %--- Time integration parameters:
16
17 nstep = 600;
18 nprint = 200;
19 dtime = 0.2;
20
21 %-- Initialize temperature field &
22 %-- grid:
23 for i=1:Nx
24 u0(i) = 0.0;
25 x(i)=i*dx;
26
27 if((i >= 44) && (i <= 84))
28 u0(i)=1.0;
29 end
30 end
31
32 %--
33 %-- Evolve temperature field:
34 %--
35
36 ncount=0;
37 for istep=1:nstep
38
39 for i=2:Nx-1

```

```

40 u0(i) = u0(i) + dtime*(u0(i+1)-2.0*u0
   (i)+u0(i-1)) ...
41 / (dx*dx);
42 end
43
44 %-- Display results:
45
46 if((mod(istep,nprint) == 0) || (istep
   == 1))
47
48 ncount=ncount+1;
49 subplot(2,2,ncount);
50 plot(x,u0);
51 time=sprintf('%d',istep);
52 title(['time' time]);
53 axis([0 Nx -0.5 1.5]);
54 xlabel('x');
55 ylabel('Temperature');
56
57 end %if
58 end %istep
59
60 compute_time=etime(clock(),time0);
61 fprintf('Compute Time: %5d\n',
   compute_time);

```

Line numbers:

8:	Get wall clock time at the beginning of execution.
10–14:	Simulation cell parameters.
12:	Number of grid points in the x -direction.
13:	Grid spacing between two grid points in the x -direction.
15–20	Time integration parameters.
17:	Number of integration steps.
18:	Print frequency to output the results.
19:	Time increment.
21–31:	Initialize the initial temperature field and grid point coordinates.
33–58:	Evolve temperature field.
39–42:	For each grid points, evaluate Eq. 4.11.
46–57:	If print frequency is reached, output the results in the graphical format.
60–61:	Compute the execution time and print it.

References

1. Grossman C, Roos HG, Stynes M (2000) Numerical treatment of partial differential equations. Springer, Berlin
2. LeVeque RJ (2007) Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems. SIAM, Philadelphia
3. Thomas JW (1995) Numerical partial differential equations: finite difference method. Springer, New York
4. Ozisik N (1994) Finite difference methods in heat transfer. CRC Press, Boca Raton
5. Arnold DN (2011) Lecture notes on numerical analysis of partial differential equations. School of Mathematics, University of Minnesota
6. Chen L (2013) Finite difference methods. Department of Mathematics, University of California, Irvine

4.4 Case Study-I

Simulation of the spinodal decomposition of a binary alloy with explicit Euler finite-difference algorithm

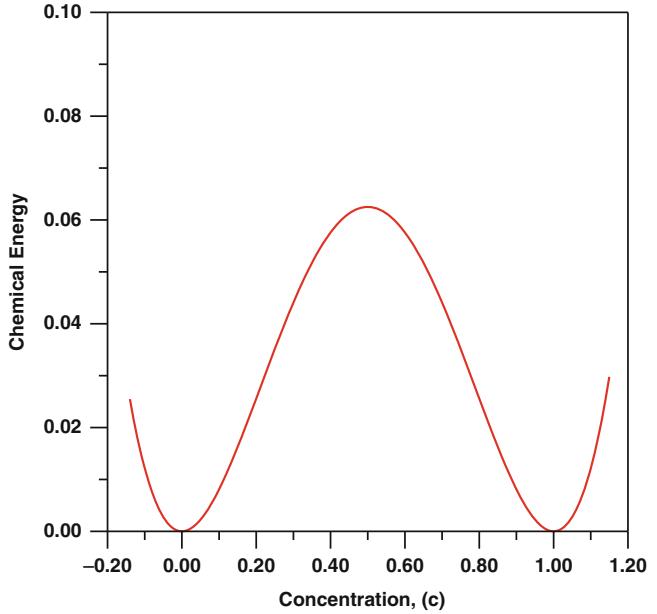
Objectives:

The objective of this case study is to demonstrate a numerical implementation of the finite difference algorithm for the solution of conserved Cahn–Hilliard equation in phase-field modeling.

4.4.1 Background

The Cahn–Hilliard model (Chap. 1, Eq. 1.1) was originated from a phase-separation model called the spinodal decomposition in a binary alloy, which is one of the phase transformations in solid state. Spinodal decomposition is a mechanism by which a solid solution can separate into distinctly different phases that have different compositions and physical properties. It is different from the classical nucleation-driven mechanism and takes place throughout the material, not just at discrete nucleation sites. Because there is no thermodynamic barrier for nucleation reactions, decomposition is only driven by the diffusion process. Owing to this simplicity, apart from its extensive applications in material science, the model is also used in other fields such as image inpainting [1], multiphase fluid flow [2–5], phase separation [6], flow visualization [7], the formation of the quantum dots [8], Taylor flow in mini/micro channels [9], pore migration in a temperature gradient [10], and tumor growth [11, 12].

Fig. 4.3 Variation of chemical energy as a function of concentration c . $A = 1$ in Eq. 4.13



4.4.2 Phase-Field Model

In this example, the total free energy appears in Chap. 1, Eq. 1.1, in its simplest form, taken as:

$$F = \int_V \left[f(c) + \frac{1}{2} \kappa (\nabla c)^2 \right] dv \quad (4.12)$$

where κ is the gradient energy coefficient and $f(c)$ is the chemical/bulk energy represented by:

$$f(c) = Ac^2(1 - c)^2 \quad (4.13)$$

which is a simple phenomenological double-well potential. A is a positive constant and controls the magnitude of the energy barrier between two equilibrium phases, which take the concentration values of $c = 0.0$ and $c = 1.0$, as can be seen from Fig. 4.3.

The other popular choices for generic chemical/bulk free energy forms are:

$$f(c) = \frac{1}{4}A(1 - c^2)^2 \quad \text{or} \quad f(c) = 4A\left(-\frac{1}{2}c^2 + \frac{1}{4}c^4\right) \quad (4.14)$$

However, for both these double-well potentials, the equilibrium concentration of phases occurs at $c = -1.0$ and $c = 1.0$.

4.4.3 Numerical Implementation

In this section, a fully discrete finite difference method to solve the Cahn–Hilliard equation is described. The problem is discretized in a two-dimensional domain; however, its extension to three dimensions is straightforward. Let N_x and N_y be the number of grid points and dx and dy be the spacing between the grid points in the x and y spatial directions, respectively. The periodic boundary condition is implemented by requiring the following:

$$\begin{aligned} c_{0,j} &= c_{Nx,j}, & c_{Nx+1,j} &= c_{1,j}, \\ c_{i,0} &= c_{i,Ny}, & c_{i,Ny+1} &= c_{i,1} \end{aligned} \quad (4.15)$$

By taking the functional derivative of the free energy functional (Eqs. 4.12 and 4.13) and using the Explicit Euler time integration scheme, the time evolution of the Cahn–Hilliard equation (Chap. 1, Eq. 1.1) can be expressed as:

$$\frac{c_{ij}^{n+1} - c_{ij}^n}{\Delta t} = \nabla^2 M \left(\frac{\delta F_{ij}}{\delta c} \right)^n \quad (4.16)$$

where M is the mobility, Δt is the time step size, and n is the current time step. $\frac{\delta F}{\delta c}$ takes the value of

$$\left(\frac{\delta F_{ij}}{\delta c}\right)^n = \mu(c_{ij}^n) - \kappa \nabla^2 c_{ij}^n \quad (4.17)$$

and $\mu(c_{ij}^n)$ is the functional derivative of chemical/bulk energy (Eq. 4.13):

$$\mu(c_{ij}^n) = A \left(2c_{ij}^n (1 - c_{ij}^n)^2 + 2(c_{ij}^n)^2 (1 - c_{ij}^n) \right) \quad (4.18)$$

In the algorithm, to approximate the discrete Laplace operator appearing in Eqs. 4.16 and 4.17, the five-point stencil described earlier will be used.

In order to monitor the variation of the free energy during the course of evolution, a discrete energy functional is defined as:

$$F_D = \sum_i^{N_x} \sum_j^{N_y} f(c_{ij}) + \frac{\kappa}{2} \left(\sum_i^{N_x-1} \sum_j^{N_y-1} (c_{i+1,j} - c_{ij})^2 + \sum_i^{N_x-1} \sum_j^{N_y-1} (c_{i,j+1} - c_{ij})^2 \right) \quad (4.19)$$

and it is calculated periodically during the solution.

4.4.4 Results and Discussion

The simulation was carried out for a square simulation cell having $N_x = 64$ and $N_y = 64$ and with $dx = dy = 1.0$. All of the parameters used in the simulation are nondimensional. The non-dimensional characteristic length L' , energy F' , and time t' are defined as:

$$L' = \left(\frac{\kappa}{A}\right)^{1/2}, \quad F' = AL^3 \quad \text{and} \quad t' = \frac{L'^2 (c_p^e - c_m^e)^2}{MF'} \quad (4.20)$$

where k is the gradient energy coefficients, A is the energy barrier in the chemical free energy functional in Eq. 4.13, M is the mobility coefficient in Eq. 4.16, and c_p^e and c_m^e are the equilibrium concentration of the phases (see Fig. 4.3).

The average composition of the alloy was chosen as $c_0 = 0.4$. This initial composition was modulated with the introduction of a random noise term of 0.02 to account for the thermal fluctuations (see function *micro_ch_pre.m* and Chap. 1). The nondimensional time increment per time step was set as $\Delta t = 0.01$ in Euler time integration (Eq. 4.16) and simulation was carried out up to 20,000 time steps.

Time evolution of the microstructure due to phase separation is summarized in Figs. 4.4 and 4.5. The time values quoted in the figure are in nondimensional form. As can be seen at time 20, the microstructure is relatively fine and contains a large number of precipitates. As time progresses, coarsening of the second phase through migration of the phase boundaries, dissolution, merging, and breakup can be easily inferred from the figure. As can be seen, the growth mainly occurs with Ostwald ripening process in which the small precipitates dissolves and absorbed by the larger ones, and as a result the number of the precipitates becomes smaller with time. Also, these figures clearly illustrate the superiority of the phase-field models over the sharp interface models for handling this very complex and collective evolution of interfaces. An animation file summarizing this simulation is given in subdirectory, *case_study_1*, of the downloadable file. Of course, this phase separation process is a result of the reduction in total free energy as shown in Fig. 4.5. Some of the microstructures at specific times are shown as insets in the figure. As can be seen in the figure, owing to significant reduction in the amount of interfaces in the microstructure, the coarsening becomes quite slow through the later stages; this is reflected in the rate of change in the total free energy.

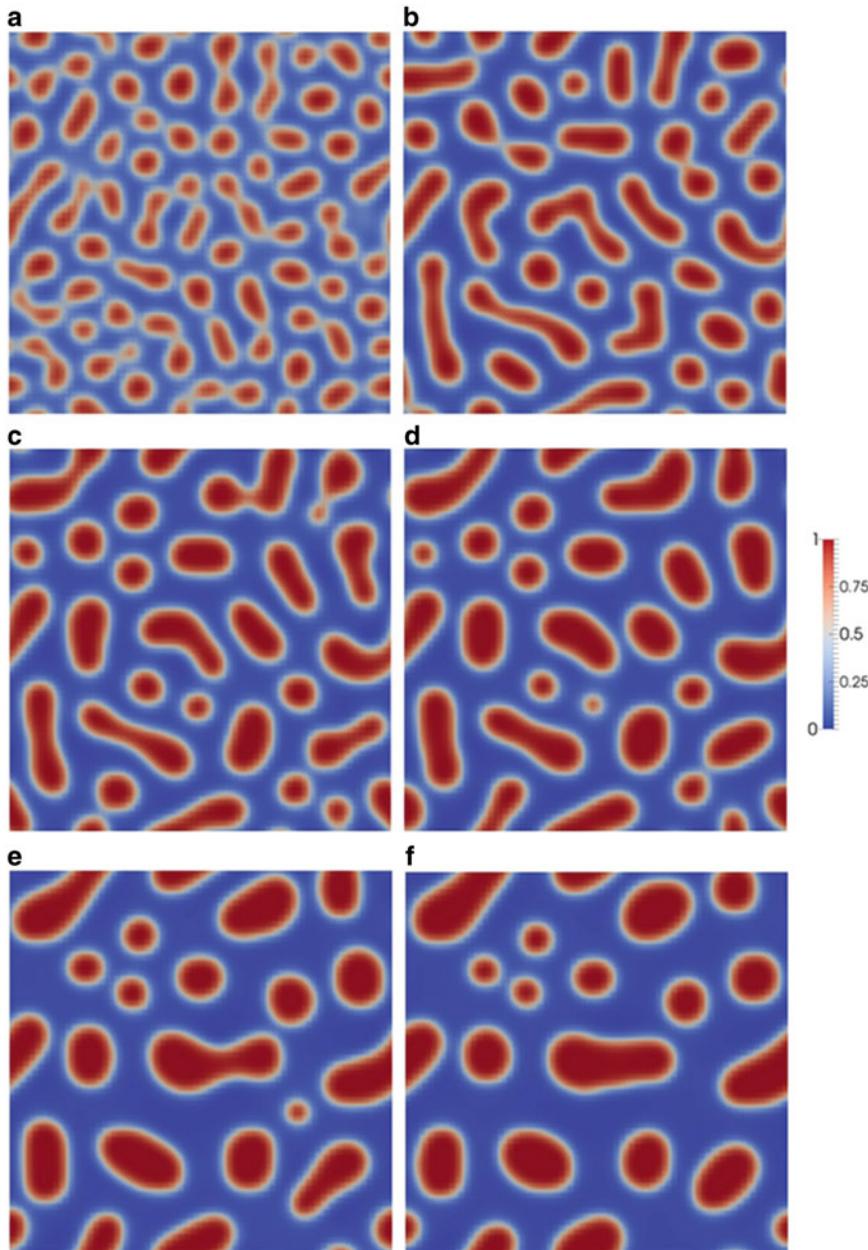


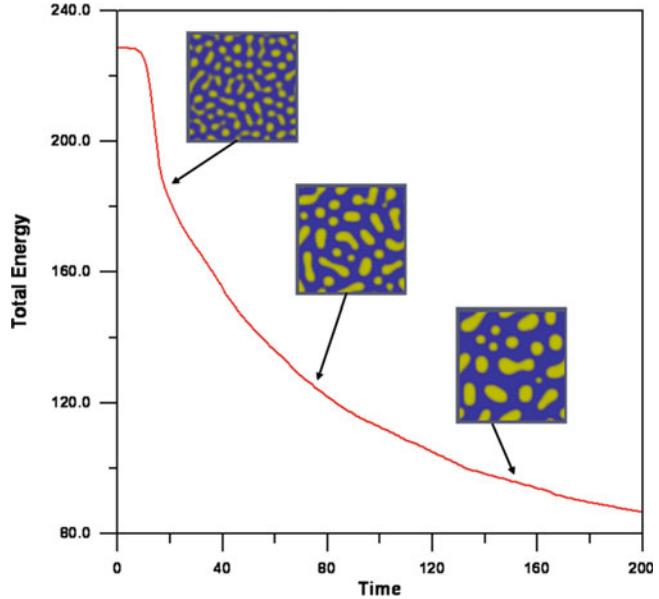
Fig. 4.4 Time evolution of microstructure as a result of phase separation. The nondimensional times are: (a) 20, (b) 50, (c) 75, (d) 100, (e) 150, and (f) 200

Finally, the explicit scheme is very simple to implement; however, it is less efficient owing to the severe time step restriction. To remedy this drawback in the solution of the Cahn–Hilliard equation

with finite difference algorithms, other time integration schemes have been suggested in which Eqs. 4.16 and 4.17 take the following forms:

Implicit Euler scheme [13]:

Fig. 4.5 Time variation of the discrete free energy (Eq. 4.19); the insets show the microstructures at the times indicated by the arrows



$$\frac{c_{ij}^{n+1} - c_{ij}^n}{\Delta t} = \nabla^2 M \left(\frac{\delta F}{\delta c} \right)^{n+1} \quad (4.21)$$

$$\frac{c_{ij}^{n+1} - c_{ij}^n}{\Delta t} = \nabla^2 M \left(\frac{\delta F}{\delta c} \right)^{n+1} \quad (4.27)$$

$$\left(\frac{\delta F}{\delta c} \right)^{n+1} = \mu(c_{ij}^{n+1}) - \kappa \nabla^2 c_{ij}^{n+1} \quad (4.22)$$

$$\left(\frac{\delta F}{\delta c} \right)^{n+1} = \mu(c_{ij}^n) - 2c_{ij}^n + 2c_{ij}^{n+1} - \kappa \nabla^2 c_{ij}^{n+1} \quad (4.28)$$

Crank–Nicholson scheme [2, 14, 15]:

$$\frac{c_{ij}^{n+1} - c_{ij}^n}{\Delta t} = \frac{1}{2} \nabla^2 M \left(\left(\frac{\delta F}{\delta c} \right)^{n+1} - \left(\frac{\delta F}{\delta c} \right)^n \right) \quad (4.23)$$

$$\left(\frac{\delta F}{\delta c} \right)^{n+1} = \mu(c_{ij}^{n+1}) - \kappa \nabla^2 c_{ij}^{n+1} \quad (4.24)$$

Semi-implicit Euler scheme [16–19]:

$$\frac{c_{ij}^{n+1} - c_{ij}^n}{\Delta t} = \nabla^2 M \left(\frac{\delta F}{\delta c} \right)^{n+1} \quad (4.25)$$

$$\left(\frac{\delta F}{\delta c} \right)^{n+1} = \mu(c_{ij}^n) - \kappa \nabla^2 c_{ij}^{n+1} \quad (4.26)$$

Linearly stabilized splitting scheme [1, 20]

Non-linearly stabilized splitting scheme [21, 22]:

$$\frac{c_{ij}^{n+1} - c_{ij}^n}{\Delta t} = \nabla^2 M \left(\frac{\delta F}{\delta c} \right)^{n+1} \quad (4.29)$$

$$\left(\frac{\delta F}{\delta c} \right)^{n+1} = \mu(c_{ij}^n) - c_{ij}^n + c_{ij}^{n+1} - \kappa \nabla^2 c_{ij}^{n+1} \quad (4.30)$$

Some details about computer implementation of the above integration schemes can be found in the listed references. A critical comparison of these schemes regarding their stability and efficiency is recently summarized in Kim et al. [23].

This problem will also be solved by utilizing the spectral methods discussed in Chap. 5 and with the finite element method discussed in Chap. 6, using identical solution parameters.

4.4.5 Source Codes

Two programs are provided in this section. The first one, *fd_ch_v1.m*, is in longhand format and may serve for easier understanding of the numerical implementation and the program structure. The second version, *fd_ch_v2.m*, is the optimized version for Matlab/Octave and therefore reduces the execution time significantly. The programs are given first, followed by the description of the functions that are called in the programs.

Program

fd_ch_v1.m

This program solves the Cahn–Hilliard phase-field equation with finite difference algorithm using five-point stencil. The time integration is carried out using explicit Euler scheme. Program is in longhand format and not optimized for Matlab/Octave.

The program makes calls to the following functions:

- **micro_ch_pre.m**
- **calculate_energ.m**
- **free_energ_ch_v1.m**
- **write_vtk_grid_values.m**

Listing

```

1    %%%%%%%%%%%%%%%%
2    %                         %
3    % FINITE-DIFFIRENCE PHASE-FIELD   %
4    %           CODE FOR SOLVING        %
5    %           CAHN-HILLIARD EQUATION %
6    %%%%%%%%%%%%%%%%
7
8    %%= get intial wall time:
9    time0=clock();
10   format long;
11
12  out2 = fopen('time_energ.out','w')
13
14  %%-- Simulation cell parameters:
15
16  Nx = 64;
17  Ny = 64;
18  NxNy=Nx*Ny;
19  dx = 1.0;
20  dy = 1.0;
21
22  %--- Time integration parameters:
23
24  nstep = 10000;
25  nprint= 50;
26  dtime = 1.0e-2;
27  ttime = 0.0;
28
29  %--- Material specific Parameters:
30
31  c0 = 0.40;
32  mobility = 1.0;
33  grad_coef= 0.5;
34
35  %
36  %---prepare microstructure:
37  %
38
39  iflag =1;
40  [con] = micro_ch_pre(Nx,Ny,c0,
41  iflag);
42
43  %
44  %--- Evolve
45
46  for istep =1:nstep
47
48  ttime = ttime +dtime;
49
50  for i=1:Nx
51  for j=1:Ny
52
53  jp=j+1;
54  jm=j-1;
55
56  ip=i+1;
57  im=i-1;
58
59  jp=j+1;
60  jm=j-1;
61
62  ip=i+1;
63  im=i-1;
64

```

```

65 if(im == 0)                                112 im=i-1;
66 im=Nx;                                     113
67 end                                         114 if(im == 0)
68 if(ip == (Nx+1))                           115 im=Nx;
69 ip=1;                                       116 end
70 end                                         117 if(ip == (Nx+1))
71 if(jm == 0)                                118 ip=1;
72 jm = Ny;                                    119 end
73 end                                         120
74 if(jp == (Ny+1))                           121 if(jm == 0)
75 jp=1;                                       122 jm = Ny;
76 end                                         123 end
77 hne=con(ip,j);                            124
78 hnw=con(im,j);                            125 if(jp == (Ny+1))
79 hns=con(i,jm);                            126 jp=1;
80 hnn=con(i,jp);                            127 end
81 hnc=con(i,j);                            128
82 lap_con(i,j) = (hnw + hne + hns + hnn 129 hne=dummy(ip,j);
83 -4.0*hnc) / (dx*dy);                     130 hnw=dummy(im,j);
84 %--- derivative of free energy:          131 hns=dummy(i,jm);
85 [dfdcon]=free_energ_ch_v1                132 hnn=dummy(i,jp);
86 (i,j,con);                                133 hnc=dummy(i,j);
87 dummy(i,j) = dfdcon -                    134
88 grad_coef*lap_con(i,j);                  135 lap_dummy(i,j) = (hnw + hne + hns
89 end %for j                                136 + hnn - 4.0*hnc) / (dx*dy);
90 end %for i                                137 %-- time integration:
91 %--                                         138
92 for i=1:Nx                                139 con(i,j) = con(i,j) + dtimes*
93 for j=1:Ny                                140 mobility*lap_dummy(i,j);
94 jp=j+1;                                     141 %-- for small deviations:
95 jm=j-1;                                     142
96 ip=i+1;                                     143 if(con(i,j) >= 0.9999);
97 im=i-1;                                     144 con(i,j)=0.9999;
98 %--                                         145 end
99 for i=1:Nx                                146 if(con(i,j) < 0.00001);
100 for j=1:Ny                               147 con(i,j)=0.00001;
101 jp=j+1;                                     148 end
102 jm=j-1;                                     149
103 ip=i+1;                                     150 end %j
104 im=i-1;                                     151 end %i
105 %---- print results                      152
106 if((mod(istep,nprint) == 0) ||           153
107 (istep == 1) )                           154
108 fprintf('done step: %5d\n',istep);        155
109 ip=i+1;                                     156
110 im=i-1;                                     157
111 ip=i+1;                                     158

```

```

159 %fname1 = sprintf('time_%d.out',
160 istep);
161
162 %for i=1:Nx
163 %for j=1:Ny
164 fprintf(out1,'%5d %5d %14.6e\n',i,
165 j,con(i,j));
166 %end
167 %end
168 fclose(out1);
169
170 %--- write vtk file:
171
172 %-- calculate total energy
173
174 [energ] = calculate_energ(Nx,Ny,
175 con,grad_coef);
176 fprintf(out2,'%14.6e %14.6e\n',
177 ttime,energ);
178 write_vtk_grid_values(Nx,Ny,dx,dy,
179 istep,con);
180 end %if
181 end %istep
182
183 %--- calculate compute time:
184
185 compute_time = etime(clock(),
186 time0);
187 fprintf('Compute Time: %10d\n',
188 compute_time);

```

<i>Line numbers:</i>	
8–12:	Get wall clock time beginning of the execution and open an output file for writing total bulk energy values.
14–21:	Simulation cell parameters.
16:	Number of grid points in the <i>x</i> -direction.
17:	Number of grid points in the <i>y</i> -direction.
18:	Total number of grid points in the simulation cell.
19:	Grid spacing between two grid points in the <i>x</i> -direction

20:	Grid spacing between two grid points in the <i>y</i> -direction.
22–28:	Time integration parameters.
24:	Number of time integration steps.
25:	Output frequency to write the results to file.
26:	Time increment for numerical integration.
27:	Total time.
29–34:	Material-specific parameters.
31:	Average composition.
32:	Value of mobility coefficient.
33:	Value of gradient energy coefficient.
36–41:	Initialize the concentration field with random modulation.
39:	Set iflag = 1 for un-optimized mode, iflag = 2 for optimized mode.
46–182:	Time evolution of concentration field.
48:	Update the total time.
50–96:	Calculate the Laplacian of the concentration with five-point finite difference stencil.
86:	Calculate the Laplacian of concentration at the current grid point.
90:	Calculate the derivative of free energy.
92:	Evaluate the terms in Eq. 4.17, and accumulate them in array dummy(Nx,Ny) for each grid point in the simulation cell.
99–152:	Time integration of Eq. 4.16.
135:	Calculate the Laplacian of the terms inside the parenthesis in Eq. 4.16.
139:	Explicit Euler time integration of concentration field, Eq. 4.16.
141–149:	If there are small variations, set the max and min values to the limits.
153–181:	If print frequency is reached, output the results to file.
159–169:	Open an output file and print results. These lines are commented out, but they can be changed, including the format of the output file.
172–176:	Calculate the total bulk energy and print the result to <i>time_energy.out</i> file for 2D plots.
178:	Write the results in vtk format for contour plots to be viewed by using Paraview.
184–187:	Calculate the total execution time and print it to screen.

Program

fd_ch_v2.m

This program solves the Cahn–Hilliard phase-field equation with finite difference algorithm using five-point stencil. The time integration is

carried out using explicit Euler scheme. Program is optimized for Matlab/Octave.

The program makes calls to the following functions:

- **micro_ch_pre.m**
- **laplace.m**
- **calculate_energ.m**
- **free_energ_ch_v2.m**
- **write_vtk_grid_values.m**

Listing:

```

1    %%%%%%%%%%%%%%%%
2    %           %
3    % FINITE-DIFFERENCE PHASE-FIELD %
4    %           CODE FOR SOLVING %
5    % CAHN-HILLIARD EQUATION %
6    % (OPTIMIZED FOR MATLAB/OCTAVE) %
7    %%%%%%%%%%%%%%%%
8    %%= get intial wall time:
9    time0=clock();
10   format long;
11
12  out2 = fopen('time_energ.
13      out','w');
14
15  %% Simulation cell parameters:
16  Nx = 64;
17  Ny = 64;
18  NxNy= Nx*Ny;
19  dx = 1.0;
20  dy = 1.0;
21
22  %% Time integration parameters:
23
24  nstep = 20000;
25  nprint= 100;
26  dtim= 1.0e-2;
27  ttime = 0.0;
28
29  %% Material specific Parameters:
30
31  c0 = 0.40;
32  mobility = 1.0;
33  grad_coef= 0.5;

34
35  %
36  %---prepare microstructure:
37  %
38
39  iflag =2;
40
41  [con] = micro_ch_pre(Nx,Ny,c0,
42  iflag);
43
44  %--- Get Laplacian templet:
45  [grad] = laplacian(Nx,Ny,dx,dy);
46
47  %
48  %--- Evolve:
49  %
50
51  for istep =1:nstep
52
53  ttime = ttime+dtim;
54
55  %--- derivative of free energy:
56
57  [dfdcon]=free_energ_ch_v2(Nx,Ny,
58  con);
59  lap_con = grad*con;
60
61  lap_con2 =grad*(dfdcon -
62  grad_coef*lap_con);
63
64  %--- Time integration:
65
66  con = con + dtim * mobility *
67  lap_con2;
68
69  %% for small deviations:
70
71  inrange =(con >= 0.9999);
72  con(inrange) = 0.9999;
73
74  %--- print results
75
76  if((mod(istep,nprint) == 0) ||
77  (istep == 1) )

```

```

78 fprintf('done step: %5d\n',istep);
79
80 %fname1 = sprintf('time_%d.out',
81 istep);
82 %out1 = fopen(fname1,'w');
83 %for i=1:Nx
84 %for j=1:Ny
85 fprintf(out1,'%5d %5d %14.6e\n',i,
86 j,con(i,j));
87 %end
88 %end
89 %fclose(out1);
90
91 %--- write vtk file:
92
93 for i=1:Nx
94 for j=1:Ny
95 ii=(i-1)*Nx+j;
96 con2(i,j) = con(ii);
97 end
98 end
99
100 %-- calculate total energy
101
102 [energ] = calculate_energ(Nx,Ny,
103 con2,grad_coef);
104 fprintf(out2,'%14.6e %14.6e\n',
105 ttime,energ);
106 write_vtk_grid_values(Nx,Ny,dx,dy,
107 istep,con2);
108 end %if
109 end %istep
110
111 %--- calculate compute time:
112
113 compute_time = etime(clock(),
114 time0);
115 fprintf('Compute Time: %10d\n',
116 compute_time);

```

Line numbers:

8–12:	Get wall clock time beginning of the execution and open output file for printing total bulk energy values.
14–21:	Simulation cell parameters.
16:	Number of grid points in the x -direction.
17:	Number of grid points in the y -direction.
18:	Total number of grid points in the simulation cell.
19:	Grid spacing between two grid points in the x -direction
20:	Grid spacing between two grid points in the y -direction.
22–28:	Time integration parameters.
24:	Number of time integration steps.
25:	Output frequency to write the results to file.
26:	Time increment for numerical integration.
27:	Total time.
29–34:	Material-specific parameters.
31:	Average composition.
32:	Value of mobility coefficient.
33:	Value of gradient energy coefficient.
36–42:	Initialize the concentration field with random modulation.
39:	Set iflag = 1 for un-optimized mode, iflag = 2 for optimized mode.
43–46:	Calculate the finite difference template for the Laplacians.
51–110:	Time evolution of the concentration field.
53:	Update the total time.
55–58:	Calculate the derivative of free energy simultaneously at all grid points in the simulation cell.
59:	Calculate the Laplacian of the concentration at all grid points in the simulation cell, Eq. 4.17.
61:	Calculate the Laplacian of the terms inside the parenthesis in Eq. 4.16.
63–65:	Explicit Euler time integration of Eq. 4.16 at all grid points in the simulation cell.
67–73:	If there are small variations, set the max and min values to the limits.
74–108:	If print frequency is reached, print the results to file.
80–89:	Open an output file and print results. These lines are commented out, but they can be changed, including the format of the output file.

(continued)

100–104:	Calculate the total bulk energy and print the result to <i>time_energy.out</i> file for 2D plots.
106:	Write the results in vtk format for contour plots to be viewed by using Paraview.
112–115:	Calculate the total execution time and print it to screen.

Function**micro_ch_pre.m**

This function initializes the microstructure for given average composition modulated with a noise term to account the thermal fluctuations in Cahn–Hilliard equation.

Variable and array list:

Nx:	Number of grid points in the <i>x</i> -direction.
Ny:	Number of grid points in the <i>y</i> -direction.
c0:	Average alloy composition.
iflag:	iflag = 1 for the un-optimized code, iflag = 2 for optimized code.
con (NxNy):	Concentration field for optimized mode (iflag = 2).
con(Nx, Ny):	Concentration field for un-optimized mode (iflag = 1).

Listing:

```

1  function [con] =micro_ch_pre(Nx,Ny,
2                                c0,iflag)
3
4  format long;
5
6  NxNy = Nx*NY;
7
8  noise = 0.02;
9
10 if(iflag == 1)
11   for i=1:Nx
12     for j=1:Ny
13       con(i,j) =c0 + noise*(0.5-rand);
14     end
15   end
16
17 else
18   con=zeros(NxNy,1);

```

```

19
20   for i=1:Nx
21     for j=1:Ny
22       ii =(i-1)*Nx+j;
23       con(ii) =c0 + noise*(0.5-rand);
24     end
25   end
26
27 end %if
28
29 end %endfunction

```

Line numbers:

5:	Total number of grid points in the simulation cell.
7:	Set the magnitude of the noise term for fluctuations.
9:	If iflag = 1 un-optimized mode.
11–15:	Introduce random fluctuation to concentration, con(Nx,Ny).
17:	for optimized mode (iflag = 2).
18–25:	Introduce random fluctuation to concentration, con(NxNy).

Function**calculate_energ.m**

This function calculates the total bulk energy of the system. The results are output to file *time_energ.out* to generate time vs. energy plots.

Variable and array list:

Nx:	Number of grid points in the <i>x</i> -direction.
Ny:	Number of grid points in the <i>y</i> -direction.
grad_coef:	Gradient energy coefficient.
energ:	Total bulk energy value.
con(Nx,Ny):	Concentration field.

Listing:

```

1  function [energ] = calculate_energ
2                                (Nx,Ny,con,grad_coef)
3
4  format long;
5
6  energ =0.0;
7
8  for i=1:Nx-1
9    ip = i + 1;

```

```

9   for j=1:Ny-1
10  jp = j + 1;
11
12  energ = energ + con(i,j)^2*
13    (1.0-con(i,j))^2 + ...
14  0.5*grad_coef*((con(ip,j)-
15    con(i,j))^2 + (con(i,jp)-
16    con(i,j))^2);
17 end
18 end
19
20 end %endfunction

```

Line numbers:

5:	Initialize
7–15:	Calculate Eq. 4.19.

Function

free_energ_ch_v1.m

This function calculates the derivative of free energy, Eq. 4.18 at the current grid point in the main program.

Variables and array list:

i:	Current grid point in the <i>x</i> -direction.
j:	Current grid point in the <i>y</i> -direction.
dfdcon:	Derivative of free energy.
con(Nx,Ny):	Concentration field.

Listing:

```

1  function [dfdcon] =free_energ_ch_
v1(i,j,con)
2
3  format long;
4
5  A=1.0;
6
7  dfdcon =A*(2.0*con .* (1-con).^2
8    -2.0*con.^2 .* (1.0-con));
9 end %endfunction

```

Line numbers:

5:	Constant in free energy function, Eq. 4.18.
7:	Derivative of free energy, Eq. 4.18 at the current grid point in the main program.

Function

free_energ_ch_v2.m

This function calculates the derivative of free energy, Eq. 4.18 simultaneously at all the grid points in the simulation cell. It is used by Matlab/Octave optimized code.

Variable and array list:

Nx:	Number of grid points in the <i>x</i> -direction.
Ny:	Number of grid points in the <i>y</i> -direction.
con(NxNy):	Concentration field.
dfdcon (NxNy):	Derivative of free energy at all grid points in the simulation cell.

Listing:

```

1  function [dfdcon] =free_energ_ch_v2
(Nx,Ny,con)
2
3  format long;
4
5  A=1.0;
6
7  dfdcon =A*(2.0*con .* (1-con).^2
8    -2.0*con.^2 .* (1.0-con));
9 end %endfunction

```

Line numbers:

5:	Constant in free energy function, Eq. 4.18.
7:	Derivative of free energy, Eq. 4.18 at all grid points in the simulation cell.

Function

write_vtk_grid_values.m

This function writes the grid point values to a vtk file to generate the contour plots to be viewed by

using Paraview. This function and its slightly modified versions are used throughout the book for visualization and animation of the results. Although, Matlab and Octave have impressive graphic routines; however, Paraview has more powerful features for visualization and spatial and temporal analysis of the data.

Variable and array list:

nx:	Number of grid points in the <i>x</i> -direction.
ny:	Number of grid points in the <i>y</i> -direction.
dx:	Grid spacing between two grid points in the <i>x</i> -direction.
dy:	Grid spacing between two grid points in the <i>y</i> -direction.
istep:	Time step number for generation of output files in sequel.
data1(nx,ny):	Data values at the grid points.

Listing:

```

1  function [ ]=write_vtk_grid_values
2    (nx,ny,dx,dy,istep,data1)
3
4    format long
5
6    %-- open output file
7
8    fname=sprintf('time_%d.vtk',
9    istep);
10   out =fopen(fname,'w');
11
12   nz=1;
13
14   % start writing ASCII VTK file:
15
16   % header of VTK file
17
18   fprintf(out,'# vtk DataFile
19   Version 2.0\n');
20   fprintf(out,'time_10.vtk\n');
21   fprintf(out,'ASCII\n');
22   fprintf(out,'DATASET STRUCTURED_
23   GRID\n');
24
25   fprintf(out,'DIMENSIONS %5d %5d
26   %5d\n',nx,ny,nz);
27
28   for i = 1:nx
29     for j = 1:ny
30
31       x =(i-1)*dx;
32       y =(j-1)*dy;
33       z = 0.0;
34
35       fprintf(out, '%14.6e %14.6e
36       %14.6e\n',x,y,z);
37     end
38   end
39
40   %--- write grid point values:
41
42   fprintf(out,'POINT_DATA %5d\n',
43   npoin);
44
45   fprintf(out,'SCALARS CON
46   float 1\n');
47
48   fprintf(out,'LOOKUP_TABLE
49   default\n');
50
51   for i = 1:nx
52     for j = 1:ny
53       ii=(i-1)*nx+j;
54
55       fprintf(out,'%14.6e\n',data1(i,j));
56     end
57   end
58
59   fclose(out);
60
61 end %endfunction

```

Line numbers:

5–9:	Open output file.
10:	nz = 1 for 2D plots, and it takes the value of number of grid points in <i>z</i> -direction for 3D plots.
12:	Total number of grid points.
16–22:	The header of the vtk file. These lines should not be altered.
23–38:	Write the coordinates of the grid points.
40–54:	The headers and grid point values.

If more than one dataset need to be written to the file:

1. These data sets should be included as input to the function in line 1, i.e., data1, data2, data3....etc.).
2. The lines 44 through 54 should be repeated for each new data set. The title in line 44, should also be changed representing the content of the data set. For example if next dataset is for order parameters something like ‘SCALARS ORP float 1\n’.

Function

laplacian.m

This function generates the Laplace operator by using five-point stencil with periodic boundaries. It is used in optimized Matlab/Octave programs throughout this chapter.

Variable and array list:

nx:	Number of grid points in the <i>x</i> -direction.
ny:	Number of grid points in the <i>y</i> -direction.
dx:	Grid spacing between two grid points in the <i>x</i> -direction.
dy:	Grid spacing between two grid points in the <i>y</i> -direction.
grad(nxny, nxny):	Laplace operator.

Listing:

```

1  function [grad] =laplacian
2    (nx,ny,dx,dy)
3    format long;
4
5    nxny=nx*ny;
6
7    r=zeros(1,nx);
8    r(1:2)=[2,-1];
9    T=toeplitz(r);
10
11   E=speye(nx);
12
13   grad=-(kron(T,E)+kron(E,T));
14

```

```

15 %-- for periodic boundaries
16
17 for i=1:nx
18   ii=(i-1)*nx+1;
19   jj=ii+nx-1;
20   grad(ii,jj)=1.0;
21   grad(jj,ii)=1.0;
22
23 kk=nxny-nx+i;
24 grad(i,kk)=1.0;
25 grad(kk,i)=1.0;
26 end
27
28 grad=grad / (dx*dy);
29
30 end %endfunction

```

Line numbers

5:	Total number of grid points in the simulation cell.
6–14:	Generate Laplace operator for five-point stencil by using Matlab/Octave built-in functions, zeros, toeplitz, speye, and kron functions.
15–29:	Modify the stencil for periodic boundaries.
28:	Introduce dimensionality to the stencil.

References

1. Bertozzi A, Esoeglu S, Gillete A (2007) Inpainting of binary images using the Cahn–Hilliard equation. *IEEE Trans Image Process* 16:285
2. Kim JS (2005) A continuous surface tension force formulation for diffuse-interface models. *J Comput Phys* 204:784
3. Boyer F (2002) A theoretical and numerical model for the study of incompressible mixture flows. *Comput Fluids* 31:41
4. Parks JM, Mauric R, Anderson PD (2012) Phase separation of viscous ternary liquid mixtures. *Chem Eng Sci* 80:270
5. Khatavkar VV, Anderson PD, Meijer HEH (2006) On scaling of diffusive-interface models. *Chem Eng Sci* 61:2364
6. Elliott CM, French DA (1987) Numerical studies of the Cahn–Hilliard equation for phase separation. *IMA J Appl Math* 38:97
7. Garcke H, Preuber T, Rumpf M, Telea A, Weikard U, van Wijk J (2001) A phase-field model for continuous clustering on vector fields. *IEEE Trans Vis Comput Graph* 7:230

8. Wise SM, Lowengrub JS, Kim JS, Thornton K, Voorhees PW, Johnson WDC (2005) Quantum dot formation on a strain-patterned epitaxial thin film. *Appl Phys Lett* 87:133102
9. Ganapathy H, Al-Hajri A, Ohadi MM (2013) Phase-field modeling of Taylor flow in mini/microchannels. Part-II: Hydrodynamics of Taylor flow. *Chem Eng Sci* 94:156
10. Zhang L, Tonks MR, Millett PC, Zhang Y, Chockalingam K, Biner SB (2012) Phase-field modeling of temperature gradient driven pore migration coupling with thermal conduction. *Comput Mater Sci* 56:161
11. Cristini V, Li X, Lowengrub J, Wise SM (2009) Nonlinear simulations of solid tumor growth using a mixture model: invasion and branching. *J Math Biol* 58:723
12. Wise SM, Lowengrub J, Frieboes HB, Cristini V (2008) Three dimensional multispecies nonlinear tumor growth-I: model and numerical method. *J Theor Biol* 253:524
13. Eyre DJ (1993) Systems for Cahn–Hilliard equations. *SIAM J Appl Math* 53:1686
14. Kim JS (2007) A numerical method for the Cahn–Hilliard equation with a variable mobility. *Commun Nonlinear Sci Numer Simul* 12:1560
15. Khiari N, Achuri T, Ben Mohamed ML, Omrani K (2007) Finite difference approximate solutions for the Cahn–Hilliard equation. *Numer Methods Partial Differ Equat* 23:437
16. Ceniceros HD, Roma AM (2007) A non-stiff, adaptive mesh refinement based method for the Cahn–Hilliard equation. *J Comput Phys* 225:1849
17. Copetti M, Elliott CM (1990) Kinetics of phase decomposition process: numerical solutions to the Cahn–Hilliard equation. *Mater Sci Technol* 6:273
18. Du Q, Nicolaides R (1991) Numerical studies of continuum model of phase transition. *SIAM J Numer Anal* 28:1310
19. Zhang L (2010) Long time behavior of difference approximations for the two-dimensional complex Ginzburg–Landau equation. *Numer Funct Anal Optim* 31:1190
20. Acar R (2009) Simulation of interface dynamics: a diffuse-interface model. *Visual Comput* 25:101
21. Eyre DJ (1998) Unconditionally gradient stable time marching the Cahn–Hilliard equation. In: Bullard JW, Kaalia R, Stoneham M, Chen L (eds) Computer and mathematical modeling of microstructure evolution. Material Research Society
22. Kim JS, Bae HO (2008) An unconditionally stable adaptive mesh refinement for Cahn–Hilliard equation. *J Korean Phys Soc* 53:672
23. Kim JS, Lee C, Lee HG, Kim J (2013) Comparison of different numerical for the Cahn–Hilliard equation. *J KSIAM* 17:197

4.5 Case Study-II

Phase-field modeling of grain growth with finite-difference algorithm

Objectives:

The objective of this case study is to demonstrate a numerical implementation of the finite difference algorithm for the solution of multicomponent non-conserved Allen–Cahn equations in phase-field modeling.

4.5.1 Background

The microstructure of most of engineering materials consists of multiple grains with different crystallographic orientations. The understanding of the factors influencing the evolution of a grain structure is of great scientific and technological importance, because physical properties, such as corrosion resistance, thermal and electrical conductivity; and mechanical properties, such strength, ductility, and toughness, depend on the mean grain size and the grain size distribution.

The local energy at the grain boundaries, or interfaces, is higher than the corresponding bulk energies of the grains forming the grain boundaries. This extra energy associated with the grain boundaries provides a thermodynamic driving force to move the grain boundary in a manner as to minimize the total free energy. Such motion of grain boundaries in polycrystalline materials results in grain growth which is a process of growth of some grains and shrinking of some others. Thus, grain growth decreases the total number of grains and increases the average grain size.

In addition to extensive theoretical and experimental studies [1], computer simulations are also applied to elucidate the mechanisms of grain growth ranging from atomistic to continuum. Computational studies are extremely useful since they permit to isolate and study the

dominant controlling effects of the grain growth leading to the microstructural evolution. At meso-scale, different computational approaches, Monte Carlo Potts model [2], Surface Evolver [3], the front-tracking method [4], vertex dynamics [5], and cellular automata [6], have been applied. In simulations, involving thousands of grains, tracking individual grain boundaries, and applying specific constitutive relations to their motion, computationally is quite demanding. Therefore, phase-field modeling approaches have been applied extensively to modeling of grain growth [7–12]. However, all these computational studies, in spite of differences in their approaches, have confirmed the parabolic grain growth kinetics introduced by Burke and Turnbull in early 1950s [13].

Several phase-field models have been proposed to simulate the grain growth kinetics of polycrystalline materials. One of the earliest and most widely applied models in which the grains of different crystallographic orientations are represented by a set of non-conserved order parameter fields is that of Chen and Yang [7]. This model has been used in 2D [8] and 3D [9] phase-field modeling simulations of grain growth. Another class of multiphase field model was proposed by Steinbach et al. [10], with a constraint on the order parameters, such that the sum of all order parameters at a given point yields to unity. The physical interpretation of this constraint is that the order parameters represent the volume fraction of grains of different orientations. Warren and coworkers [11, 12] have proposed a two order parameter model, in which the crystalline order and predominant local orientation of the crystal are represented.

4.5.2 Phase-Field Model

For this case study, the phase-field model is the adaptation of the grain growth model given by Fan and Chen [8]. In the model, each grain is described by one order parameter η_i which takes the value of one for a designated grain and the value of zero in other grains. The evolution of the

order parameters described by the non-conserved Allen–Cahn equation in the form of:

$$\frac{\partial \eta_i}{\partial t} = -L_i \frac{\delta F}{\delta \eta_i}, \quad i = 1, 2, \dots, N \quad (4.31)$$

where L_i is the mobility coefficient and F is the free energy functional which is taken as,

$$F = \int_V \left[f(\eta_1, \eta_2, \dots, \eta_N) + \sum_i^N \frac{\kappa_i}{2} |\nabla \eta_i|^2 \right] dv \quad (4.32)$$

in which κ_i are the gradient energy coefficients and f is the local free energy density.

The specific form of local free energy which is independent of orientation is given in [8] as,

$$f(\eta_1, \eta_2, \dots, \eta_N) = \sum_i^N \left(-\frac{A}{2} \eta_i^2 + \frac{B}{4} \eta_i^4 \right) + \sum_i^N \sum_{i \neq j}^N \eta_i^2 \eta_j^2 \quad (4.33)$$

in which A and B are positive constants. For $A = B = 1$, for two order parameters η_1 and η_2 the shape of this local free functional is given in Fig. 4.6. As can be seen, it goes through a minimum at equal depth for η values of zero and one.

4.5.3 Numerical Implementation

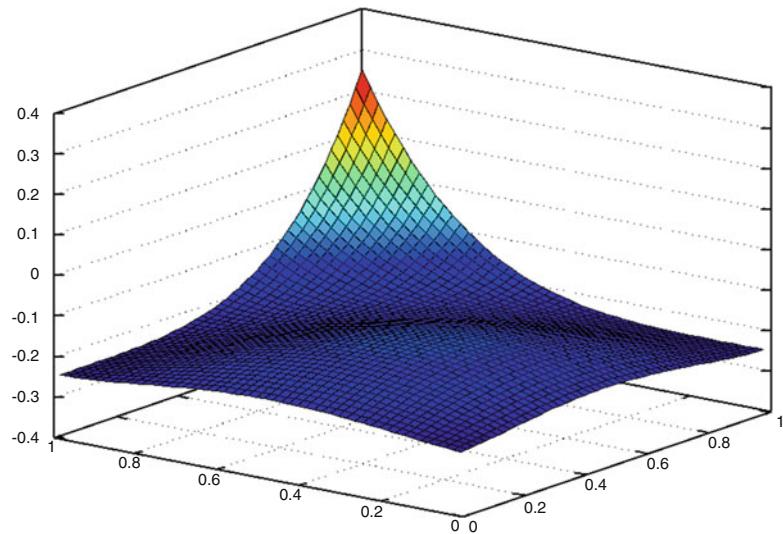
Substituting Eqs. 4.32 and 4.33 into Eq. 4.31, the governing equation for evolution becomes,

$$\frac{\partial \eta_i}{\partial t} = -L_i \left(-A\eta_i + B\eta_i^3 + 2\eta_i \sum_{i \neq j}^N \eta_j^2 - \kappa_i \nabla^2 \eta_i \right), \quad i = 1, 2, \dots, N \quad (4.34)$$

The discrete Laplace operator will be approximated by using the five-point stencil described earlier.

The time integration of Eq. 4.34 will be carried out with simple explicit Euler time marching scheme. Thus,

Fig. 4.6 For $A = B = 1$ and for two order parameters η_1 (x-axis) and η_2 (y-axis) the shape of the local free functional (z-axis), Eq. 4.33



$$\frac{\partial \eta_i}{\partial t} = \frac{\eta_i^{n+1} - \eta_i^n}{\Delta t} = -L \left(-A\eta_i^n + B(\eta_i^n)^3 + 2\eta_i^n \sum_{i \neq j}^N (\eta_j^n)^2 - \kappa_i \nabla^2 \eta_i^n \right), i = 1, 2, \dots, N \quad (4.35)$$

which will be evaluated at each time step.

4.5.4 Results and Discussion

The simulations were carried out for a square simulation cell having $N_x = 64$ and $N_y = 64$ and with $dx = dy = 0.5$. All of the parameters used in the simulation were in nondimensional form.

First set of simulations were carried out for an ideal grain growth which is a special case of normal grain growth where the grain boundary motion is driven only by the local curvature of the grain boundary. For this specific case, it is shown that the rate of growth is proportional to the driving force and the driving force is proportional to the total amount of grain boundary energy. For a shrinking spherical grain embedded into a large second grain, it is shown that the change in the radius of the shrinking grain can be approximated as:

$$D^2 - D_0^2 = kt \quad (4.36)$$

where D and D_0 are the current and initial grain radius of the spherical grain, respectively, t is the time, and k is a temperature-dependent constant given by an Arrhenius' equation,

$$k = k_0 \exp\left(\frac{-Q}{RT}\right) \quad (4.37)$$

in which k_0 is a constant, T is the absolute temperature, R is the gas constant, and Q is the activation energy for grain boundary mobility.

Figure 4.7 shows the time evolution of the spherical grain with initial radius of $D_0 = 14dx$ during the course of simulation. With increasing time, the uniform shrinkage of the spherical grain is apparent from the figure. The variation of the area fraction of the grain with time is shown in Fig. 4.8. Even though every coarse mesh is used in the simulations, the results obtained from the simulations agree very well with Eq. 4.36. As the analytical solution suggests, the radius of the spherical grain changed linearly with time or its shrinking rate remained constant with time.

In the second set of simulations, the evolution of polycrystalline microstructure due to the grain

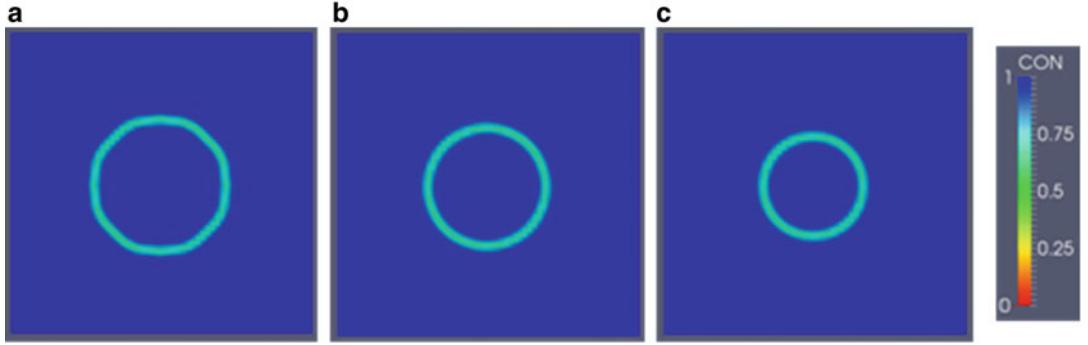
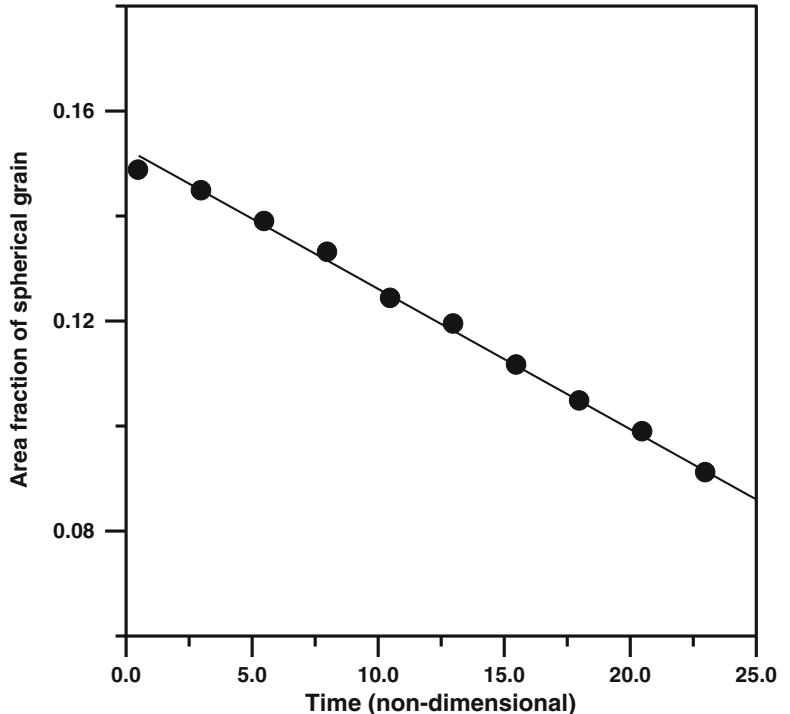


Fig. 4.7 Time evolution of a spherical grain. Nondimensional times are: (a) 0.5, (b) 12.5, and (c) 25.0

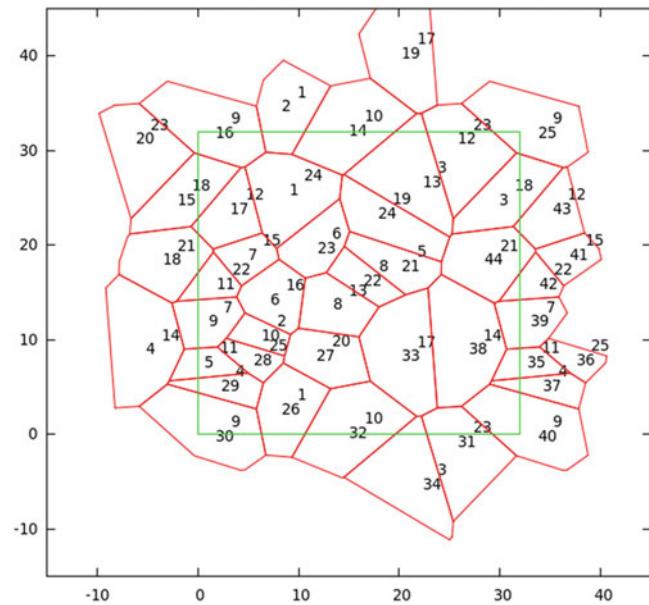
Fig. 4.8 Variation of area fraction of spherical grain with time



growth is studied. The Matlab/Octave program that prepares the input file for these polycrystal simulations based on Voronoi tessellation is described in [Appendix A](#). Figure 4.9 shows the graphical output generated with this program for a randomly distributed 25 grains microstructure which was used in the following phase-field simulations. In the figure, the Voroni cells are represented with red lines and the green square box is the simulation cell of the phase-field

model. The upper numbers 1 through 25 represents the grain numbers and the other lower ones are the Voroni cell numbers as explained in [Appendix A](#). As can seen, the resulting grain microstructure is periodic. The input file *grain_25.inp* that contains the information shown in Fig. 4.9 is given in the subdirectory *case_study_2* in the downloadable file. The function *init_grain_micro.m* reads this input file and maps the topology of the grain microstructure by

Fig. 4.9 The graphical output of the *grain_25.inp* file used in the phase-field simulations



assigning the relevant order parameters to the grid points of the simulation cell.

The time evolution of the polycrystalline microstructure composed of 25 grains is shown in Fig. 4.10. As can be seen, the evolution proceeds with the growth of the larger grains and the disappearance of the smaller ones. Figure 4.11 shows more quantitative information from the output file *area_fractions.out*. The variation of sizes of those first nine grains with time is shown in the figure. As can be seen, although the small grains disappear early, however, growth rate of larger ones differs from each other. Such information can be used in statistical analysis for determination of mean grain size, etc.

The movie files resulting from these simulations can be found in subdirectory *case_study_2* in downloadable file.

4.5.5 Source Codes

Two programs are provided in this section. The first one, *fd_ca_v1.m*, is in longhand format and the second version, *fd_ca_v2.m*, is the optimized version for Matlab/Octave. There are two flags in

both programs. The iflag should set as: iflag = 1 for two grains simulations and iflag = 2 for the polycrystal simulations. The input file *grain_25.inp* can be found in the subdirectory *case_study_2*. The flag isolve = 1 is for longhand, un-optimized code and isolve = 2 is the optimize code. The description of the programs and functions are given below. The functions described earlier are not repeated again in here.

Program

fd_ca_v1.m

This program solves the non-conserved multicomponent Allen–Cahn equation with finite difference algorithm using five-point stencil. The time integration is carried out with explicit Euler scheme. It is not optimized and in longhand format.

The program makes calls to the following functions:

- ***init_grain_micro.m***
- ***free_energ_ca_v1.m***
- ***write_vtk_grid_values.m***

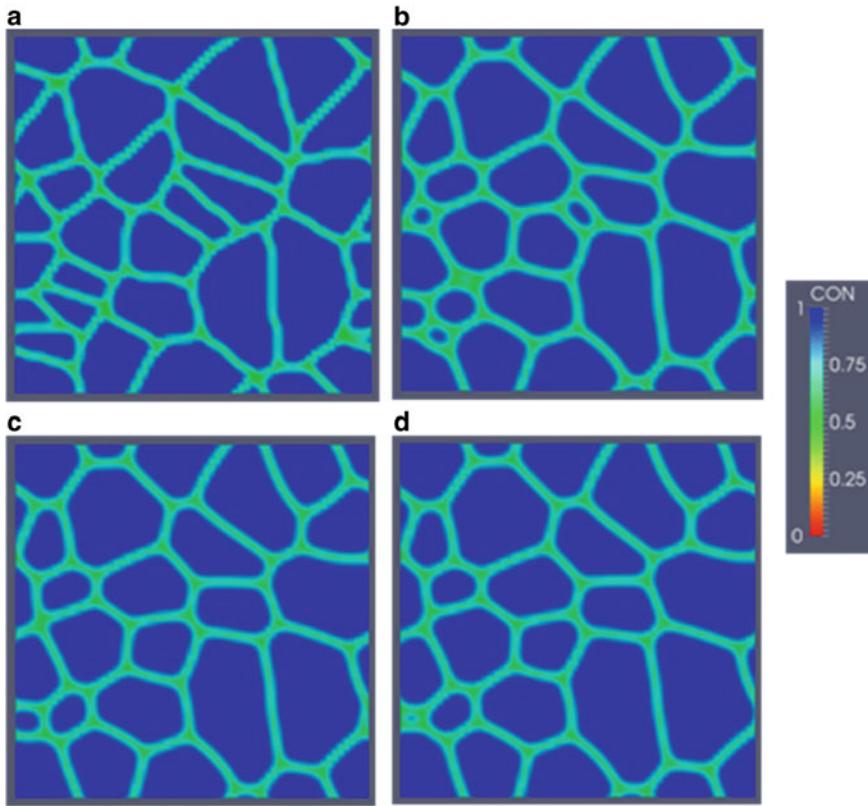
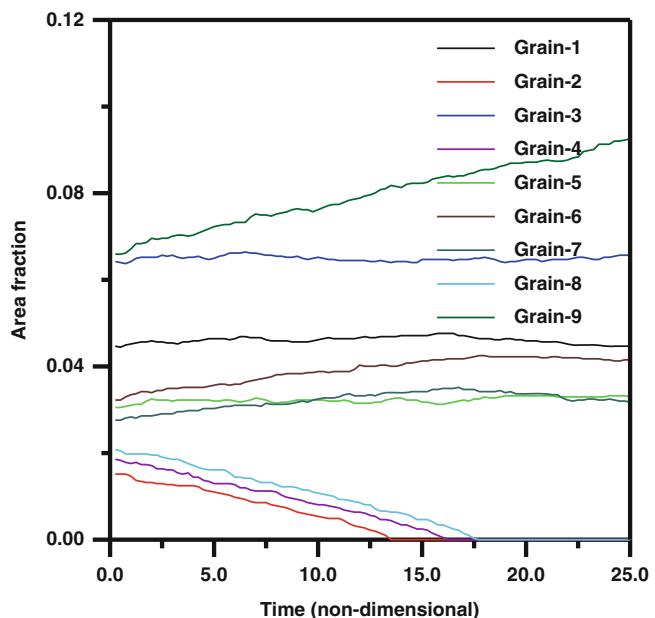


Fig. 4.10 Time evolution of polycrystalline microstructure due to grain growth. The nondimensional times are: (a) 0.5, (b) 12.5, (c) 18.75, and (d) 25.0

Fig. 4.11 Variation of area fraction of first nine grains with time



Listing:

```

1  %%%%%%%%%%%%%%%%
2  %
3  % FINITE DIFFERENCE PHASE-FIELD %
4  %      CODE FOR SOLVING %
5  %      ALLEN-CAHN EQUATION %
6  %
7  %== get initial wall time
8  time0=clock();
9  format long;
10 out2=fopen('area_frac.out','w');
11 %-- Simulation cell parameters:
12 Nx = 64;
13 Ny = 64;
14 NxNy= Nx*NY;
15 dx = 0.5;
16 dy = 0.5;
17 %--- Time integration parameters:
18 nstep = 100;
19 nprint= 10;
20 dtime = 0.005;
21 ttime = 0.0;
22 %--- Material Parameters
23 mobil = 5.0;
24 grcoef = 0.1;
25 %
26 %--- Generate initial grain_
27 structure
28 %
29 iflag = 2;
30 isolve = 1;
31 [etas,ngrain,glist] =
32 init_grain_micro(Nx,Ny,dx,dy,
33 iflag,isolve);
34 %
35 %-- Evolve:
36
37 for istep =1:nstep
38
39
40
41
42
43
44 %
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93

```

```

94 hns=eta(i,jm);
95 hnn=eta(i,jp);
96 hnc=eta(i,j);
97
98 lap_eta(i,j) =(hnw + hne + hns + hnn
-4.0*hnc) / (dx*dy);
99 %
100 %
101 %%Derivative of free energy
102 %
103
104 dfdeta =free_energ_fd_ca_v1(i,j,
ngrain,etas,eta,igrain);
105 %
106 %
107 %--- Time integration:
108 %
109
110 eta(i,j) = eta(i,j) - dtime*mobil*
(dfdata - grcoef*lap_eta(i,j));
111
112 %% for small deviations:
113
114 if(eta(i,j) >= 0.9999);
115 eta(i,j) = 0.9999;
116 end
117 if(eta(i,j) < 0.00001);
118 eta(i,j) = 0.00001;
119 end
120
121 end %j
122 end %i
123 %
124 grain_sum = 0.0;
125 for i=1:Nx
126 for j=1:Ny
127 etas(i,j,igrain) =eta(i,j);
128 grain_sum =grain_sum + eta(i,j);
129 end
130 end
131
132 end
133 end
134
135 %% Check volume fraction of current
grain:
136
137 grain_sum =grain_sum/NxNy;
138
139 if(grain_sum <= 0.001)
140 glist(igrain) =0;
141 fprintf('grain: %5d is eliminated
\n',igrain);
142 end
143 %---
144
145 end %if
146 end %igrain
147
148 if(mod(istep,nprint) == 0)
149
150 fprintf('done step: %5d\n',istep);
151
152 %fname1 =sprintf('time_%d.out',
istep);
153 %out1 = fopen(fname1,'w');
154
155 %for i=1:Nx
156 %for j=1:Ny
157
158 % gg= 0.0;
159 %for igrain =1:ngrain
160 %gg = gg +etas(i,j,igrain)^2;
161 %end
162
163 %fprintf(out1,'%5d%5d%14.6e\n',i,
j,gg);
164 %end
165 %end
166
167 %fclose(out1);
168
169 %--- write vtk file & calculate area
fraction of grains:
170
171 eta2=zeros(Nx,Ny);
172
173 fprintf(out2,'%14.6e',ttime);
174
175 for igrain=1:ngrain
176 ncount=0;
177 for i=1:Nx
178 for j=1:Ny
179
180 eta2(i,j) = eta2(i,j)
+ etas(i,j,igrain)^2;

```

```

181
182 if(etas(i,j,igrain) >= 0.5)
183 ncount=ncount+1;
184 end
185
186 end
187 end
188 ncount=ncount/(NxNy);
189 fprintf(out2,'%14.6e',ncount);
190 end
191 fprintf(out2,'\n');
192
193 %--
194
195 write_vtk_grid_values(Nx,Ny,dx,
dy,istep,eta2);
196
197 end %end if
198
199 end %istep
200
201 %--- calculate compute time:
202
203 compute_time = etime(clock(), time0);
204 fprintf('Compute Time: %10d\n',
compute_time);
205

```

Line numbers:

8–13:	Get wall clock time beginning of the execution and open output file for printing area fraction of each grain.
14–21:	Simulation cell parameters.
16:	Number of grid points in the <i>x</i> -direction.
17:	Number of grid points in the <i>y</i> -direction.
18:	Total number of grid points in the simulation cell.
19:	Grid spacing between two grid points in the <i>x</i> -direction.
20:	Grid spacing between two grid points in the <i>y</i> -direction.
22–28:	Time integration parameters.
24:	Number of time integration steps.
25:	Output frequency to write the results to file.
26:	Time increment for time integration.
27:	Total time.
29–33:	Material parameters.
31:	Mobility coefficient.

32:	Gradient energy coefficient.
35–41:	Generate initial grain microstructure. iflag = 1 for bi-crystal and iflag = 2 is for polycrystal.
43–199:	Time integration.
48:	Update the total time.
50–146:	Loop over each grain.
52:	If glist is equal to one, which indicates that the current grain area fraction is greater than 0.001, continue the calculation. Otherwise, the current grain does not exist anymore.
54–60:	Assign order parameters to temporary array eta(Nx,Ny) from the common array etas(Nx, Ny,ngrain) for the current grain.
62–98:	Calculate the Laplacian term in Eq. 4.35.
101–104:	Determine the derivative of the free energy for the current grain at the current grid point under consideration, first three terms inside the bracket in Eq. 4.35.
110:	With explicit Euler integration, time integration of order parameter of current point at current grid point, Eq. 4.35.
112–119:	If there are small deviations, set the max and min values to the limits.
126–133:	Calculate the total area of the current grain, also return the order parameters values from the temporary array etas(Nx,Ny) to common array etas(Nx,Ny,ngrain).
135–142:	Check the area fraction of the current grain. If it is less then 0.001, set its value in glist (ngrain) as zero which indicates that it is extinct. Also print message “grain # is eliminated” to screen.
148–197:	If print frequency is reached, print the results to file.
152–167:	Open an output file and print the results. These lines are commented out but they can be changed, including the output format.
169–195:	Prepare the data to be written to vtk file and calculate the area fraction of each grain and print them to file <i>area_fract.out</i> .
195:	Write the results to vtk file for contour plots to be viewed by using Paraview.
201–204:	Calculate the total execution time and print it to screen.

Program**fd_ca_v2.m**

This program solves the non-conserved multi-component Allen–Cahn equation with five-point stencil finite difference algorithm. The time

integration is carried out with explicit Euler integration scheme. It is optimized for Matlab/Octave.

The program makes calls to the following functions:

- **init_grain_micro.m**
- **laplacian.m**
- **free_energ_ca_v2.m**
- **write_vtk_grid_values.m**

Listing:

```

1    %%%%%%%%%%%%%%%%
2    %
3    % FINITE DIFFERENCE PHASE-FIELD
4    %     CODE FOR SOLVING
5    %     ALLEN-CAHN EQUATION
6    %     (OPTIMIZED FOR MATLAB/OCTAVE)
7    %
8    %== get initial wall time
9    time0=clock();
10   format long;
11
12   out2=fopen('area_frac.out','w');
13
14   %-- Simulation cell parameters:
15
16   Nx = 64;
17   Ny = 64;
18   NxNy= Nx*Ny;
19   dx = 0.5;
20   dy = 0.5;
21
22   %-- Time integration parameters:
23
24   nstep = 5000;
25   nprint= 100;
26   dtim= 0.005;
27   ttime = 0.0;
28
29   %-- Material Parameters
30
31   mobil = 5.0;
32   grcoef = 0.1;
33
34   %
35
36   %--- Generate initial grain_
37   structure
38   iflag = 2;
39   isolve = 2;
40
41   [etas,ngrain,glist] =
42   init_grain_micro(Nx,Ny,dx,dy,
43   iflag,isolve);
44
45   %--- Get Laplacian templet
46   %
47   [grad]=laplacian(Nx,Ny,dx,dy);
48
49   %
50   % Evolve:
51   %
52
53   eta=zeros(Nx*Ny,1);
54
55   for istep=1:nstep
56
57   ttime=ttime+dtim;
58
59   for igrain=1:ngrain
60
61   if(glist(igrain) == 1)
62
63   eta=etas( :, igrain );
64
65   %
66   %--Derivative of free energy
67   %
68
69   dfdeta=free_energ_fd_ca_v2(Nx,Ny,
70   ngrain,etas,eta,igrain);
71
72   %
73   %--- Time integration:
74
75   eta=eta - dtim*mobil*(dfdeta -
76   grcoef*grad*eta);
77
78   %-- for small deviations:
79
80   inrange=(eta >= 0.9999);
81
82   eta(inrange) = 0.9999;
```

```

81  inrange=(eta < 0.00001);
82  eta(inrange) = 0.00001;
83
84 %--
85
86 etas( :,igrain) =eta;
87
88 %-- Check volume fraction of current
89 % grain:
90
91 grain_sum = 0.0;
92 grain_sum = sum(eta)/NxNy;
93
94 if(grain_sum <= 0.001)
95 glist(igrain) =0;
96 fprintf('grain: %5d is eliminated
97 \n',igrain);
98 end
99 end %if
100 end %igrain
101
102 if(mod(istep,nprint) == 0)
103
104 fprintf('done step: %5d\n',istep);
105
106 %fname1 =sprintf('time_%d.out',
107 istep);
108 %out1 = fopen(fname1,'w');
109 %for i=1:Nx
110 %for j=1:Ny
111 %ii =(i-1)*Nx+j
112
113 % gg= 0.0;
114 %for igrain =1:ngrain
115 %gg = gg +etas(ii,igrain)^2;
116 %end
117
118 %fprintf(out1,'%5d %5d %14.6e\n',
119 i,j,gg);
120 %end
121
122 %fclose(out1);
123
124 %--- write vtk file & calculate area
125 fraction of grains:
126 eta2=zeros(Nx,Ny);
127
128 fprintf(out2,'%14.6e',ttime);
129
130 for igrain=1:ngrain
131 ncount=0;
132 for i=1:Nx
133 for j=1:Ny
134 ii =(i-1)*Nx+j;
135 eta2(i,j) =eta2(i,j)+etas(ii,
136 igrain)^2;
137
138 if(etas(ii,igrain) >= 0.5)
139 ncount=ncount+1;
140
141 end
142
143 end
144 end
145 ncount=ncount/(NxNy);
146 fprintf(out2,'%14.6e',ncount);
147
148 fprintf(out2,'\n');
149
150 %--
151
152 write_vtk_grid_values(Nx,Ny,dx,
153 dy,istep,eta2);
154 end %end if
155
156 end %istep
157
158 %--- calculate compute time:
159
160 compute_time = etime(clock(),
161 time0);
162 fprintf('Compute Time: %10d\n',
163 compute_time);

```

Line numbers:

Line numbers:

9–14:	Get wall clock time beginning of the execution and open output file for printing area fraction of each grain.
15–22:	Simulation cell parameters.
17:	Number of grid points in the x -direction.

(continued)

18:	Number of grid points in the y -direction.	152:	Write the results to vtk file for contour plots to be viewed by using Paraview.
19:	Total number of grid points in the simulation cell.	158–161:	Calculate the total execution time and print it to screen.
20:	Grid spacing between two grid points in the x -direction.		
21:	Grid spacing between two grid points in the y -direction.		
23–29:	Time integration parameters.		
25:	Number of time integration steps.		
26:	Output frequency to write the results to file.		
27:	Time increment for time integration.		
28:	Total time.		
30–34:	Material parameters.		
32:	Mobility coefficient.		
33:	Gradient energy coefficient.		
36–42:	Generate initial grain microstructure. iflag = 1 for bi-crystal and iflag = 2 is for polycrystal.		
44–48:	Calculate finite difference template for the Laplacians.	Nx:	Number of grid points in the x -direction.
50–156:	Time integration.	<td>Number of grid points in the y-direction.</td>	Number of grid points in the y -direction.
53:	Initialize temporary array eta(NxNy).	dx:	Grid spacing between two grid points in the x -direction.
57:	Update the total time.	dy:	Grid spacing between two grid points in the y -direction.
59–100:	Loop over each grain.	iflag:	If iflag = 1 bicrystal simulation, iflag = 2 polycrystal simulation.
61:	If glist is equal to one, which indicates that the current grain area fraction is greater than 0.001, continue the calculation. Otherwise, the current grain does not exist anymore.	isolve:	If isolve = 1, for un-optimized, longhand code, if isolve = 2 for Matlab/Octave optimized code.
63:	Assign order parameters to temporary array eta(NxNy) from the common array etas(NxNy,ngrain) for the current grain.	ngrain:	Number of grains in the simulation.
66–70:	Calculate the derivative of free energy at all grid points in the simulation cell, first three terms inside bracket in Eq. 4.35.	glist(ngrain):	Flag for existing grains.
72–76:	Explicit Euler time integration of Eq. 4.35 at all grid points.	etas(NxNy, ngrain):	Common array containing order parameters of grains, isolve = 2.
77–83:	If there are small deviations, set the max and min values to the limits.	etas(Nx,Ny, ngrain):	Common array containing order parameters of grains, isolve = 1.
86:	Return the order parameters values from the temporary array etas(NxNy) to common array etas(NxNy,ngrain).		
88–96:	Check the area fraction of the current grain. If it is less then 0.001, set its value in glist (ngrain) as zero which indicates that it is extinct. Also print message “grain # is eliminated” to screen.		
102–154:	If print frequency is reached, print the results to file.		
106–122:	Open an output file and print the results. These lines are commented out but they can be changed, including the output format.		
124–149:	Prepare the data to be written to vtk file and calculate the area fraction of each grain and print them to file <i>area_fract.out</i> .		

Function

init_grain_micro.m

This function generates the order parameters for a circular bicrystal or a polycrystalline micro-structure consist of 25 grains which is generated with the source code given in [Appendix A](#).

Variable and array list:

Nx:	Number of grid points in the x -direction.
Ny:	Number of grid points in the y -direction.
dx:	Grid spacing between two grid points in the x -direction.
dy:	Grid spacing between two grid points in the y -direction.
iflag:	If iflag = 1 bicrystal simulation, iflag = 2 polycrystal simulation.
isolve:	If isolve = 1, for un-optimized, longhand code, if isolve = 2 for Matlab/Octave optimized code.
ngrain:	Number of grains in the simulation.
glist(ngrain):	Flag for existing grains.
etas(NxNy, ngrain):	Common array containing order parameters of grains, isolve = 2.
etas(Nx,Ny, ngrain):	Common array containing order parameters of grains, isolve = 1.

Listing:

```

1  function [etas,ngrain,glist] =
2    init_grain_micro(Nx,Ny,dx,dy,
3      iflag,isolve)
4
5    format long;
6
7    if(iflag == 2)
8      in=fopen('grain_25.inp','r');
9      end
10
11    %-----
12    % generate two grains
13    %-----
```

```

13 if(iflag == 1)                               61 % read the data generated from
14                                         voroni_1.m
15 ngrain =2;
16
17 %-etas(, 1) first grain
18 %-etas(, 2) second grain
19
20 x0 = Nx/2;
21 y0 = Ny/2;
22
23 radius = 14.0;   % radius of second
24 grain
25
26 for i=1:Nx
27 for j=1:Ny
28 ii=(i-1)*Nx+j;
29
30 if(isolve == 2)
31 etas(ii,1)=1.0;
32 etas(ii,2)=0.0;
33 else
34 etas(i,j,1)=1.0;
35 etas(i,j,2)=0.0;
36 end
37
38 xlength=sqrt((i-x0)^2+(j-y0)^2);
39 if(xlength <= radius);
40
41 if(isolve == 2)
42 etas(ii,1)=0.0;
43 etas(ii,2)=1.0;
44 else
45 etas(i,j,1)=0.0;
46 etas(i,j,2)=1.0;
47 end
48
49 end %if
50 end %j
51 end %i
52
53 end %iflag
54
55 %-----
56 % generate polycrystal
57 %microstructure %
58 if(iflag == 2)
59
60 %-----                               61 %-----%
62 %-----                               63
63                                         twopi=8.0*atan(1.0);
64 epsilon=1.0e-4;
65 ndime=2;
66
67                                         nvpoint=fscanf(in,'%d',1);
68 nvnode=fscanf(in,'%d',1);
69 nvelem=fscanf(in,'%d',1);
70 ngrain=fscanf(in,'%d',1);
71
72 for ipoin=1:nvpoint
73 jpoint=fscanf(in,'%d',1);
74 dummy=fscanf(in,'%lf %lf',[2,1]);
75 for idime=1:ndime
76 vcord(jpoint,idime)=dummy(idime);
77 end
78 end
79 end
80
81 for ielem=1:nvelem
82 jelem=fscanf(in,'%d',1);
83 dummy=fscanf(in,'%d',
84 [nvnode+1,1]);
85 for inode=1:nvnode+1
86 vlnods(ielem,inode)=dummy(inode);
87 end
88
89 %-----%
90 for ielem=1:nvelem
91
92 jnode=0;
93 for inode=1:nvnode
94 knode=vlnods(ielem,inode);
95 if(knode ~= 0)
96 jnode = jnode +1;
97 end
98 end
99 nnodes2(ielem)=jnode;
100 end
101
102 %-----%
103 % form the grid
104 %-----%
105 for i=1:Nx
106 gx(i)=i*dx;
107 end
108 for j=1:Ny

```

```

109 gy(j)=j*dy;
110 end
111
112 %-----
113 % initialize order parameters
114 %-----
115 for i=1:Nx
116 for j=1:Ny
117 for igrain=1:ngrain
118
119 if(isolve==2)
120 ii=(i-1)*Nx+j;
121 etas(ii,igrain)=0.0;
122 else
123 etas(i,j,igrain)=0.0;
124 end
125
126 end%igrain
127 end%j
128 end%i
129
130 %--
131 %--
132
133 for i=1:Nx
134 for j=1:Ny
135
136 ii=(i-1)*Nx+j;
137
138 for ielem=1:nvelem
139
140 igrain=vlnods(ielem,nvnode+1);
141
142 theta=0.0;
143
144 mnode=nnode2(ielem);
145
146 for inode=1:mnode
147
148 knode=vlnods(ielem,inode);
149
150 xv1=vcord(knode,1);
151 yv1=vcord(knode,2);
152
153 jnode=vlnods(ielem,inode+1);
154 if(inode==mnode)
155 jnode=vlnods(ielem,1);
156 end
157
158 xv2=vcord(jnode,1);
159 yv2=vcord(jnode,2);
160
161 p1x=xv1-gx(i);
162 p1y=yv1-gy(j);
163
164 p2x=xv2-gx(i);
165 p2y=yv2-gy(j);
166
167 x1=sqrt(p1x*p1x+p1y*p1y);
168 x2=sqrt(p2x*p2x+p2y*p2y);
169
170 if(x1*x2 <= epsilon)
171 theta=twopi;
172 else
173 tx1=((p1x*p2x+p1y*p2y)/(x1*x2));
174 end
175
176 if(abs(tx1)>=1.0)
177 tx1=0.9999999999;
178 end
179
180 theta=theta+acos(tx1);
181
182 end%inode
183
184 if(abs(theta-twopi)<=epsilon)
185
186 if(isolve==2)
187 etas(ii,igrain)=1.0;
188 else
189 etas(i,j,igrain)=1.0;
190 end
191
192 end
193
194 end%ielem
195
196 end%i
197 end%j
198
199 %-----
200
201 end%iflag
202
203 %--- initialize glist:
204
205 for igrain=1:ngrain
206 glist(igrain)=1.0;

```

```
207 end
208
209 end %endfunction
```

Line numbers:

5–7:	If polycrystalline simulation (iflag = 2) open input file, <i>grain_25.inp</i> .
9–53:	Bicrystal simulation.
20–21:	Center coordinates of the simulation cell.
23:	Radius of the second grain.
30–33:	If the simulation will be done with the optimized code, initialize order parameter of first grain as <i>etas(NxNy,1)</i> as one and second grain, <i>etas(NxNy,2)</i> , as zero.
33:	Else, simulation will be done with the un-optimized code, then initialize the order parameters of grains <i>etas(Nx,Ny,1)</i> as one and <i>etas(Nx,Ny,2)</i> as zero.
38–49:	Check if the current grid point is inside or on the boundary of the second grain. If it is, then, change the order parameters of the grains for this grid point.
55–201:	Generate order parameters for a polycrystalline microstructure.
64:	Value of 2π .
65:	A small constant value.
68–88:	Read voroni data from the file.
68:	Number of voroni points.
69:	Number of voroni nodes per voroni element.
70:	Number of voroni elements.
71:	Number of grains.
73–79:	Read voroni point coordinates.
81–87:	Read voroni element connectivity list.
90–100:	For each voroni element calculate the number of nodes in the connectivity list.
102–110:	For given number of grid points and spacing, form the <i>x</i> and <i>y</i> Cartesian coordinates of the grid points.
113–129:	Initialize the order parameters of grains, depending on the solution algorithm, if optimized code (isolve = 2) the common array takes the form <i>etas(NxNy,ngrain)</i> , otherwise <i>etas(Nx,Ny,ngrain)</i> .
133–197:	For each grid points in the simulation cell, loop over each grain and check that if grid point inside or edges of that grain, assign the grain number to that grain and initialize its order parameter as one. The routine is based on well-known <i>point inside the box algorithm</i> .
203–207:	Initialize <i>glist</i> array with ones.

Function

free_energ_fd_ca_v1.m

This function evaluates the derivative of free energy for the grain that is under consideration at the current grid point in the main program.

Variable and array list:

i:	Index of current grid point in the <i>x</i> -direction.
j:	Index of current grid point in the <i>y</i> -direction.
ngrain:	Number of grains in the solution.
igrain:	Current grain number under consideration in the main program.
dfdeta:	The derivative of the free energy for the current grid point.
etas(Nx,Ny, ngrain):	Common array for order parameters of the grains.
eta(Nx,Ny):	Order parameters of current grain under consideration in the main program.

Listing:

```
1 function [dfdeta] =
2   free_energ_fd_ca_v1(i,j,ngrain,
3     etas,eta,igrain)
4
5   format long;
6
7   A=1.0;
8   B=1.0;
9
10  sum = 0.0;
11
12  for jgrain=1:ngrain;
13
14    if(jgrain ~= igrain)
15      sum = sum +etas(i,j,jgrain)^2;
16    end
17
18  dfdeta = A*(2.0*B* eta(i,j)*sum
19            +eta(i,j)^3 - eta(i,j));
20 end %endfunction
```

Line numbers:

5–6:	Constants in the free energy (Eq. 4.33).
8–16:	Summation in the free energy function (Eq. 4.33).
18:	The derivative of the free energy for the grid point that is under consideration in the main program.

```

15 end
16
17 end
18
19 dfdeta = A*(2.0*B* eta .*sum
+eta.^3 - eta);
20
21 end %endfunction

```

Function**free_energ_fd_ca_v2.m**

This function evaluates the derivative of free energy for the grain that is under consideration simultaneously at all the grid points.

Variable and array list:

Nx:	Number of grid points in the <i>x</i> -direction.
Ny:	Number of grid points in the <i>y</i> -direction.
ngrain:	Number of grains in the solution.
igrain:	Current grain number under consideration in the main program.
dfdeta (NxNy):	The derivative of the free energy for all grid points in the simulation cell.
etas(NxNy, ngrain):	Common array for order parameters of the grains.
eta(NxNy):	Order parameters of current grain under consideration in the main program.

Listing:

```

1 function [dfdeta] = free_
energ_fd_ca_v2 (Nx,Ny,ngrain,
etas,eta,igrain)
2
3 format long;
4
5 A=1.0;
6 B=1.0;
7
8 NxNy = Nx*Ny;
9 sum=zeros (NxNy,1);
10
11 for jgrain=1:ngrain;
12
13 if(jgrain ~= igrain)
14 sum = sum +etas (:,jgrain).^2;

```

Line numbers:

5–6:	Constants in the free energy (Eq. 4.33).
8:	Total number of grid points in the simulation.
9–17:	Summation in the free energy function (Eq. 4.33) for all grid points.
19:	The derivative of the free energy for all grid points in the simulation cell.

References

- Atkinson HV (1988) Overview no 65: theories of normal grain growth in pure single phase systems. Acta Metall 36:469
- Blikstein P, Tschiatschin AP (1999) Monte Carlo simulation of grain growth. Mater Res 2:133
- Wakai F, Enomoto N, Ogawa H (2000) Three-dimensional microstructural evolution in ideal grain growth general statistics. Acta Mater 48:1297
- Frost HJ, Thompson CV (1996) Computer simulation of grain growth. Curr Opin Solid State Mater Sci 1:361
- Weygand D, Brechet Y, Lepinoux A (1998) A vertex dynamics simulation of grain growth in two dimensions. Philos Mag B 78:329
- Liu Y, Baudin T, Penelle R (1997) Simulation of normal grain growth by cellular automata. Scr Mater 34:1679
- Chen LQ, Yang W (1994) Phys Rev B 50:15752
- Fan D, Chen LQ (1997) Computer simulation of grain growth using a continuum field model. Acta Mater 45:611
- Krill CE III, Chen LQ (2002) Computer simulation of 3D grain growth using a phase-field model. Acta Mater 50:3059
- Serezende JLL (1996) Physica D 94:135
- Kobayashi R, Warren JA, Carter WC (2000) Physica D 140:141
- Lobkovsky AE, Warren JA (2001) Phys Rev E 63:051605
- Burke JE, Turnbull D (1952) Recrystallization and grain growth. Prog Metal Phys 3:220

4.6 Case Study-III

Phase-field modeling of solid state sintering

Objectives:

The objective of this case study is to demonstrate a numerical implementation of the finite difference algorithm for the coupled solution of the conserved Cahn–Hilliard and multiple non-conserved Allen–Cahn equations in a phase-field model.

4.6.1 Background

Sintering is widely used material processing technique for converting powder compacts into dense solids. Historically, it originates from processing of ceramics from ancient times. Today, it is almost the main production route for microelectronic packaging, production of nanostructures, and manufacturing of net-shape parts in automotive and aerospace industries [1].

Several mechanisms collectively and cooperatively operate during sintering. These include bulk diffusion through lattice, surface diffusion and diffusion along the grain boundaries, grain growth and grain boundary migration, vapor transport (evaporation and condensation), and finally rigid translation and rotation of particles. Operation of these mechanisms results in two oppositely desired property, densification and grain growth, in sintered microstructures. Close control of the remaining porosity in densification and grain growth is essential to reach the desired mechanical and physical properties of sintered materials. To achieve these objectives, experiments not always sufficient alone, complementary and guiding modeling and simulation algorithms of sintering have been always desired.

Varity of modeling and simulation algorithms developed for sintering can be grouped into three categories. The sharp interface models solve the continuum diffusion, material flux equations by utilizing the finite difference or finite element techniques [2–8]. However, as any sharp

interface models, it becomes difficult and computationally inefficient as the complexity of the diffusion paths and the number of evolving interfaces increases in the simulations. Nevertheless, these early models have contributed significantly to the theoretical understanding of the sintering. Second sets of modeling approaches are stochastic in nature and based on Monte-Carlo algorithms [9–12]. The phase-field models, in varying degree of details, for simulation of sintering have also been developed for determination of sintering rates, microstructure evolution (pore shrinkage and grain growth) [13–17].

4.6.2 Phase-Field Model

The phase-field model chosen for this case study is based on the early work of Wang [13]. The references cited above also follow the work of Wang only differing in some details in formulation and solution algorithms of resulting phase-field equations.

The model is based on the representation of the microstructure with two types of field variables. The first one is the conserved density field, ρ , that takes the value of one at the solid phase and zero at the pores, changing rapidly at the solid–pore interface. The second is the non-conserved order parameter, η_i , which is used to distinguish the different particles in the microstructure. Again, it takes the value of one for a designated particle and the value of zero in other particles. Also, the non-conserved order parameters changes from one to zero, or zero to one, smoothly at the evolving grain boundaries. The representation of the microstructure with these field variables, the free energy functional of the system is described by,

$$F = \int_V \left[f(\rho, \eta_1, \dots, \eta_p) + \frac{\kappa_\rho}{2} (\nabla \rho)^2 + \sum_i \frac{\kappa_\eta}{2} (\nabla \eta_i)^2 \right] dv \quad (4.38)$$

where κ_ρ and κ_η are the gradient energy coefficients for concentration and grain boundary energies, respectively.

The chemical free energy function $f(\rho, \eta_{1..p})$ in Eq. 4.38 is approximated by a Landau type polynomial potential:

$$f(\rho, \eta_{1..p}) = A\rho^2(1-\rho)^2 + B \left[\rho^2 + 6(1-\rho) \sum_i \eta_i^2 - 4(2-\rho) \sum_i \eta_i^3 + 3 \left(\sum_i \eta_i^2 \right)^2 \right] \quad (4.39)$$

where A and B are the constants. With this potential, the equilibrium value of mass density, $\rho = 1$, is reached at the solid particles and $\rho = 0$ at the pores. In addition, multicomponent non-conserved order parameters η_i also become zero at the pores.

The evolution equation for ρ follows the Cahn–Hilliard equation:

$$\frac{\partial \rho}{\partial t} = \nabla \cdot \left(D \nabla \frac{\delta F}{\delta \rho} \right) = \nabla \cdot D \nabla \left(\frac{\partial f}{\partial \rho} - \kappa_\rho \nabla^2 \rho \right) \quad (4.40)$$

where D is the microstructure-dependent diffusivity/mobility coefficient. Since, various diffusion paths take place during the course of sintering, D is assumed takes the form given below:

$$D = D_{vol}\varphi(\rho) + D_{vap}[1 - \varphi(\rho)] + D_{surf}\rho(1 - \rho) + D_{GB} \sum_i \sum_{i \neq m} \eta_i \eta_m \quad (4.41)$$

where D_{vol} is the bulk diffusivity in the lattice, D_{vap} is the diffusivity of the vapor phase, D_{surf} is the surface diffusivity, and D_{GB} is the grain boundary diffusivity. The interpolation function, $\varphi(\rho)$, in the equation is taken as:

$$\varphi = \rho^3(10 - 15\rho + 6\rho^2) \quad (4.42)$$

which ensures that the bulk diffusivity is zero at the pores and maximum at the solid regions. Equation 4.41 automatically accounts different diffusion mechanisms without explicitly tracking the surfaces and the grain boundaries.

The evolution of the non-conserved order parameters η_i representing the particles and

grain boundaries follows the Allen–Cahn equation:

$$\frac{\partial \eta_i}{\partial t} = -L \frac{\delta F}{\delta \eta_i} = -L \left(\frac{\partial f}{\partial \eta_i} - \kappa_\eta \nabla^2 \eta_i \right) \quad (4.43)$$

where L is the grain boundary mobility.

4.6.3 Numerical Implementation

The Laplace operator and the dimensional derivatives in Eqs. 4.40 and 4.43 are approximated with finite difference algorithm by utilizing five-point stencil in two-dimensional space. The time integration is carried out by simple explicit Euler time marching scheme. The nondimensionalized values of the parameters that appear in the model are given in Table 4.1.

4.6.4 Results and Discussion

The simulations were carried out for a square domain having number of grid points along the x -direction $N_x = 100$ and along the y -direction $N_y = 100$. The grid spacing between two grid points taken as $dx = dy = 0.5$. The nondimensionalized parameters given in Table 4.1 were used in the solutions.

Table 4.1 The nondimensional values of the parameters used in the simulations

D_{vol}	D_{vap}	D_{surf}	D_{GB}	κ_ρ	κ_η	L
0.040	0.002	16.0	1.6	5.0	2.0	10.0

The first set of simulation was carried out to illustrate the neck and grain boundary development between two unequal sized and initially non-contacting spherical particles. For this case, the time evolution of the sintering is summarized in Fig. 4.12. As can be seen, the neck and the grain boundary formation proceed quite rapidly initially, later their growth slows down and almost reaches to a steady state. Through the later stages the size of the neck almost remains constant. The changes in the size and shape of the particles can be clearly seen from the figure. This is of course as result of the transport of the matter to the neck region between the two particles with the various diffusion paths given by Eq. 4.41. The faster shrinkage of the smaller particle and slow migration of the grain boundary, in agreement with the earlier studies, is also noticeable in the figure. If the simulations were carried for much longer period, the small particle and the grain boundary that formed between the two particles eventually disappear from the system.

In the second set of simulations the effects of the various diffusion parameters on the sintering kinetics is elucidated. For this purpose, a circular pore having radius $R = 10$ (in terms of grid numbers) is placed between the two crystals at the beginning of the simulation. The change in the volume fraction of the pore was also recorded during the course of the simulations. In the first simulation the parameters were the same as given in Table 4.1. In the second simulation, the grain boundary diffusivity is increased by a factor of five to $D_{GB} = 8.0$ while the other diffusivity parameters were kept the same.

In the last simulation, the surface diffusivity is increased by a factor of two, $D_{surf} = 32.0$, and the others were again the same as in Table 4.1. For these cases, the evolution of pore at first 200 time steps and after 20,000 final steps is shown in Fig. 4.13. As can be seen, the pore shape changes from initial spherical shape to a lenticular shape owing to contact with the grain boundary. Under the equilibrium conditions, the contact angle (dihedral angle) is about 72° . The densification, resulting from the shrinkage of the pore, for all cases is shown in Fig. 4.14. As can be seen, also from Fig. 4.13, the surface diffusion

appears to be larger contributing factor to the densification as predicted from earlier studies.

Finally, the sintering behavior of multiple unequal sized particles is shown in Fig. 4.15. As given earlier, the sintering proceeds with the formation of necks and grain boundaries between the particles and densification takes place by elimination of porosity between the particles with time. Again, the evolution of pores to lenticular shapes as grain boundaries develop between the particles can be seen from the figure.

The movies resulting from these simulations can be found in subdirectory *case_study_3*.

4.6.5 Source Codes

Two programs are provided in this section. The first one, *fd_sint_v1.m*, is in longhand format and the second one, *fd_sint_v2.m*, is the optimized version for Matlab/Octave.

Program

fd_sint_v1.m

This program solves the phase-field modeling of sintering with finite difference algorithm. The code is in longhand format and not optimized for Matlab/Octave.

The program makes calls to the following functions:

- **micro_sint_pre.m**
- **free_energ_sint_v1.m**
- **write_vtk_grid_values.m**

Listing:

```

1 %%%%%%%%%%%%%%%%
2 %
3 % FINITE-DIFFRENCE PHASE-FIELD %
4 % CODE FOR SINTERING %
5 %
6 %
7 %% get intial wall time:
8 time0=clock();

```

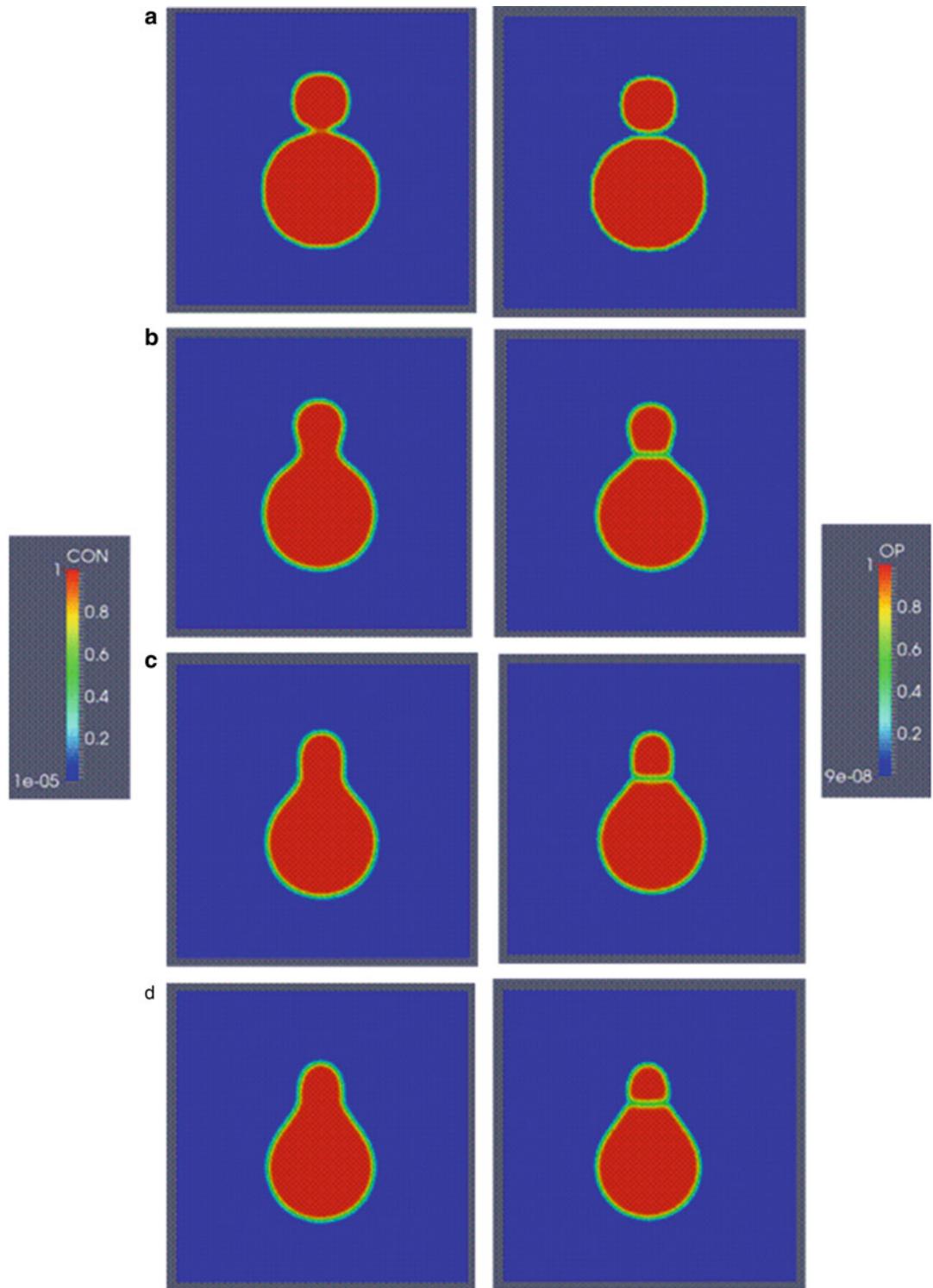


Fig. 4.12 The neck and grain boundary evolution in sintering two unequal sized particles. The left column is concentration field and the right column is non-conserved

order parameters. (a) At time step 100, (b) at time step 5000, (c) at time step 12,500, and (d) at time step 20,000

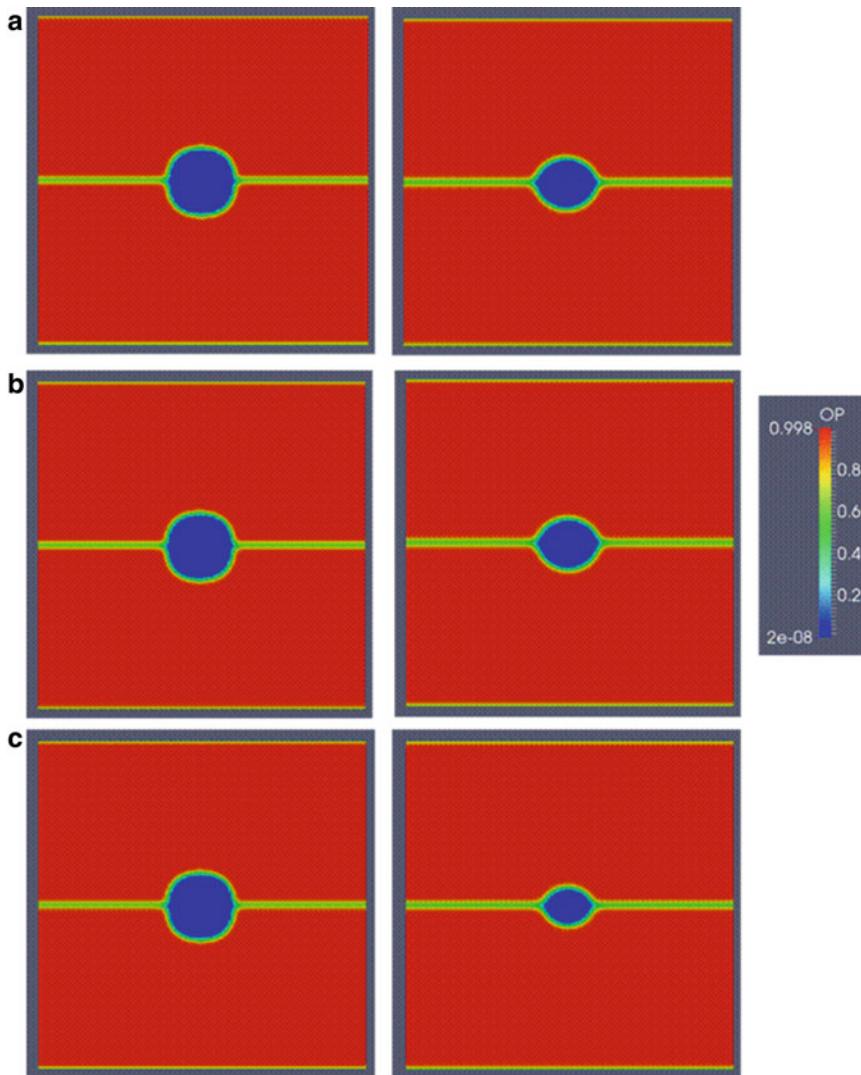


Fig. 4.13 The shrinkage of a pore between a bicrystal during sintering with different diffusion parameters.
(a) For diffusion parameters are the same as in Table 4.1,
(b) is for $D_{GB} = 8.0$, and the others the same as in the

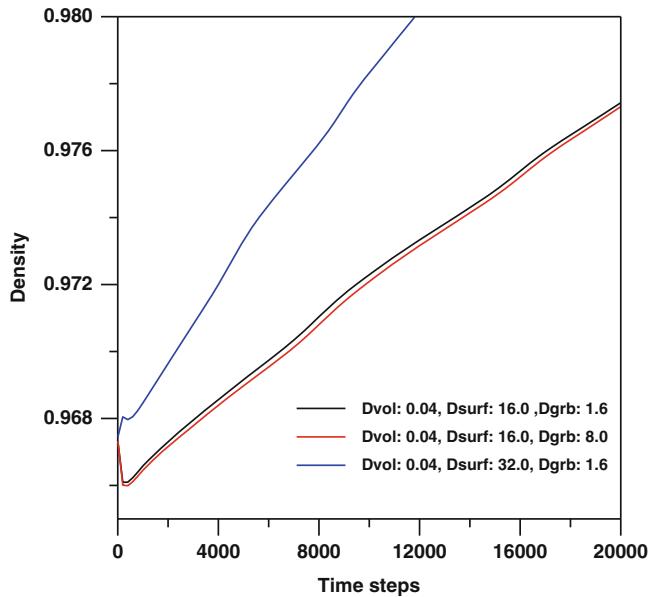
table, and **(c)** is for $D_{surf} = 32.0$ and others the same as in the table. First column is after 100 time steps, and second column is after 20,000 time steps

```

9    format long;
10
11    %-- Simulation cell parameters:
12
13    Nx = 100;
14    Ny = 100;
15    NxNy= Nx*Ny;
16
17    dx = 0.5;
18    dy = 0.5;
19
20    %--- Time integration parameters:
21
22    nstep =      5000;
23    nprint =     50;
24    dtime= 1.0e-4;
25
26    %--- Material specific Parameters:
27    npart = 2;
28    %

```

Fig. 4.14 The evolution of density as result of shrinkage of the pore in a bicrystal with different diffusivity parameters



```

29  coefm = 5.0;
30  coefk = 2.0;
31  coefl = 5.0;
32  %
33  dvol = 0.040;
34  dvap = 0.002;
35  dsur = 16.0;
36  dgrb = 1.6;
37  %----
38  %-----%prepare microstructure:
39  %-----%
40  iflag =1;
41  [npart,etas,con] =micro_sint_pre
42  (Nx,Ny,npart,iflag);
43
44  %-- initialize eta:
45  for i=1:Nx
46  for j=1:Ny
47  eta(i,j) = 0.0;
48  end
49  end
50
51  for istep =1:nstep
52  %-- evolve concentration:
53
54
55
56  iflag =1;
57
58  for i=1:Nx
59  for j=1:Ny
60
61  jp=j+1;
62  jm=j-1;
63
64  ip=i+1;
65  im=i-1;
66
67  jp=j+1;
68  jm=j-1;
69
70  ip=i+1;
71  im=i-1;
72
73  if(im == 0)
74  im=Nx;
75  end
76  if(ip == (Nx+1))
77  ip=1;
78  end
79
80  if(jm == 0)
81  jm = Ny;
82  end
83

```

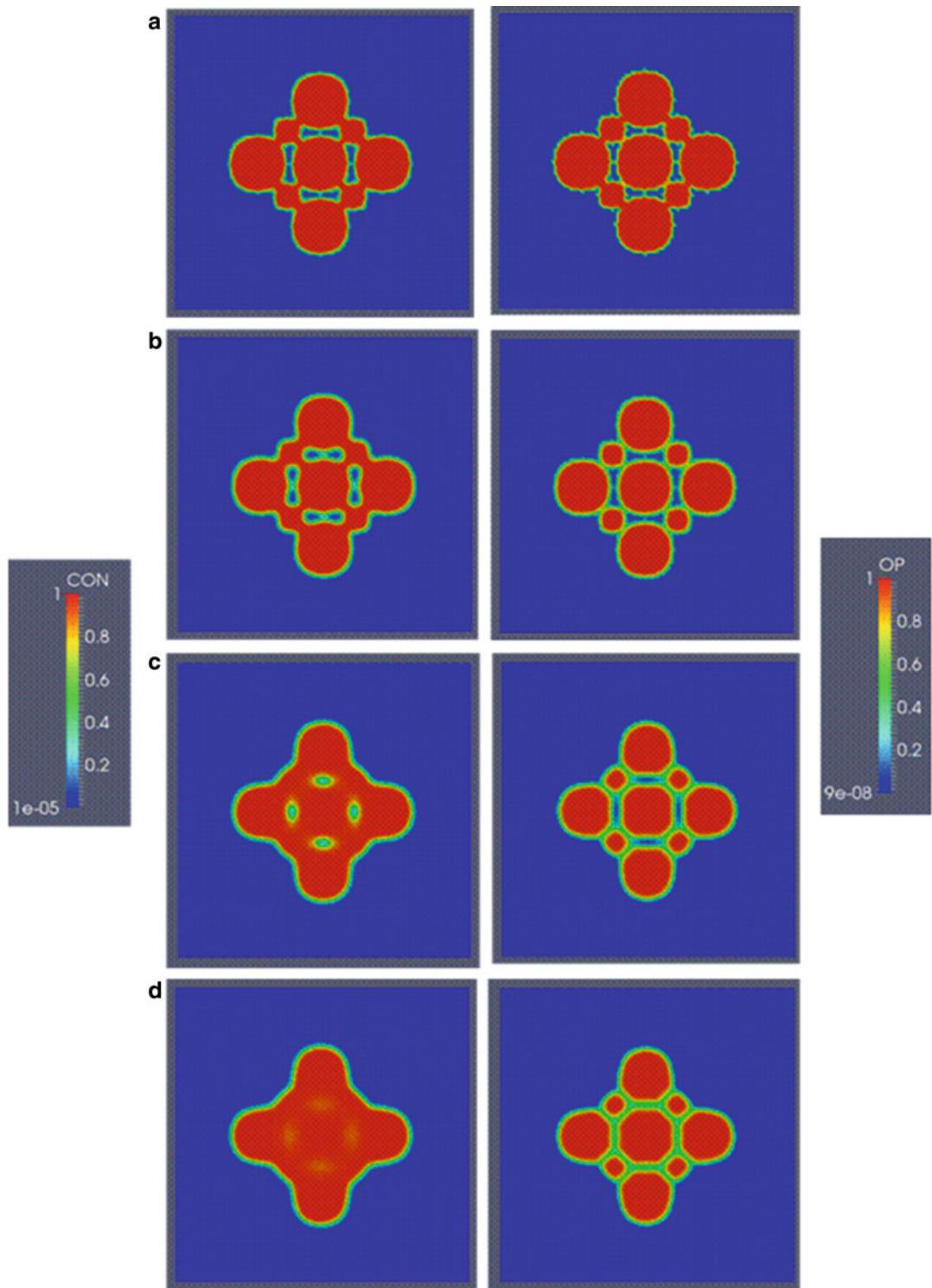


Fig. 4.15 Sintering of multiple unequal sized particles. The left column is concentration field and the right column is non-conserved order parameters. **(a)** At time step

250, **(b)** at time step 650, **(c)** at time step 1250, and **(d)** at time step 2500

```

84 if(jp == (Ny+1))           130 jm = Ny;
85 jp=1;                     131 end
86 end
87                                     132
88 hne=con(ip,j);             133 if(jp == (Ny+1))
89 hnw=con(im,j);             134 jp=1;
90 hns=con(i,jm);             135 end
91 hnn=con(i,jp);             136
92 hnc=con(i,j);              137 hne=dummy(ip,j);
93                                     138 hnw=dummy(im,j);
94 lap_con(i,j)=(hnw + hne + hns + 139 hns=dummy(i,jm);
95 hnn -4.*hnc)/(dx*dy);      140 hnn=dummy(i,jp);
96 %--- derivative of free energy: 141 hnc=dummy(i,j);
97                                     142
98 [dfdcon,dfdeta]=free_energ_ 143 lap_dummy(i,j)=(hnw + hne + hns +
99     sint_v1(i,j,con,eta,etas, 144 hnn -4.*hnc)/(dx*dy);
100    npart,iflag);            145 %-- Mobility:
101 dummy(i,j) = dfdcon -      146
102   0.5*coefm*lap_con(i,j); 147 phi = con(i,j)^3 *(10.0-15.0*
103 end %for j                148   con(i,j) + 6.0*con(i,j)^2);
104 end %for i
105 %--                                149 sum=0.0;
106 for i=1:Nx                 150 for ipart =1:npart
107 for j=1:Ny                 151 for jpart =1:npart
108                                     152 if(ipart ~= jpart)
109                                     153 sum = sum + etas(i,j,ipart)*
110 jp=j+1;                   154 end
111 jm=j-1;                   155 end
112                                     156 end
113 ip=i+1;                   157
114 im=i-1;                   158 mobil = dvol*phi + dvap*(1.0-phi) +
115                                     159 dsur*con(i,j)*(1.0-con(i,j)) +
116 jp=j+1;                   160   dgrib*sum;
117 jm=j-1;                   161 %-- time integration:
118                                     162 con(i,j) = con(i,j) + dtimemobil*
119 ip=i+1;                   163   lap_dummy(i,j);
120 im=i-1;                   164 %-- for small deviations:
121                                     165
122 if(im == 0)                166 if(con(i,j) >= 0.9999);
123 im=Nx;                     167 con(i,j)= 0.9999;
124 end                         168 end
125 if(ip == (Nx+1))           169
126 ip=1;                      170 if(con(i,j) < 0.00001);
127 end                         171 con(i,j) = 0.00001;
128                                     172 end
129 if(jm == 0)                 173

```

```
174 end %j
175 end %i
176
177 %%-- evolve etas:
178
179 iflag =2;
180
181 for ipart =1:npart
182
183 for i=1:Nx
184 for j=1:Ny
185 eta(i,j) =etas(i,j,ipart);
186 end
187 end
188
189 for i=1:Nx
190 for j=1:Ny
191
192 jp=j+1;
193 jm=j-1;
194
195 ip=i+1;
196 im=i-1;
197
198 jp=j+1;
199 jm=j-1;
200
201 ip=i+1;
202 im=i-1;
203
204 if(im == 0)
205 im=Nx;
206 end
207 if(ip == (Nx+1))
208 ip=1;
209 end
210
211 if(jm == 0)
212 jm =Ny;
213 end
214
215 if(jp == (Ny+1))
216 p=1;
217 end
218
219 hne=eta(ip,j);
220 hnw=eta(im,j);
221 hns=eta(i,jm);
222 hnn=eta(i,jp);
223 hnc=eta(i,j);

224
225 lap_eta(i,j) =(hnw + hne + hns +
226 hnn -4.*hnc)/(dx*dy);
227 % --- derivative of free energy:
228
229 [dfdcon,dfdeta]=free_energ_
sint_v1(i,j,con,eta,etas,
npart,iflag);
230
231 %%-- Time integration
232
233 eta(i,j) =eta(i,j) - dtim* coefl*
(dfdeta - 0.5*coefk*lap_eta(i,j));
234
235 %%-- for small deviations:
236
237 if(eta(i,j) >= 0.9999);
238 eta(i,j) = 0.9999;
239 end
240
241 if(eta(i,j) < 0.0001);
242 eta(i,j) = 0.0001;
243 end
244
245 end %j
246 end %i
247
248 %%-
249
250 for i=1:Nx
251 for j=1:Ny
252 etas(i,j,ipart) =eta (i,j);
253 end
254 end
255
256 end %ipart
257
258 %---- print results
259
260 if((mod(istep,nprint) == 0) ||
(istep == 1) )
261
262 fprintf('done step: %5d\n',istep);
263
264 %fname1 =sprintf('time_%d.out',
istep);
265 %out1 = fopen(fname1,'w');
266
267 %for i=1:Nx
```

```

268 %for j=1:Ny
269
270 %sum = 0.0;
271 %for ipart=1:npart
272 %sum = sum +etas(i,j,ipart)^2;
273 %end
274 fprintf(out1,'%5d %5d %14.6e %
14.6e\n',i,j,con(i,j),sum);
275 %end
276 %end
277
278 %fclose(out1);
279
280
281 %--- write vtk file:
282
283 phi2=zeros(Nx,Ny);
284
285 for i=1:Nx
286 for j=1:Ny
287
288 for ipart=1:npart
289
290 phi2(i,j) = phi2(i,j) +etas(i,j,
ipart)^2;
291 end
292 end
293 end
294 end
295 end
296
297
298 write_vtk_grid_values(Nx,Ny,dx,
dy,istep,con,phi2);
299
300 end %end if
301
302 end %istep
303
304 %--- calculate compute time:
305
306 compute_time = etime(clock(),
time0);
307 fprintf('Compute Time: %10d\n',
compute_time);
308

```

Line numbers:

7–8:	Get wall clock time at the beginning of execution.
11–19:	Simulation cell parameters.
13:	Number of grid points in the x -direction.
14:	Number of grid points in the y -direction.
15:	Total number of grid points in the simulation cell.
17:	Grid spacing between two grid points in the x -direction.
18:	Grid spacing between two grid points in the y -direction.
20–25:	Time integration parameters.
22:	Number of integration steps.
23:	Output frequency to write the results to file.
24:	Time increment for numerical integration.
26–37:	Model-specific parameters (see Eqs. 4.40, 4.43 and Table 4.1).
27:	Number of particles in the simulation, set to either 2 or 9.
29:	Gradient coefficient for the concentration field, Eq. 4.38.
30:	Gradient coefficient for the order parameters. Eq. 4.38.
31:	Mobility of the order parameters, Eq. 4.43.
33:	Bulk diffusion coefficient.
34:	Vapor diffusion coefficient.
35:	Surface diffusion coefficient.
36:	Grain boundary diffusion coefficient.
39–44:	Initialize concentration and order parameters and define the microstructure in the simulation cell.
41:	iflag = 1 for the un-optimized code and iflag = 2 for the optimized code.
45–51:	Initialize temporary array eta(Nx,Ny) for the time integration of the order parameters.
52–303:	Time evolution of concentration and order parameters.
54–175:	Time evolution of concentration field.
56:	Set iflag = 1 for the functional derivative of free energy.
58–103:	Calculate the laplacian of the term inside the parenthesis in Eq. 4.40.
94:	Calculate the laplacian of the concentration.
96–99:	Functional derivative of free energy respect to concentration at the current point.
100:	Calculate the terms inside parenthesis in Eq. 4.40.
143:	Calculate the laplacian of the terms inside the parenthesis in Eq. 4.40.

(continued)

145–159:	Calculate the diffusivity/mobility parameter for the current grid point.
147:	Calculate the value of interpolation function, Eq. 4.42.
149–156:	Calculate the summations in Eq. 4.41.
158:	The value of the diffusivity/mobility parameter at the grid point.
160–163:	Explicit Euler time integration of the concentration field, Eq. 4.40 for the current grid point.
164–175:	If there are small deviations, set the max and min values to the limits.
177–256:	Evolve the order parameters.
179:	Set iflag = 2 for functional derivative of free energy.
181–256:	Loop over the number of particles.
183–188:	Assign the current particle order parameter values from common array etas(Nx,Ny, npart) to eta(Nx,Ny).
225:	The Laplacian of the order parameter, Eq. 4.43.
227–230:	Functional derivative of the free energy for the current order parameter at the current grid point.
231–234:	Explicit Euler time integration of the order parameter, Eq. 4.43 for the current grid point.
235–243:	If there are small deviations, set the max and min values to limits.
250–254:	Return the current values of the order parameter, eta(Nx,Ny) to common array etas (Nx,Ny,npart).
258–300:	If print frequency is reached, print the results to file.
264–278:	Open and output file and the print the results. These lines are commented out, but they can be changes, including the format of the output file.
281–298:	Write the results in vtk format for contour plots to be viewed by Paraview.
304–307:	Calculate total execution time and print it to screen.

Program

fd_sint_v2.m

This program solves the phase-field modeling of sintering with finite difference algorithm. The code is optimized for Matlab/Octave.

The program makes calls to the following functions:

- **micro_sint_pre.m**
- **free_energ_sint_v2.m**
- **laplacian.m**
- **write_vtk_grid_values.m**

Listing:

```

1    %%%%%%%%
2    %
3    % FINITE-DIFFRENCE PHASE-FIELD %
4    % CODE FOR SINTERING %
5    % (OPTIMIZED FOR MATLAB/OCTAVE) %
6    %%%%%%%%
7    %% get intial wall time:
8    time0=clock();
9    format long;
10   %% Simulation cell parameters:
11   Nx = 100;
12   Ny = 100;
13   NxNy= Nx*Ny;
14
15   dx = 0.5;
16   dy = 0.5;
17
18   %% Time integration parameters:
19
20   nstep = 20000;
21   nprint = 100;
22   dtim= 1.0e-4;
23
24   %% Material specific Parameters:
25
26   npart = 2;
27
28   %
29   coefm = 5.0;
30   coefk = 2.0;
31   coefl = 10.0;
32
33   dvol = 0.040;
34   dvap = 0.002;
35   dsur = 16.0;
36   dgrb = 1.6;
37
38   %---
39   %prepare microstructure:
40   %---
```

```

41
42 iflag = 2;
43
44 [npart,etas,con] = micro_sint_pre
45 (Nx,Ny,npart,iflag);
46 %--- Get Laplacian templet:
47
48 [grad]=laplacian(Nx,Ny,dx,dy);
49
50 %---
51 %- Evolution
52 %---
53
54 eta=zeros(NxNy,1);
55
56 for istep=1:nstep
57
58 %-- evolve concentration:
59
60 iflag=1;
61
62 %-- derivative of free energy:
63
64 [dfdcon,dfdeta]=
65 free_energ_sint_v2(Nx,Ny,con,eta,
66 etas,npart,iflag);
67
68 %-- Mobility:
69
70 phi=zeros(NxNy,1);
71
72 phi=con.^3.* (10.0-15.0*con +
73 6.0*con.^2);
74
75 sum=zeros(NxNy,1);
76 for ipart=1:npart
77 for jpart=1:npart
78 if(ipart ~= jpart)
79 sum = sum + etas( :,ipart).*etas( :,
80 jpart);
81 end
82 end
83 mobil=dvol*phi + dvap*(1.0-phi) +
84 dsur*con.* (1.0-con) + dgrb*sum;
85 %-- time integration:
86
87 lap_con=grad*con;
88
89 lap_con2=grad*(dfdcon
90 -0.5*coefm*lap_con);
91 con=con+dtime*mobil.*lap_con2;
92
93 %-- for small deviations:
94
95 inrange=(con >= 0.9999);
96 con(inrange)=0.9999;
97 inrange=(con < 0.00001);
98 con(inrange)=0.00001;
99
100
101 %-- evolve etas:
102
103 iflag=2;
104
105 for ipart=1:npart
106 eta(:)=etas( :,ipart);
107
108 [dfdcon,dfdeta]=
109 free_energ_sint_v2(Nx,Ny,con,eta,
110 etas,npart,iflag);
111 eta=eta-dtime * coefl*(dfdeta -
112 0.5 *coefk*grad*eta);
113
114 %-- for small deviations:
115
116 inrange=(eta >= 0.9999);
117 eta(inrange)=0.9999;
118 inrange=(eta < 0.0001);
119 eta(inrange)=0.0001;
120
121 etas( :,ipart)=eta( :);
122 end
123
124
125 %--- print results
126
127 if((mod(istep,nprint)==0) ||
128 (istep==1))

```

```

129 fprintf('done step: %5d\n',istep);
130
131 %fname1 = sprintf('time_%d.out',
132 istep);
133 %out1 = fopen(fname1,'w');
134 %for i=1:Nx
135 %for j=1:Ny
136 %ii =(i-1)*Nx+j;
137
138 %sum = 0.0;
139 %for ipart=1:npart
140 %sum = sum +etas(ii,ipart)^2;
141 %end
142 %fprintf(out1,'%5d %5d %14.6e %
14.6e\n',i,j,con(ii),sum);
143 %end
144 %end
145
146 %fclose(out1);
147
148
149 %--- write vtk file:
150
151 phi2=zeros(Nx,Ny);
152
153 for i=1:Nx
154 for j=1:Ny
155 ii =(i-1)*Nx+j;
156
157 phi1(i,j) =con(ii);
158
159 for ipart=1:npart
160
161 phi2(i,j) =phi2(i,j) +etas(ii,
162 ipart)^2;
163 end
164
165 end
166 end
167
168
169 write_vtk_grid_values(Nx,Ny,dx,
170 dy,istep,phi1,phi2);
171 end %end if
172
173 end %istep
174
175 %--- calculate compute time:
176
177 compute_time = etime(clock(),
178 time0);
179 fprintf('Compute Time: %10d\n',
179 compute_time);

```

Line numbers:

7–8:	Get wall clock time at the beginning of execution.
11–19:	Simulation cell parameters.
13:	Number of grid points in the <i>x</i> -direction.
14:	Number of grid points in the <i>y</i> -direction.
15:	Total number of grid points in the simulation cell.
17:	Grid spacing between two grid points in the <i>x</i> -direction.
18:	Grid spacing between two grid points in the <i>y</i> -direction.
20–25:	Time integration parameters.
22:	Number of integration steps.
23:	Output frequency to write the results to file.
24:	Time increment for numerical integration.
26–37:	Model-specific parameters (see Eqs. 4.40 and 4.43, Table 4.1).
27:	Number of particles in the simulation, set to either 2 or 9.
29:	Gradient coefficient for the concentration field, Eq. 4.38.
30:	Gradient coefficient for the order parameters. Eq. 4.38.
31:	Mobility of the order parameters, Eq. 4.43.
33:	Bulk diffusion coefficient.
34:	Vapor diffusion coefficient.
35:	Surface diffusion coefficient.
36:	Grain boundary diffusion coefficient.
39–44:	Initialize concentration and order parameters, and define the microstructure in the simulation cell.
46–49:	Calculate the finite difference template for the Laplacian.
51–173:	Time evolution of concentration and order parameters.
54	Initialize the temporary array eta(NxNy), for time integration of order parameters.
58–99:	Evolve the concentration field.
60:	Set iflag = 1 for functional derivative of free energy.

(continued)

62–65:	Derivative of free energy for the concentration field at all grid points in the simulation cell.
66–84:	Calculate the mobility values at all grid points.
68–70:	Calculate the values of the interpolation function at all the grid points, Eq. 4.42.
72–77:	Calculate summations in Eq. 4.41 for all the grid points.
83:	The values of diffusivity/mobility parameter at all the grid points.
85–92:	Time integration of the concentration field at all the grid points.
87:	Calculate the Laplacian of the concentration field, Eq. 4.40.
89:	Calculate the Laplacian of the parenthesis, in Eq. 4.40.
91:	Explicit Euler time integration of the concentration, Eq. 4.40, at all grid points.
93–99:	If there are small deviations, set the max and min values to the limits.
101–123:	Evolve the order parameters.
103:	Set iflag = 2 for functional derivative of free energy function.
105–122:	Loop over the number of particles.
107:	Assign current particle order parameter values from common array etas(NxNy, npart).
109:	The derivative values of the free energy for the current order parameter at the all grid points.
111:	Explicit Euler time integration of the current order parameter, Eq. 4.43, for all the grid points.
114–119:	If there are small deviations, set the max and min values to the limits.
121:	Return the current order parameter values to common array etas(NxNy, npart).
127–171:	If print frequency is reached, print the results to file.
131–147:	Open and output file and the print the results. These lines are commented out, but they can be changes, including the format of the output file.
149–169:	Write the results in vtk format for contour plots to be viewed by Paraview.
177–178:	Calculate total execution time and print it to screen.

Function

micro_sint_pre.m

This function initializes the concentration and the order parameters in the simulation cell and

defines the particle positions for either two particles or nine particle simulations. It is used in both longhand (iflag = 1) and Matlab/Octave optimized code (iflag = 2).

Variable and array list:

Nx:	Number of grid points of the simulation cell in the x -direction.
<td>Number of grid points of the simulation cell in the y-direction.</td>	Number of grid points of the simulation cell in the y -direction.
npart:	Number of particles in the simulation.
iflag:	iflag = 1 for longhand code, iflag = 2 for optimized code.
etas(Nx,Ny, npart):	Common array of non-conserved order parameters for the particles. In the optimized code, this array is in the form of etas(NxNy,npart) where, NxNy is the total number of grid points in the simulation cell.
con(Nx,Ny):	Concentration field it takes value of one in the particles and zero elsewhere. In the optimized code, this is one-dimensional array in the form of con(NxNy).

Listing:

```

1  function [npart,etas,con] =
2    micro_sint_pre(Nx,Ny,npart,iflag)
3    format long;
4    %--- initialize:
5    if(iflag == 1)
6      for ipart =1:npart
7        for i=1:Nx
8          for j=1:Ny
9            con(i,j) =0.0 ;
10           etas(i,j,ipart) = 0.0 ;
11         end
12       end
13     end
14   end
15 end % ipart
16 end % if
17
18 if(iflag == 2)
19
20 NxNy = Nx*Ny;
21 con =zeros(NxNy,1);
22 etas=zeros(NxNy,npart);
23 end
24
25 %---
```

```
27 if(npart ~= 2)                                76 con(ii) = 0.999;
28                                         77 etas(ii,ipart) =0.999;
29 R = 10.0;                                     78 else
30                                         79 con(i,j)= 0.999;
31 xc(1)=29.0;                                  80 etas(i,j,ipart) =0.999;
32 yc(1)=50.0;                                  81 end
33                                         82
34 xc(2)=50.0;                                  83 end %if
35 yc(2)=50.0;                                  84
36                                         85 end % j
37 xc(3)=71.0;                                  86 end % i
38 yc(3)=50.0;                                  87
39                                         88 end % ipart
40 xc(4)=50.0;                                  89
41 yc(4)=29.0;                                  90 end % if
42                                         91
43 xc(5)=50.0;                                  92 %---
44 yc(5)=71.0;                                  93
45                                         94 if(npart == 2)
46 xc(6)=39.0;                                  95
47 yc(6)=39.0;                                  96 R1 = 20.0;
48                                         97 R2 = 0.5*R1;
49 xc(7)=61.0;                                  98
50 yc(7)=39.0;                                  99
51                                         100 x1 = Nx/2;
52 xc(8)=39.0;                                  101 y1 = 40.0; ;
53 yc(8)=61.0;                                  102 y2 = 70.0;
54                                         103
55 xc(9)=61.0;                                  104
56 yc(9)=61.0;                                  105 for i=1:Nx
57                                         106 for j=1:Ny
58 for ipart=1:npart                            107
59                                         108 ii =(i-1)*Nx+j;
60 Rx = R;                                     109
61                                         110 xx1 =sqrt((i-x1)^2 +(j-y1)^2);
62 if(ipart > 5 )                             111 xx2 =sqrt((i-x1)^2 +(j-y2)^2);
63 Rx = 0.5*R;                                 112
64 end                                         113 if( xx1 <= R1)
65                                         114
66 for i=1:Nx                                 115 if(iflag == 2)
67 for j=1:Ny                                 116 con(ii) = 0.999;
68                                         117 etas(ii,1) =0.999;
69 ii =(i-1)*Nx+j;                           118 else
70                                         119 con(i,j)=0.999;
71 xx1 =sqrt((i-xc(ipart))^2 +           120 etas(i,j,1) =0.999;
(j-yc(ipart))^2);                           121 end
72                                         122
73 if( xx1 <= Rx)                           123 end % if
74                                         124
75 if(iflag == 2)                           125 if(xx2 <= R2)
```

```

126
127 if(iflag == 2)
128
129 con(ii) = 0.999;
130 etas(ii,1) = 0.0;
131 etas(ii,2) = 0.999;
132 else
133 con(i,j) = 0.999;
134 etas(i,j,1)=0.0;
135 etas(i,j,2)=0.9999;
136 end
137
138 end % if
139
140 end % j
141 end % i
142
143 end %if
144
145 %---
146
147 end %end function

```

Line numbers:

5–24:	Initialize the concentration and order parameters.
7–16:	Initialization in longhand format.
18–24:	Initialization for Matlab/Octave optimized code.
27–90:	Place nine spherical particles into the simulation cell.
29:	The radius of the first five large particles.
31–56:	Center coordinates of the particles.
58–64:	Set the particle radius (first five are the large ones).
75–78:	If data for the optimized code, set the field variables as con(NxNy) and etas(NxNy, npart).
79–81:	Else, (for longhand format) set con(Nx,Ny) and etas(Nx,Ny,npart).
94–143:	Place two particles into the simulation cell.
96–97:	Radius of the particles.
100–102:	Center coordinates of the particles.
115–118:	If data for the optimized code, set the field variables as con(NxNy) and etas(NxNy, npart).
118–120:	Else, (for longhand format) set con(Nx,Ny) and etas(Nx,Ny,npart).
125–143:	Repeat the same procedure for the second particle.

Function

free_energ_sint_v1.m

This function provides the functional derivatives for concentration field and for order parameters at the current grid point under consideration in the main program.

Variable and array list:

i:	Current grid id in the x -direction.
j:	Current grid id in the y -direction.
iflag:	iflag = 1 derivative is provided for the concentration field, iflag = 2 the derivative is provided for the order parameter.
npart:	Number of particles in the simulation.
dfdcon:	The value of derivative of free energy function respect to concentration at the current grid point.
dfdeta:	The value of derivative of free energy function respect to current order parameter at the current grid point.
con(Nx,Ny):	Concentration field.
eta(Nx,Ny):	Order parameter that is under consideration in the main program.
etas(Nx,Ny, npart):	Common array for order parameters.

Listing:

```

1 function [dfdcon,dfdeta]
2   =free_energ_sint_v1(i,j,con,eta,
3     etas,npart,iflag)
4
5 A=16.0;
6 B= 1.0;
7
8 dfdcon =0.0 ;
9 dfdeta =0.0 ;
10
11
12 if(iflag == 1)
13 %--
14 %-- derivative of free energy for
15 %-- concentration:
16
17 sum2 = 0.0 ;
18 sum3 = 0.0 ;

```

```

19
20 for ipart = 1:npart
21
22 sum2 = sum2 + etas(i,j,ipart).^2;
23 sum3 = sum3 + etas(i,j,ipart).^3;
24 end
25
26 dfdcon = B*(2.0*con(i,j) + 4.*sum3 -
27   6.0*sum2) - 2.0*A*con(i,j)^2 .* ...
28     (1.0-con(i,j)) + 2.0*A*
29     con(i,j)*(1.0-con(i,j))^2;
30
31 end %if
32
33 if(iflag == 2)
34 %-
35 %% derivative of free energy
36 %% for etas:
37 %-
38 sum2 = 0.0;
39
40 for ipart = 1: npart
41 sum2 = sum2 + etas(i,j,ipart)^2;
42 end
43 dfdeta = B*(-12.0*eta(i,j)^2 .*
44   (2.0-con(i,j)) + 12.0 *eta(i,j)*
45   (1.0-con(i,j)) + ...
46   12.0*eta(i,j)*sum2);
47 end %if
48 end %endfunction

```

Line numbers:

5–6:	Set constants in free energy function, Eq. 4.39.
12–29:	Functional derivative of the free energy for the concentration at the current grid point.
20–24:	Summations in Eq. 4.39 for the current point.
26–27:	Functional derivative of the free energy respect to concentration at grid point (i,j).
31–45:	Functional derivative of the free energy for the current order parameter under consideration in the main program.
36–41:	Summations in Eq. 4.39.
43–44:	Functional derivative of the free energy for the current order parameter at current grid point (i,j).

Function

free_energ_sint_v2.m

This function provides the functional derivatives of concentration field and order parameter at all grid points in the simulation cell. It is optimized for Matlab/Octave.

Variable and array list:

Nx:	Total number of grid points in the x-direction.
<b b="" ny:<="">	Total number of grid points in the y-direction.
<b b="" iflag:<="">	iflag = 1 derivative is provided for the concentration field, iflag = 2 the derivative is provided for the order parameter.
<b b="" npart:<="">	Number of particles in the simulation.
<b (nxny):<="" b="" dfdcon="">	The value of derivative of free energy function respect to concentration at all grid points.
<b (nxny):<="" b="" dfdeta="">	The value of derivative of free energy function respect to current order parameter at all grid points.
<b b="" con(nxny):<="">	Concentration field.
<b b="" eta(nxny):<="">	Order parameter that is under consideration in the main program.
<b b="" etas(nxny,="" npart):<="">	Common array for order parameters.

Listing:

```

1 function [dfdcon,dfdeta] =
2   free_energ_sint_v2(Nx,Ny,con,eta,
3   etas,npart,iflag)
4
5 format long;
6
7 A=16.0;
8 B= 1.0;
9
10 NxNy =Nx*Ny;
11
12 dfdcon =zeros(NxNy,1);
13 dfdeta =zeros(NxNy,1);
14
15 if(iflag == 1)
16 %-
17 %% derivative of free energy
18 %% for concentration:

```

```

16 %--
17
18 sum2=zeros(NxNy,1);
19 sum3=zeros(NxNy,1);
20
21
22
23 for ipart = 1:npart
24
25 sum2 = sum2 + etas( :,ipart).^2;
26 sum3 = sum3 + etas( :,ipart).^3;
27 end
28
29 dfdcon = B*(2.0*con + 4.*sum3 -
6.0*sum2) - 2.0*A*con.^2 .* ...
(1.0-con) + 2.0*A*con.* ...
(1.0-con).^2;
31
32 end %if
33
34 if(iflag == 2)
35 %--
36 %% derivative of free energy
37 for etas:
38 %--
39 sum2=zeros(NxNy,1);
40
41 for ipart = 1: npart
42
43 sum2 = sum2 + etas( :,ipart).^2;
44 end
45
46 dfdeta = B*(-12.0*eta.^2 .*
(2.0-con) +12.0 *eta .* (1.0-con) +...
47 12.0*eta.*sum2);
48 end %if
49
50 end %endfunction

```

Line numbers:

5–6:	Set constants in free energy function, Eq. 4.39.
8:	Total number of grid points in the simulation cell.
13–32:	Functional derivative of free energy for the concentration field at all grid points.
17–27:	Summations in Eq. 4.39 for all grid points.
29–30:	Functional derivative of free energy respect to concentration at all grid points.

34–48:	Functional derivative of free energy for the current order parameter at all grid points.
40–44:	Summation in Eq. 4.39.
46–47:	Functional derivative of the free energy respect to current order parameter, in the main program, for the all grid points in the simulation cell.

References

- German RM (1996) Sintering theory and practice. Wiley, New York
- Svoboda J, Riedel H (1995) Quasi equilibrium for coupled grain-boundary and surface diffusion. Acta Mater 43:499.
- Svoboda J, Riedel H (1995) New solutions describing the formation of necks in solid-state sintering. Acta Mater 43:1
- Zhang W, Schneibel J (1995) The sintering of two particles by surface and grain boundary diffusion: a two dimensional numerical study. Acta Mater 43:4377
- Pan J, Cocks ACF (1995) A numerical technique for the analysis of coupled surface and grain boundary diffusion. Acta Mater 43:1395
- Pan J, Le H, Kucherenko S, Yeomans J (1998) A model for the sintering of spherical particles of different sizes by solid-state diffusion. Acta Mater 46:4671
- Maximenko A, Olevsky E (2004) Effective diffusion coefficients in solid-state sintering. Acta Mater 52:2953
- Parhami F, McMeeking RM, Cocks ACF, Suo Z (1999) A model for the sintering and coarsening of rows of spherical particles. Mech Mater 31:43
- Zeng P, Tikare V (1998) Potts model simulation of grain size distribution during final stage sintering. MRS Proc 529:77
- Yamashita T, Uehara T, Watanabe R (2005) Multi-layered Potts model simulation morphological changes of the neck during sintering in Cu–Ni system. Mater Trans 46:88
- Dudek MR, Gouyet JF, Kolb M (1998) Q+1 state Potts model of late state sintering. Surf Sci 401:220
- Braginsky M, Tikare V, Olevsky E (2005) Numerical simulation of solid-state sintering. Int J Solid Struct 42:621
- Wang Y (2006) Computer modeling and simulation of solid-state sintering: a phase-field approach. Acta Mater 54:953
- Deng J (2012) A phase-field model of sintering with direction dependent diffusion. Mater Trans 53:1:385

15. Kumar V, Fang ZZ, Fife PC (2010) Phase-field simulations of grain growth during sintering of two unequal-sized particles. *Mater Sci Eng A* 528:254
16. L Liu, Gao F, Li B, Hu G (2011) Phase-field simulation of process of sintering ceramics *Adv Mater Res* 154–155:1674
17. Zhang R, Chen Z, Fang W, Qu X (2014) Thermodynamically consistent phase-field model for sintering process with multiphase powders. *Trans Nonferrous Met Soc China* 24:783

4.7 Case Study-IV

Phase-field modeling of dendritic solidification

Objectives:

The objective of this study is to demonstrate the implementation of the anisotropy in the interfacial energy and the mobility that appears in phase-field models.

4.7.1 Background

Formation of complex microstructures during the solidification from a liquid phase, such as formation of snow flakes and cast microstructure of metals, has fascinated the scientist over the centuries. In particular, evolution of microstructural scale of dendrites during the solidification determines many physical and mechanical properties of metals, since almost every metallic system originates from the liquid state.

The experimental studies performed on succinonitrile by Glicksman and coworkers [1, 2] provided much-needed data to guide the development of solidification theories. The early foundation of most solidification theories is based on the time-dependent Stefan problem, sharp-interface approach, as described in chapter I. This theory describes the evolution of the diffusion field around the solidification front with two conditions at the solid–liquid interface. The first condition relates to the velocity of the moving solid–liquid interface to the difference in thermal fluxes across the interface. The second,

called Gibbs–Thomson condition, relates the interfacial temperature to the thermodynamic equilibrium, the local interfacial curvature, and interface kinetics.

Owing to difficulties in solving set of differential equations resulting from sharp-interface approaches, phase-field modeling approach has been very successful in simulation of dendritic solidification process. Caginalp and Fife [3] adopted the phase-field modeling approach to simulate the dendritic solidification in 1980s. Kobayashi [4] developed a simple phase-field model for the solidification of the one-component system. His model included the interfacial anisotropy. The model was successful to elucidate the qualitative relations between the shapes of crystals and some physical parameters and noises to give a crucial influence on the side branch structure of dendrites. The first phase-field model for alloys was developed by Wheeler et al. [5, 6], called the WBM model. Kim et al. [7, 8] presented another model for alloys by adopting the thin-interface limit, which is known as the KKS model. Karma [9] presented a phase-field formulation to quantitatively simulate microstructural pattern formation in alloys, and the thin-interface limit of this formulation yielded a much less stringent restriction on the choice of interface thickness than previous formulations and permitted to eliminate nonequilibrium effects at the interface. Dendrite growth simulations with vanishing solid diffusivity showed that both the interface evolution and the solute profile in the solid were accurately modeled by this approach. Solidifications with forced flow or convection were also studied in binary alloys in 2D and 3D [10–14]. The solidification of multicomponent and multiphase alloys was also investigated using phase-field methods [15–18]. In order to reduce the computational burden and to provide a better spatial resolution, the adaptive meshing algorithms for finite difference [19, 20], finite element method [21–25], and finite volume method [26–28] have also been developed for these purposes.

4.7.2 Phase-Field Model

The model given below is based on the most celebrated work of Kobayashi [4] which is one of the earliest phase-field models for the dendritic solidification. It was chosen here for its simplicity; it also forms the foundation to most of the refined and advanced models cited above.

The model includes two variables: one is the non-conserved phase-field parameter, $\varphi(r, t)$, which takes the value of one in the solid and zero in the liquid. The other is the temperature field $T(r, t)$ which also evolves as the solidification progresses. Of course, r is the spatial position and t is the time. In the following, Ginzburg–Landau type free energy is considered:

$$F(\varphi, m) = \int_V \frac{1}{2} \varepsilon^2 |\nabla \varphi|^2 + f(\varphi, m) dv \quad (4.44)$$

where the first term in the integrant is the gradient energy, ε is the anisotropic gradient energy coefficient which determines the thickness of the interface layer. The second term is the local free energy which is a double-well potential and has local minimums at $\varphi = 0$ and $\varphi = 1$. The m is the driving force for the interface motion proportional to supercooling. The functional form for free energy, $f(\varphi, m)$, is taken as:

$$f(\varphi, m) = \frac{1}{4} \varphi^4 - \left(\frac{1}{2} - \frac{1}{3} m \right) \varphi^3 + \left(\frac{1}{4} - \frac{1}{2} m \right) \varphi^2 \quad (4.45)$$

The interfacial anisotropy is introduced by assuming that ε depends on the direction of the

$$\tau \frac{\partial \varphi}{\partial t} = \frac{\partial}{\partial y} \left(\varepsilon \frac{\partial \varepsilon}{\partial \theta} \frac{\partial \varphi}{\partial x} \right) - \frac{\partial}{\partial x} \left(\varepsilon \frac{\partial \varepsilon}{\partial \theta} \frac{\partial \varphi}{\partial y} \right) + \nabla \cdot (\varepsilon^2 \nabla \varphi) + \varphi(1 - \varphi) \left(\varphi - \frac{1}{2} + m \right) \quad (4.51)$$

The evolution of temperature field, $T(r, t)$, is derived from the conservation law of enthalpy as:

$$\frac{\partial T}{\partial t} = \nabla^2 T + \kappa \frac{\partial \varphi}{\partial t} \quad (4.52)$$

T is nondimensionalized so that the characteristic cooling temperature is zero and the equilibrium

outer normal vector at the interface. Its value evaluated as,

$$\varepsilon = \bar{\varepsilon} \sigma(\theta) \quad (4.46)$$

where $\bar{\varepsilon}$ is the mean value of ε and $\sigma(\theta)$ represents the anisotropy, which is expressed as:

$$\sigma(\theta) = 1 + \delta \cos(j(\theta - \theta_0)) \quad (4.47)$$

in which δ is the strength of the anisotropy, j is the mode number of anisotropy, which takes the value of four cubic lattices and six for hexagonal lattices. θ_0 is the initial offset angle and taken as a constant. The angle, θ , is defined as:

$$\theta = \tan^{-1} \left(\frac{\partial \varphi / \partial y}{\partial \varphi / \partial x} \right) \quad (4.48)$$

The parameter m that appears in Eq. 4.45 is assumed to be dependent upon the degree of supercooling and the temperature. This dependency is expressed as:

$$m(T) = \left(\frac{\alpha}{\pi} \right) \tan^{-1} [\gamma (T_{eq} - T)] \quad (4.49)$$

where α is a positive constant and T_{eq} is the equilibrium temperature.

The time-dependent Ginzburg–Landau or Allen–Cahn equation is assumed for the evolution,

$$\tau \frac{\partial \varphi}{\partial t} = - \frac{\delta F}{\delta \varphi} \quad (4.50)$$

By taking the functional derivative of Eq. 4.44, the time evolution becomes,

temperature is one. κ is a dimensionless latent heat which is proportional to the latent heat and inversely proportional to the strength of cooling. For simplicity, the diffusion constants are set to be identical in both solid and liquid phases.

4.7.3 Numerical Implementation

The Laplace operator and the directional derivatives in Eqs. 4.51 and 4.52 are approximated with finite difference algorithm by utilizing five-point stencil in two-dimensional space. The time integration is carried by simple explicit Euler time marching scheme. The nondimensionalized values of the parameters that appear in the equations are given in Table 4.2.

In order to promote the solidification a stable nucleus having radius of 5 grid spacing was placed to the center of the simulation cell. In the simulations, no additional noise term was introduced to promote side branching of dendrites.

4.7.4 Results and Discussion

The simulations were carried out for a square simulation cell having $N_x = 300$, $N_y = 300$ and the grid spacing $dx = dy = 0.03$. The nondimensionalized parameters given above were used in the solutions. The time evolution of dendritic structures and temperature field for strength of anisotropy $j = 4$ (for cubic lattices) are shown in Fig. 4.16a, b; and for strength of anisotropy $j = 6$ (for hexagonal lattices) are shown in Fig. 4.17a, b.

As can be seen from the figures, depending on the strength of the anisotropy parameter, either four or six primary arms develop from the initial seed. As the solution progresses, the side arms

Table 4.2 The nondimensional parameters used in the simulations

τ	$\bar{\varepsilon}$	κ	δ	j	α	T_{eq}	θ_0	dx, dy
0.0003	0.01	1.8	0.02	4.0, 6.0	0.9	1.0	0.2	0.03

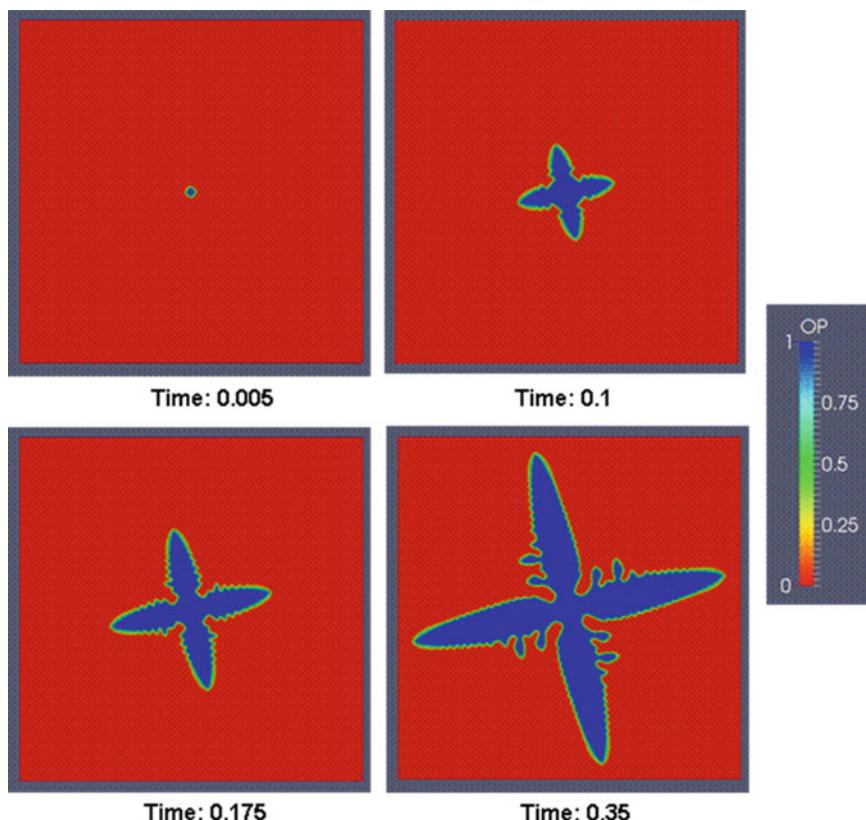


Fig. 4.16 (a) The time evolution of dendritic solidification from the initial seed with strength of anisotropy, $j = 4$. (b) Time evolution of the temperature field

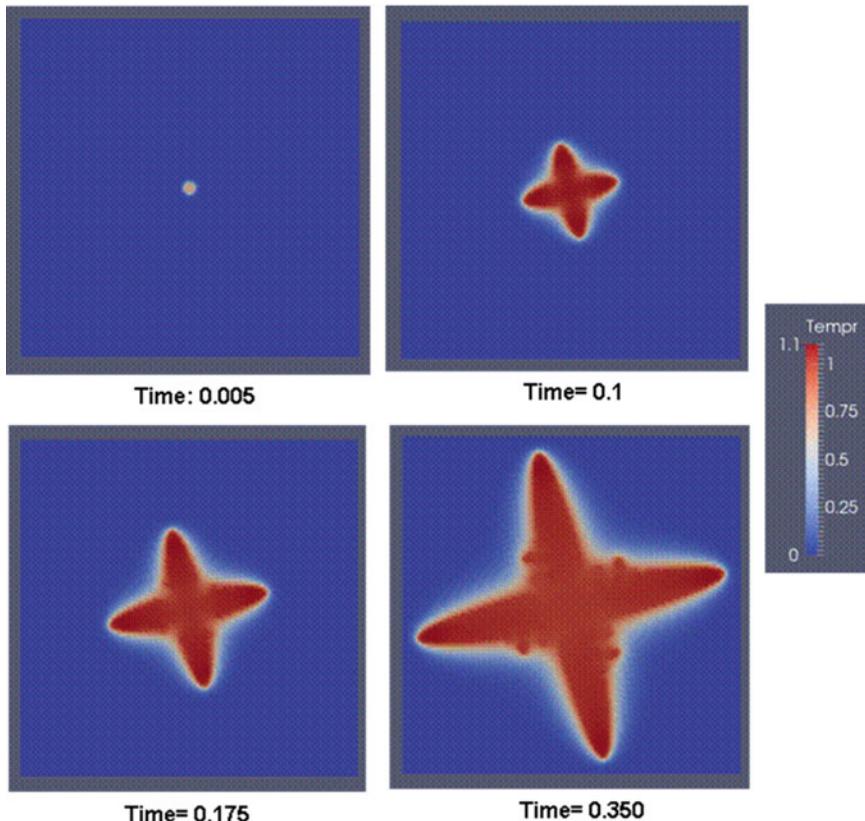


Fig. 4.16 (continued)

develop. The density of the side arms is slightly larger for the hexagonal dendrite morphology. Even though there was no noise term in the solution and every coarse grid was employed in these simulations, the results obtained qualitatively agrees very well with the references given earlier.

The movie files resulting from these simulations can be found in subdirectory *case_study_4* in downloadable file.

4.7.5 Source Codes

Two programs are provided in this section. The first one, *fd_den_v1.m*, is in longhand format and *fd_den_v2.m* is the optimized version for Matlab/Octave. The functions and programs are given below.

Program

fd_den_v1.m

This program solves the phase-field modeling of the dendritic solidification with finite difference algorithm. The code is in longhand format and not optimized for Matlab/Octave.

The program makes calls to the following functions:

- **nucleus.m**
- **write_vtk_grid_values.m**

Listing:

```

1      %%%%%%%%
2      %
3      % PHASE-FIELD FINITE-DIFFERENCE %

```

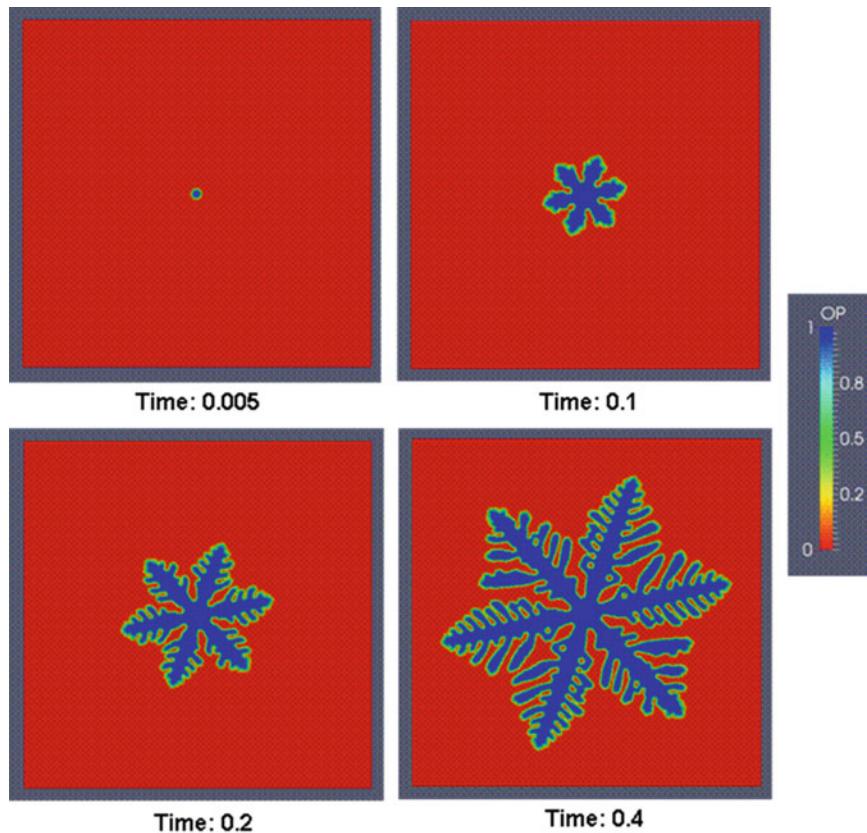


Fig. 4.17 (a) The time evolution of dendritic solidification from the initial seed with strength of anisotropy, $j = 6$.
(b) Time evolution of the temperature field

```

4 % CODE FOR DENDRITIC SOLIDIFICATION %
5 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6 %--- Material specific parameters:
7 %%get initial wall time:
8 time0=clock();
9 format long;
10
11 %% Simulation cell parameters:
12
13 Nx = 300;
14 Ny = 300;
15 NxNy= Nx*Ny;
16
17 dx = 0.03;
18 dy = 0.03;
19
20 %% Time integration parameters:
21
22 nstep = 4000;
23 nprint = 50;
24 dtime = 1.0e-4;
25
26 %%--- Initialize and introduce
27 %% initial nuclei:
28 tau = 0.0003;
29 epsilonb = 0.01;
30 mu = 1.0;
31 kappa = 1.8;
32 delta = 0.02;
33 aniso = 4.0;
34 alpha = 0.9;
35 gamma = 10.0;
36 teq = 1.0;
37 theta0 = 0.2;
38 seed = 5.0;
39 %
40 pix=4.0*atan(1.0);
41
42 %%--- Initialize and introduce
43 %% initial nuclei:

```

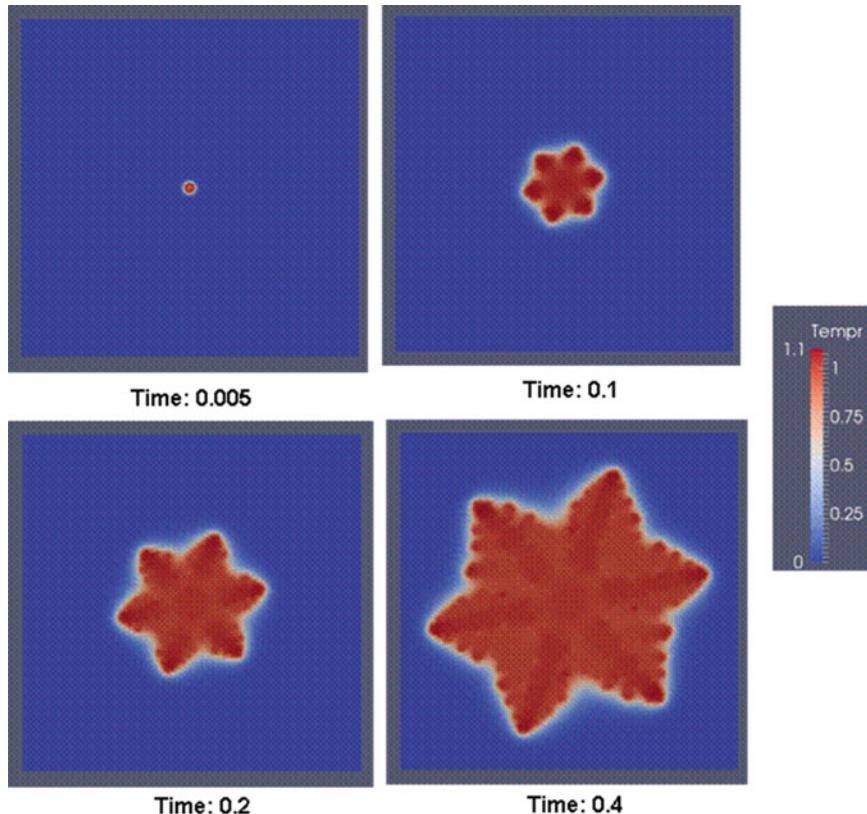


Fig. 4.17 (continued)

```

43
44 [phi,tempr] = nucleus(Nx,Ny,seed);
45
46 %---
47 %- Evolution
48 %---
49
50 for istep=1:nstep
51
52 %---
53 % calculate the laplacians and epsilon:
54 %---
55
56 for i=1:Nx
57 for j=1:Ny
58
59 jp=j+1;
60 jm=j-1;
61 ip=i+1;
62 im=i-1;
63
64
65 jp=j+1;
66 jm=j-1;
67
68 ip=i+1;
69 im=i-1;
70
71 if(im== 0)
72 im=Nx;
73 end
74 if(ip == (Nx+1))
75 ip=1;
76 end
77
78 if(jm== 0)
79 jm = Ny;
80 end
81
82 if(jp == (Ny+1))
83 jp=1;
84 end
85
86 hne=phi(ip,j);

```

```

87 hnw=phi(im,j);
88 hns=phi(i,jm);
89 hnn=phi(i,jp);
90 hnc=phi(i,j);
91
92 lap_phi(i,j) = (hnw + hne + hns +
93   hnn - 4.0*hnc) / (dx*dy);
93
94 hne=tempr(ip,j);
95 hnw=tempr(im,j);
96 hns=tempr(i,jm);
97 hnn=tempr(i,jp);
98 hnc=tempr(i,j);
99
100 lap_tempr(i,j) = (hnw + hne + hns + hnn
101   - 4.0*hnc) / (dx*dy);
101
102 %--gradients:
103
104 phidx(i,j) = (phi(ip,j) - phi(im,j)) /
105   (2.0*dx);
105 phidy(i,j) = (phi(i,jp) - phi(i,jm)) /
106   (2.0*dy);
106
107 %-- calculate angle:
108
109 theta = atan2(phidy(i,j),
110   phidx(i,j));
110
111 %--- epsilon and its derivative:
112
113 epsilon(i,j) = epsilonb*(1.0+delta *
114   *cos(aniso*(theta-theta0)));
114
115 epsilon_deriv(i,j) = -epsilonb*aniso*
116   delta*sin(aniso*(theta-theta0));
116
117 end%for j
118 end%for i
119
120 %---
121
122 for i=1:Nx
123 for j=1:Ny
124
125 jp=j+1;
126 jm=j-1;
127
128 ip=i+1;
129 im=i-1;
130
131 jp=j+1;
132 jm=j-1;
133
134 ip=i+1;
135 im=i-1;
136
137 if(im==0)
138   im=Nx;
139 end
140 if(ip==(Nx+1))
141   ip=1;
142 end
143
144 if(jm==0)
145   jm=Ny;
146 end
147
148 if(jp==(Ny+1))
149   jp=1;
150 end
151
152 phioold=phi(i,j);
153
154 %-- first term:
155
156 term1= (epsilon(i,jp)*epsilon_deriv
157   (i,jp)*phidx(i,jp) - ...
158   epsilon(i,jm)*epsilon_deriv(i,jm)*
159   phidx(i,jm))/(2.0*dy);
159
160 %-- second term:
161 term2= -(epsilon(ip,j)*epsilon_deriv
162   (ip,j)*phidy(ip,j) - ...
163   epsilon(im,j)*epsilon_deriv(im,j)*
164   phidy(im,j))/(2.0*dx);
164
165 %-- factor m:
166 m=alpha/pi * atan(gamma*
167   (teq-temp(i,j)));
167
168 %-- Time integration:
169
170 phi(i,j)=phi(i,j) +(dtime/tau)*
171   (term1+term2+epsilon(i,j)^2*lap_phi
172   (i,j) +...
173   phioold*(1.0-phioold)*
174   (phioold-0.5+m));

```

```

172
173 %-- evolve temperature:
174
175 tempr(i,j) =tempr(i,j) +dtim*lap_
176     tempr(i,j) +kappa*(phi(i,j)-phold);
177 end
178 end
179
180 %---- print results
181
182 if(mod(istep,nprint) == 0 )
183
184 fprintf('done step: %5d\n',istep);
185
186 %fname1 =sprintf('time_%d.out',
187     istep);
188 %out1 = fopen(fname1,'w');
189 %for i=1:Nx
190 %for j=1:Ny
191 %ii=(i-1)*Nx+j;
192 %fprintf(out1,'%5d %5d %14.6e %14.6e
193 %\n',i,j,phi(i,j),tempr(i,j))
194 %end
195 %fclose(out1)
196
197 %--- write vtk file:
198
199 write_vtk_grid_values(Nx,Ny,dx,dy,
200     istep,phi,tempr);
201
202 end%if
203
204 end%istep
205
206 %--- calculate compute time:
207
208 compute_time = etime(clock(),time0);
209 fprintf('Compute Time: %10d\n',
210     compute_time);
211

```

Line numbers:

7–8:	Get wall clock time beginning of the execution.
11–19:	Simulation cell parameters.
13:	Number of grid points in the x -direction.
14:	Number of grid points in the y -direction.
17:	Grid spacing between two grid points in the x -direction.
18:	Grid spacing between two grid points in the y -direction.
20–25:	Time integration parameters.
22:	Number of time integration steps.
23:	Output frequency to write the results to file.
24:	Time increment for numerical integration.
26–37:	Model-specific parameters (see Eqs. 4.51 and 4.52, also Table 4.2).
38:	The size of the initial seed. (In grid numbers, see function nucleus).
40:	The value of pi.
42–45:	Initialize the phi and tempr arrays and introduce the seed in the center of the simulation cell.
50–204:	Time integration of Eqs. 4.51 and 4.52.
53–118:	Calculate the $\nabla^2\varphi$, ∇^2T and the ε and its derivatives $\partial\varepsilon/\partial\theta$ in Eqs. 4.51 and 4.52.
92:	Calculate the $\nabla^2\varphi$ in Eq. 4.51.
100:	Calculate the ∇^2T in Eq. 4.52.
102–106:	Calculate the Cartesian derivatives of phi, $\nabla\varphi$.
109:	Calculate angle, $\theta = \tan^{-1}((\partial\varphi/\partial y)/(\partial\varphi/\partial x))$, Eq. 4.48.
113:	Calculate ε , Eqs. 4.46 and 4.47.
115:	Calculate derivative of ε , $\partial\varepsilon/\partial\theta$.
152:	Assign the values of phi from previous time step to phold.
156–157:	Calculate the first term of Eq. 4.51.
161–162:	Calculate the second term of Eq. 4.51.
166:	Calculate value of m in Eq. 4.49.
170–171:	Evolve φ with Euler time integration.
175:	Evolve temperature with Euler time integration, Eq. 4.52.
180–202:	If print frequency is reached, output the results.
186–195:	Opens an output file and writes the grid points, φ and temperature field to file. These lines are commented out, but they can be changed including the format of the output file.

(continued)

199:	Writes the results for contour plot in vtk format to be viewed by Paraview.
208–209:	Calculates the total execution time and prints on the screen.

Program

fd_den_v2.m

This program solves the phase-field modeling of the dendritic solidification with finite difference algorithm and it is optimized for Matlab/Octave.

The program makes calls to the following functions:

- **nucleus.m**
- **gradient_mat.m**
- **vec2matx.m**
- **laplacian.m**
- **write_vtk_grid_values.m**

Listing:

```

1    %%%%%%%%%%%%%%%%
2    % PHASE-FIELD FINITE-DIFFERENCE %
3    %           CODE FOR             %
4    % DENDRITIC SOLIDIFICATION      %
5    % (OPTIMIZED FOR MATLAB/OCTAVE) %
6    %%%%%%%%%%%%%%%%
7    %%= get initial wall time:
8    time0=clock();
9    format long;
10
11   %-- Simulation cell parameters:
12
13   Nx = 300;
14   Ny = 300;
15   NxNy= Nx*Ny;
16
17   dx = 0.03;
18   dy = 0.03;
19
20   %-- Time integration parameters:
21
22   nstep = 4000;
23   nprint = 50;
24   dtimre = 1.e-4;
25
26   %--- Material specific parameters:
27
28   tau = 0.0003;
29   epsilonb = 0.01;
30   mu = 1.0;
31   kappa = 1.8;
32   delta = 0.02;
33   aniso = 6.0;
34   alpha = 0.9;
35   gamma = 10.0;
36   teq = 1.0;
37   theta0 = 0.2;
38   seed = 5.0;
39   %
40   pix=4.0*atan(1.0);
41
42   %--- Initialize and introduce
43   %     initial nuclei:
44   [phi,temp] = nucleus(Nx,Ny,seed);
45
46   %--- Laplacian templet:
47
48   [laplacian] = laplacian(Nx,Ny,
49   dx,dy);
50   %---
51   %--- Evolution
52   %---
53
54   for istep=1:nstep
55
56   phiold=phi;
57
58   %---
59   % calculate the laplacians
60   % and epsilon:
61
62   phi2 = reshape(phi',NxNy,1);
63
64   lap_phi2 = laplacian*phi2;
65
66   [lap_phi]=vec2matx(lap_phi2,Nx);
67
68   %--
69
70   tempx = reshape(temp',NxNy,1);
71
72   lap_tempx=laplacian*tempx;
```

```

73
74 [lap_temp] =vec2matx
    (lap_tempx,Nx);
75
76 %--gradients of phi:
77
78 [phidy,phidx]=gradient_mat
    (phi,Nx,Ny,dx,dy);
79
80 %-- calculate angle:
81
82 theta=atan2(phidy,phidx);
83
84 %--- epsilon and its derivative:
85
86 epsilon=epsilononb*(1.0+delta*
    cos(aniso*(theta-theta0)));
87
88 epsilon_deriv=-epsilononb*aniso*
    delta*sin(aniso.*(theta-theta0));
89
90 %--- first term:
91
92 dummyx=epsilon.*epsilon_deriv.
    *phidx;
93
94 [term1,dummy]=gradient_mat
    (dummyx,Nx,Ny,dx,dy);
95
96 %--- second term:
97
98 dummyy=-epsilon.*epsilon_deriv.
    *phidy;
99
100 [dummy,term2]=gradient_mat
    (dummyy,Nx,Ny,dx,dy);
101
102 %--- factor m:
103
104 m=(alpha/pix)*atan(gamma*
    (teq-temp));
105
106 %-- Time integration:
107
108 phi=phi+(dtime/tau)*(term1+
    term2+epsilon.^2.*lap_phi+...
109 phibold.*(1.0-phibold).*...
    (phibold-0.5+m));
110
111
112 %-- evolve temperature:
113
114 temp=tempr+dtime*lap_temp +
    kappa*(phi-phibold);
115
116
117 %---- print results
118
119 if(mod(istep,nprint)==0)
120
121 fprintf('done step: %5d\n',istep);
122
123 %fname1=sprintf('time_%d.out',
    istep);
124 %out1=fopen(fname1,'w');
125
126 %for i=1:Nx
127 %for j=1:Ny
128 %ii=(i-1)*Nx+j;
129 %fprintf(out1,'%5d %5d %14.6e %14.6e
    \n',i,j,phi(i,j),temp(i,j))
130 %end
131 %end
132
133 %fclose(out1)
134
135 %--- write vtk file:
136
137 write_vtk_grid_values(Nx,Ny,dx,dy,
    istep,phi,temp);
138
139 end %if
140
141 end %istep
142
143 %--- calculate compute time:
144
145 compute_time=etime(clock(),
    time0);
146 fprintf('Compute Time: %10d\n',
    compute_time);
147

```

Line numbers:

7-9:	Get wall clock time beginning of the execution.
11-19:	Simulation cell parameters.
13:	Number of grid points in the <i>x</i> -direction.
14:	Number of grid points in the <i>y</i> -direction.

(continued)

17:	Grid spacing between two grid points in the x -direction.
18:	Grid spacing between two grid points in the y -direction.
20–25:	Time integration parameters.
22:	Number of time integration steps.
23:	Output frequency to write the results to file.
24:	Time increment for numerical integration.
26–37:	Model-specific parameters (see Eqs. 4.51 and 4.52, also Table 4.2).
38:	The size of the initial seed. (In grid numbers, see function nucleus).
40:	The value of pi.
42–45:	Initialize the phi and temp arrays and introduce the seed in the center of the simulation cell.
46–49:	Calculate the finite difference template for the laplacians.
54–141:	Time integration of Eqs. 4.51 and 4.52.
56:	Assign the values of phi from the previous time step to phold.
59–89:	Calculate the $\nabla^2\varphi$, ∇^2T and the ϵ and its derivatives $\partial\epsilon/\partial\theta$ in Eqs. 4.48, 4.51 and 4.52.
62:	Convert phi(Nx,Ny) matrix into one-dimensional array phi2(Nx*Ny).
64:	Calculate $\nabla^2\varphi$ for all grid points.
66:	Convert lap_phi2 (Nx*Ny) from one-dimensional array to lap_phi(Nx,Ny) square matrix.
70:	Convert temp(Nx,Ny) matrix into one-dimensional array tempx(Nx*Ny).
72:	Calculate ∇^2T in Eq. 4.52 for all grid points.
74:	Convert lap_tempx(Nx*Ny) from one-dimensional array to lap_temp(Nx,Ny) square matrix.
76–79:	Calculate Cartesian derivatives of phi, $\nabla\varphi$.
82:	Calculate angle, $\theta = \tan^{-1}((\partial\varphi/\partial y)/(\partial\varphi/\partial x))$, Eq. 4.48, for all grid points.
84–89:	Calculate ϵ , Eq. 4.46 and its derivative $\partial\epsilon/\partial\theta$, Eq. 4.51, for all grid points.
90–95:	Calculate the first term of Eq. 4.51 for all grid points.
96–101:	Calculate the second term of Eq. 4.51 for all grid points.
102–105:	Calculate the value of m, Eq. 4.49 for all grid points.
108–109:	Evolve φ with Euler time integration.
114:	Evolve temperature with Euler time integration, Eq. 4.52.
117–139:	If print frequency is reached, output the results.

123–133:	Opens an output file and writes the grid points, φ and temperature field to file. These lines are commented, but can be changed including the format of the output file.
137:	Writes the results for contour plot in vtk format to be viewed by Paraview.
145–146:	Calculates the total execution time and prints on the screen.

Function

nucleus.m

This function introduces initial solid nuclei in the center of the simulation cell. The size of the nuclei is in terms of grid numbers.

Variable and array list:

Nx:	Number of grid points in x -direction.
Ny:	Number of grid points in y -direction.
seed:	Initial nuclei size (in terms of grid numbers).
phi(Nx,Ny):	Phase-field parameter.
temp(Nx,Ny):	Temperature

Listing:

```

1  function [phi,temp] = nucleus
2    (Nx,Ny,seed)
3    format long;
4
5    %for i=1:Nx
6    %for j=1:Ny
7
8    %phi(i,j) = 0.0;
9    %temp(i,j) = 0.0;
10
11  %end
12  %end
13
14  phi = zeros(Nx,Ny);
15  temp = zeros(Nx,Ny);
16
17  for i=1:Nx
18  for j=1:Ny
19  if ((i-Nx/2)*(i-Nx/2)+(j-Ny/2)*
(j-Ny/2) < seed)
20  phi(i,j) = 1.0;
```

```

21 end
22 end
23 end
24
25 end %endfunction

```

Line numbers:

5–12:	Initialize phi and tempr arrays with longhand format (note these lines are commented out).
14–15:	Initialize phi and tempr arrays
17–23:	Compare the vector length starting from the center of the simulation cell. If it is smaller or equal to the value of seed, assign value of one to phi.

Function

gradient_mat.m

This function calculates the one-dimensional gradients for x and y directions of a given matrix. Function utilizes the built-in function *gradient* in Matlab/Octave, returns the values that are adjusted for grid spacing and the periodic boundaries:

Variable and array list:

Nx:	Number of grid points in the x -direction.
Ny:	Number of grid points in the y -direction.
dx:	Grid spacing between two grid points in the x -direction.
dy:	Grid spacing between two grid points in the y -direction.
matx(Nx, Ny):	Input matrix.
matdx(Nx, Ny):	Derivatives of input matrix in the x -direction.
matdy(Nx, Ny):	Derivatives of input matrix in the y -direction.

Listing:

```

1 function [matdx,matdy] =
2     gradient_mat (matx,Nx,Ny,dx,dy)
3 format long;
4
5 %--
6 %-- this function uses build-in
7 %-- function gradient
8 %-- in matlab/octave.

```

```

8 %--
9
10 [matdx,matdy] = gradient (matx) ;
11
12 %--- for periodic boundaries:
13
14 matdx(1:Nx,1) = 0.5* (matx(1:Nx,2)
15 - matx(1:Nx,Nx)) ;
15 matdx(1:Nx,Nx)= 0.5* (matx(1:Nx,1)
16 - matx(1:Nx,Nx-1));
16
17 matdy(1,1:Ny) = 0.5* (matx(2,1:Ny)
18 - matx(Ny,1:Ny)) ;
18 matdy(Ny,1:Ny)= 0.5* (matx(1,1:Ny)
19 - matx(Ny-1,1:Ny)) ;
19
20 matdx = matdx/dx;
21 matdy = matdy/dy;
22
23 end %endfunction

```

Line numbers:

10:	Evaluate one-dimensional gradients by utilizing built-in function <i>gradient</i> in Matlab/Octave.
14–22:	Rearrange the values for periodic boundaries and grid spacing.

Function

vec2matx.m

This function turns one-dimensional array to a square $[N,N]$ matrix. It is based on the built-in function *vect2mat* in Matlab/Octave and also utilizes two other built-in functions *ceil* and *reshape*.

Variable and array list:

N:	Size of the square matrix.
V(N*N):	One-dimensional input array.
M(N,N):	Output square $[N,N]$ matrix.

Listing:

```

1 function [M] = vec2matx (V, N)
2
3 format long;
4

```

```

5 V = V.' ;
6 V = V(:) ;
7
8 R = ceil (length (V) / N) ;
9
10 M = reshape (V, N, R).' ;
11
12 end %endfunction

```

References

1. Huang SC, Glicksman ME (1981) Fundamentals of dendritic solidification: steady-state tip growth. *Acta Metall* 29: 701
2. Glicksman ME (1984) Free dendritic growth. *Mater Sci Eng* 65:45
3. Caginalp G, Fife P (1986) Phase-Field methods for interface boundaries. *Phys Rev B* 33:7792
4. Kobayashi R (1993) Modeling and numerical simulations of dendritic crystal-growth. *Physica D* 63:410
5. Wheeler AA, Boettger WJ, Mcfadden GB (1992) Phase-field model for isothermal phase-transitions in binary-alloys. *Phys Rev A* 45:7424
6. Wheeler AA, Boettger WJ, Mcfadden GB (1993) Phase-field model of solute trapping during solidification. *Phys Rev E* 47:1893
7. Kim SG, Kim WT, Suzuki T (1999) Phase-field model for binary alloys. *Phys Rev E* 60:7186
8. Kim SG, Kim WT, Suzuki T (1998) Interfacial compositions of solid and liquid in a phase-field model with finite interface thickness for isothermal solidification in binary alloys. *Phys Rev E* 58:3316
9. Karma A (2001) Phase-field formulation for quantitative modeling of alloy solidification. *Phys Rev Lett* 87:045501
10. Karma A, Rappel WJ (1998) Quantitative phase-field modeling of dendritic growth in two and three dimensions. *Phys Rev E* 57:4323
11. Tong X, Beckermann C, Karma A, Li Q (2001) Phase-field simulations of dendritic crystal growth in a forced flow. *Phys Rev E* 63:061601
12. Jeong JH, Goldenfeld N, Dantzig JA (2001) Phase field model for three-dimensional dendritic growth with fluid flow. *Phys Rev E* 64:041602
13. Tsai YL, Chen CC, Lan CW (2010) Three-dimensional adaptive phase field modeling of directional solidification of a binary alloy: 2D–3D transitions. *Int J Heat Mass Trans* 53:2272
14. Du LF, Zhang R, Zhang LM (2013) Phase-field simulation of dendritic growth in a forced liquid metal flow coupling with boundary heat flux. *Sci China Technol Sci* 56:2586
15. Nestler B, Choudhury A (2011) Phase-field modeling of multi-component systems. *Curr Opin Solid State Mater Sci* 15:93
16. Nestler B, Garcke H, Stinner B (2005) Multicomponent alloy solidification: phase-field modeling and simulations. *Phys Rev E* 71:041609
17. Zhang RJ, Li M, Allison J (2010) Phase-field study for the influence of solute interactions on solidification process in multicomponent alloys. *Comp Mater Sci* 47:832
18. Singer-Loginova I, Singer HM (2008) The phase field technique for modeling multiphase materials. *Rep Prog Phys* 71:106501
19. Braun RJ, Murray BT, Soto J Jr (1997) Adoptive finite-difference computations of dendritic growth using a phase-field model. *Model Simul Mater Sci Eng* 5:365
20. Braun RJ, Murray BT (1997) Adoptive phase-field computations of dendritic crystal growth. *J Crystal Growth* 174:41
21. Tonhardt R, Amberg J (1998) Phase-field simulation of dendritic growth in a shear flow. *J Crystal Growth* 194:406
22. Tonhardt R, Amberg G (2000) Dendritic growth of randomly oriented nuclei in a shear flow. *J Crystal Growth* 213:161
23. Palle N, Dantzig JA (1996) An adoptive mesh refinement scheme for solidification problems. *Metall Mater Trans* 27A:707
24. Provatas N, Goldenfeld N, Dantzig JA (1999) Adoptive mesh refinement computation of solidification microstructures using dynamics data structures. *J Comp Phys* 148:265
25. Provatas N, Goldefeld NJ, Dantzig JA (1998) *Phys Rev Lett* 80:3308
26. Lan CW, Liu C, Hsu M (2002) An adoptive finite volume method for incompressible heat flow problems in solidification. *J Comp Phys* 178:464
27. Lan CW, Hsu CM, Liu CC. (2002) Efficient adoptive phase-field simulation of dendritic growth in a forced flow at low supercooling. *J Crystal Growth* 241:379
28. Lan CW, Chang YC, Shih CJ (2003) Adoptive phase-field simulation of non-isothermal free dendritic growth of a binary alloy. *Acta Mater* 51:1857

4.8 Case Study-V

Multicellular systems, the role of elastic mismatch and cell motility

Objectives:

The objective of this case study is to demonstrate how to modify the free energy functional so that each cell, described with a non-conserved order parameter in the phase-field model, undergoes to large deformations and shape changes while preserving its original volume. Formulism still allows minimization of the total energy;

however, when cells are brought together they deform collectively rather than shrink/grow and absorb each other.

4.8.1 Background

The understanding of cell migration is crucial to many biological processes, including chemotaxis, embryogenesis, and cancer metastasis. One example is the cancer metastasis, where a cancer cell squeezes through the endothelium to reach the blood stream and eventually forms a secondary tumor elsewhere in the body [1]. Even though, the experiments have generated vast amount of quantitative data, the precise mechanisms of collective behavior of cells leading to cell motion have not very well understood. A number of recent theoretical studies have attempted to model the cell migration in an effort to elucidate the motility mechanisms [2]. Of course, such efforts offer an alternative perspective of the diseases and may help to optimize their treatments and detections.

In order to investigate the structural evolution of cellular systems several cell models have been developed, including the vertex dynamics [3, 4], center dynamics models [5, 6], and the cellular Potts models [7, 8]. Recently, the phase-field modeling approaches finding applications in

modeling and simulation efforts in biology and medicine [9–21].

4.8.2 Phase-Field Model

The phase-field model chosen for this case study is based on the most recent study of Palmieri et al. [1]. The references cited above are similar to the model described in their work; however, there are some differences in degree of details of the phase-field formulation, introduction of the cell dynamics, and the solution methodologies of the resulting phase-field equations. As described in [1], the model is successful: (1) To tracking the cell boundaries every efficiently, by taking advantage of the phase-field formulism. (2) Ability to describe extremely large deformations. (3) The mechanical properties and the velocities of each cell can be modeled individually in a very efficient way.

In the model, each cell is described by an order parameter φ which takes the value of one within the cell and the value of zero elsewhere, similar to the grain growth phase-field model described earlier in *case study-II*. Each cell treated as 2D soft body for which the equilibrium shape minimizes the following free energy:

$$F_0 = \sum_n \left[\gamma_n \int_V \left[(\nabla \varphi_n)^2 + \frac{30}{\lambda^2} \varphi_n^2 (1 - \varphi_n)^2 \right] dV + \frac{\mu_n}{\pi R^2} \left(\pi R^2 - \int_V (\varphi_n^2) dV \right)^2 \right] \quad (4.53)$$

where λ corresponds to the width of the boundary of any cell. In the model, the noninteracting cells tend to be circular with a radius R . The energetic costs associated to change in cell area while keeping its volume approximately constant are determined by the parameter μ_n . It is shown that γ_n is the parameter that controls the elasticity of the cells. As can be seen, the first integrand corresponds to the standard non-conserved Allen–Cahn equation with gradient energy and the double-well potential for the order parameter.

The free energy above describes the cells individually; total energy accounting the interactions is,

$$F = F_o + F_{\text{int}} \quad (4.54)$$

in which F_{int} is taken as,

$$F_{\text{int}} = \frac{30\kappa}{\lambda^2} \int_V \left(\sum_{n,m \neq n} \varphi_n^2 \varphi_m^2 \right) dV \quad (4.55)$$

Again, it is similar to grain growth or sintering phase-field approaches given before in *case studies-II* and *IV*.

The time evolution of each cell is described by:

$$\frac{\partial \varphi_n}{\partial t} + v_n \cdot \nabla \varphi_n = -\frac{1}{2} \frac{\delta F}{\delta \varphi_n} \quad (4.56)$$

where v_n is the time-dependent cell velocity and is chosen to be spatially uniform in the model for the sake of simplicity. The velocity of each cell is divided into two parts,

$$v_n = v_{n,I} + v_{n,A} \quad (4.57)$$

where $v_{n,I}$ is the inactive part of the cell velocity, arising from the forces exerted on the cell by the other cells, and $v_{n,A}$ is the active part (self-propulsion) due to the internal process that require energy consumption. The constitutive

equations for $v_{n,I}$ can be chosen from thermodynamic principles and the following form is chosen for the model:

$$v_{n,I} = \frac{60\kappa}{\xi\lambda^2} \int_V \left(\varphi_n (\nabla \varphi_n) \sum_{m \neq n} \varphi_m^2 \right) \quad (4.58)$$

where ξ is interpreted as the friction between the cells and the surrounding liquid environment. Since, living cells never reach thermodynamic equilibrium; therefore, $v_{n,A}$ is chosen such that the instantaneous velocity of the isolated cells has a constant magnitude $|v_{n,A}| = v_{n,A}$ in the model.

By taking the functional derivatives in Eq. 4.56, the time evolution of cells is described by the following two equations:

$$\begin{aligned} \frac{\partial \varphi_n}{\partial t} &= \gamma_n \nabla^2 \varphi_n - \frac{30}{\lambda^2} \left[\gamma_n \varphi_n (1 - \varphi_n) (1 - 2\varphi_n) + 2\kappa \sum_{m \neq n} \varphi_n \varphi_m^2 \right] \\ &\quad - \frac{2\mu}{\pi R^2} \varphi_n \left[\int_V (\varphi_n^2) dV - \pi R^2 \right] - v_n \cdot \nabla \varphi_n \end{aligned} \quad (4.59)$$

and

$$v_n = v_{n,A} + \frac{60\kappa}{\lambda^2 \xi} \int_V \left(\varphi_n (\nabla \varphi_n) \sum_{m \neq n} \varphi_m^2 \right) \quad (4.60)$$

4.8.3 Numerical Implementation

Again, the Laplace operator and the directional derivatives in Eqs. 4.59 and 4.60 are approximated with finite difference algorithm by utilizing five-point stencil in two-dimensional space. The time integration is carried by simple explicit Euler time marching scheme. The nondimensionalized values of the parameters that appear in the equations are given in Table 4.3.

Table 4.3 The nondimensionalized values of the parameters used in the model

γ_n	λ	κ	μ	ξ
5.0 and 2.5	7	60	40	1.5e3

4.8.4 Results and Discussion

All the simulations were carried out for a square simulation cell having $N_x = 200$, $N_y = 200$ and the grid spacing $dx = dy = 1.0$. The nondimensionalized parameters given above were used in all simulations.

First simulation was carried out for two cells with identical elastic properties and size ($R = 25$) moving toward each other with opposite velocities. The magnitude of the self-propulsion $v_{n,A}$ for each cell was identical (with opposite sign) and is taken as 0.5 in nondimensionalized units. The time evolution of the collision of the cells is shown in Fig. 4.18. As can be seen, at first step the interface is almost sharp, at subsequent time steps, a diffusive interface forms. As cells contact each other, they deform while keeping their volume approximately constant. Opposite to that seen in earlier chapters for grain growth and sintering, in this case the cells neither penetrate nor absorb each other.

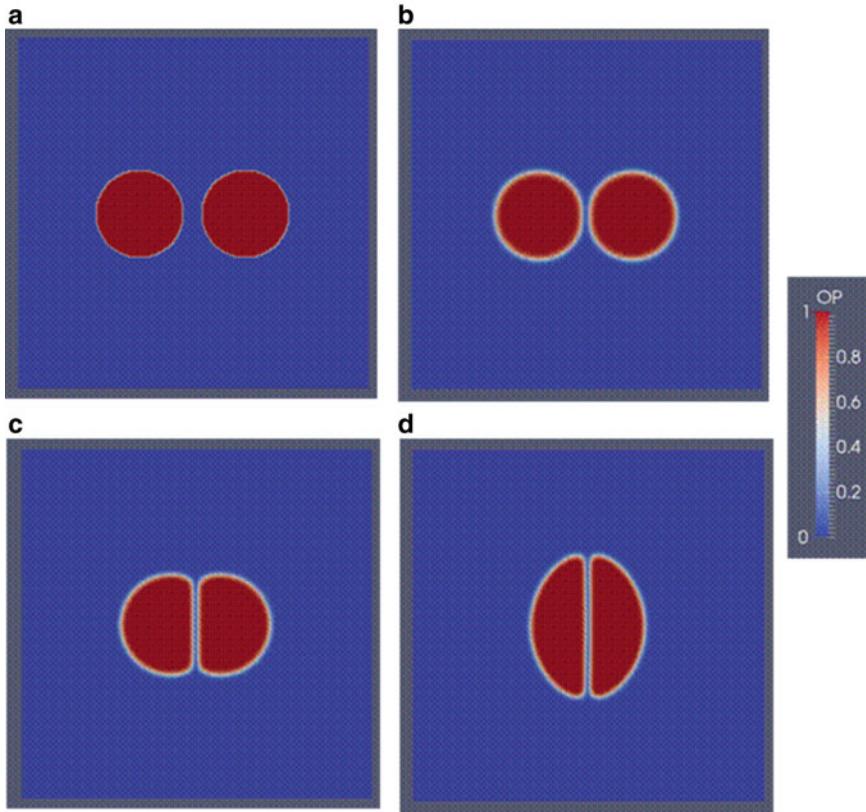


Fig. 4.18 Simulation results for two normal cells that are moving toward each other with opposite self-propulsion. (a) time step = 1, (b) time step = 700, (c) time step = 2000, and (d) time step = 5000

In the next set of simulations, the collective behavior of hard and soft cells (cancer cells) are considered as in original work in [1]. There was 69 cells, identical in size $R = 12$, in the simulations, and five of them designated as the soft cells (cancer cells). Their modulus γ_n was taken as 2.5, which is half of those normal cells. The magnitude of self-propulsion $v_{n,A}$ of each cell, including the cancer cells, was identical and taken as 0.2. However, their sign was also assigned randomly as either positive or negative.

The time evolution of collective motion of cells is shown in Fig. 4.19. As can be seen, since the cells were placed into the simulation cells randomly, there was small overlapping initially. During first two hundred steps, the

cells were only mobile with the interaction from other cells, the self-propulsion component was not included into their mobility. During this period, the initial overlapping is relaxed as cells deform slightly and rearrange themselves to minimize the free energy. The collective evolution behavior (A through F in Fig. 4.19) is excellent qualitative agreement with the results given in [1].

Although, no effort is made to track the displacements of the soft cells (cancer cells), the faster migration of soft cells resulting from their ability to deform easily can be clearly deduced from the figure and from the movies that are included together with the source codes, in subdirectory *case_study_5* in downloadable file.

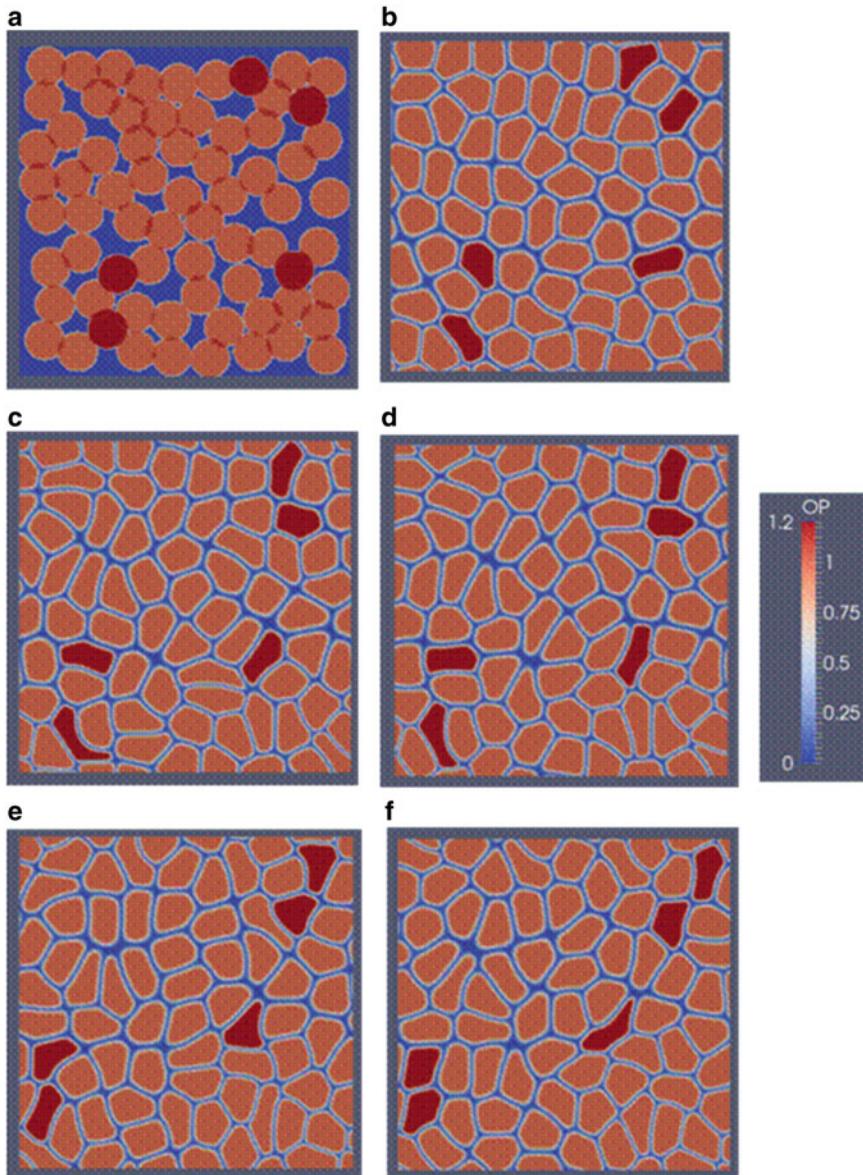


Fig. 4.19 Collective behavior of normal and soft cell (cancer cells) dark red. (a) time step = 1, (b) time step = 2500, (c) time step 7500, (d) time step 1000, (e) time step = 15,000, and (f) time step = 17,500

4.8.5 Source Codes

Two programs are provided in this section. The first one, *fd_cell_v1.m*, is in longhand format and *fd_ch_v2.m* is the optimized version for Matlab/Octave.

Program

fd_cell_dyna_v1.m

This program solves the phase-field modeling of the cell dynamics with finite difference algorithm. The code is in longhand format and not optimized for Matlab/Octave.

The program makes calls to the following functions:

- **micro_poly_cell.m**
- **free_energy_v1.m**
- **write_vtk_grid_values.m**

Listing:

```

1    %%%%%%%%%%%%%%%%
2    %                               %
3    % FINITE-DIFFERENCE PHASE-FIELD %
4    %      CODE FOR CELL-DYNAMICS   %
5    %%%%%%%%%%%%%%%%
6
7    %% get initial wall time:
8    time0=clock();
9    format long;
10
11 %% Simulation cell parameters:
12
13 Nx = 200;
14 Ny = 200;
15 NxNy= Nx*Ny;
16
17 dx = 1.0;
18 dy = 1.0;
19
20 %% Time integration parameters:
21
22 nstep = 3000;
23 nprint = 50;
24 dtim=5.e-3;
25
26 %% Material specific Parameters:
27
28 %%-
29 %ncell = 80;
30 %R= 12.0;
31 %%-
32 ncell = 2;
33 R =25.0;
34 %%-
35 %
36 %
37 gamma = 5.0;
38 lamda = 7.0;
39 kappa = 60.0;

40 mu   = 40.0;
41 kisa = 1.5e3;
42
43 %
44
45 pix=4.0*atan(1.0);
46 const1 = 30.0/lamda^2;
47 const2 = 2.0*mu/(pix*R^2);
48 const3 = 60.0*kappa/(lamda^2*kisa);
49
50 %---
51 %prepare initial microstructure:
52 %---
53
54 [ncell,phis,vac,nccel,ccell] =
55 micro_poly_cell(Nx,Ny,ncell,R);
56 %---
57 %- Evolution
58 %---
59
60 for istep=1:nstep
61
62 if(istep >= 500 )
63 dtim = 1.0e-2;
64 end
65
66 for icell=1:ncell
67
68 %% cell elasticity parameter:
69
70 gamma_cell = gamma;
71
72 for iccel = 1:nccel
73
74 if(icell == ccell(iccel))
75
76 gamma_cell = 0.5*gamma;
77
78 end
79 end
80
81 %% Assign cells
82
83 for i=1:Nx
84 for j=1:Ny
85 phi(i,j) = phis(i,j,icell);
86 end
87 end
88

```

```

89
90 %----
91 % calculate the laplacian and
92 % gradient terms:
93
94 for i=1:Nx
95 for j=1:Ny
96
97 jp=j+1;
98 jm=j-1;
99
100 ip=i+1;
101 im=i-1;
102
103 jp=j+1;
104 jm=j-1;
105
106 ip=i+1;
107 im=i-1;
108
109 if(im==0)
110 im=Nx;
111 end
112 if(ip==(Nx+1))
113 ip=1;
114 end
115
116 if(jm==0)
117 jm=Ny;
118 end
119
120 if(jp==(Ny+1))
121 jp=1;
122 end
123
124 hne=phi(ip,j);
125 hnw=phi(im,j);
126 hns=phi(i,jm);
127 hnn=phi(i,jp);
128 hnc=phi(i,j);
129
130 lap_phi(i,j) =(hnw + hne + hns
+ hnn -4.0*hnc) / (dx*dy);
131
132 %--gradients of phi:
133
134 phidx(i,j) = (phi(ip,j) -
phi(im,j))/(2.0*dx);
135 phidy(i,j) = (phi(i,jp)
- phi(i,jm))/(2.0*dy);
136
137 end %for j
138 end %for i
139
140 %---
141 %--- volum integrations:
142
143 vinteg = 0.0;
144 vintegx = 0.0;
145 vintegy = 0.0;
146
147 for i=1:Nx
148 for j=1:Ny
149
150 vinteg = vinteg + phi(i,j)^2;
151
152 sum_phi = 0.0;
153
154 for jcell =1:ncell
155 if(icell ~= jcell)
156
157 sum_phi = sum_phi +
phis(i,j,jcell)^2;
158 end
159 end
160
161 vintegx = vintegx + phi(i,j)*
phidx(i,j)*sum_phi;
162
163 vintegy = vintegy + phi(i,j)*
phidy(i,j)*sum_phi;
164 end
165 end
166
167
168 for i=1:Nx
169 for j=1:Ny
170
171 %-- Second term:
172 %-- derivative of free energy
173
174 [dfdphi]=free_energy_v1(i,j,icell,
ncell,gamma,kappa,phi,phis);
175
176 term2 = -const1*dfdphi;
177
178 %---

```

```

179 %---Third term
180
181 term3 = -const2*(vinteg-pix*R^2)
182 *phi(i,j);
183 %--- cell velocity:
184
185 if(istep<= 200 )
186 vac_cell = 0.0;
187 else
188 vac_cell=vac(icell);
189 end
190
191 %---cell velocity vector:
192
193 vnx= vac_cell + const3*vintegx;
194
195 vny= vac_cell + const3*vintegy;
196
197 if(ncell== 2)
198 vnx= vac_cell;
199 vny= 0.0;
200 end
201
202
203 %--- fourth term:
204
205 vnphi = vnx*phidx(i,j) + vny*
206 phidy(i,j);
207 term4x(i,j)=vnphi;
208
209 %-- time integration:
210
211 phi(i,j) = phi(i,j) + dtimex*
212 (gamma_cell*lap_phi(i,j) +...
213 term2+term3-vnphi);
214 end%for j
215 end%for i
216
217 for i=1:Nx
218 for j=1:Ny
219
220 %-- for small deviations:
221
222 if (phi(i,j) >= 0.9999);
223 phi(i,j) = 0.9999;
224 end
225 if(phi(i,j) < 0.000);
226 phi(i,j) = 0.0;
227 end
228 %---
229
230 phis(i,j,icell) = phi(i,j);
231
232 end%for j
233 end%for i
234
235 end%icell
236
237 %---- print results
238
239 if((mod(istep,nprint) == 0) ||
240 (istep == 1))
241 fprintf('done step: %d\n',istep);
242
243 %fname1 = sprintf('time_%d.out',
244 istep);
245 %out1 = fopen(fname1,'w');
246 %for icell=1:ncell
247 %fprintf(out1,'#cell No: %d\n',
248 icell);
249 %for i=1:Nx
250 %for j=1:Ny
251 %fprintf(out1,'%d %d %14.6e\n',
252 i,j,phis(i,j,icell));
253 %end
254 %end
255 %fclose(out1)
256
257 %--- write vtk file:
258
259
260 for i=1:Nx
261 for j=1:Ny
262 phi1(i,j) =0.0;
263 end
264 end
265
266 for icell=1:ncell
267
268 add = 1.0;
269
270 for iccel = 1:ncel
271 if(icell == ccell(iccel))
272 add = 1.2;
273 end

```

```

274 end
275
276
277 for i=1:Nx
278 for j=1:Ny
279
280 phi1(i,j) = phi1(i,j) + (phis(i,j,
icell)^2 * add);
281
282 end % for j
283 end % for i
284
285 end % for jcell
286
287 write_vtk_grid_values(Nx,Ny,dx,
dy,istep,phi1);
288
289 end %endif
290
291
292 end %istep
293
294 %--- calculate compute time:
295
296 compute_time = etime(clock(),
time0);
297 fprintf('Compute Time: %10d\n',
compute_time);

```

Line numbers:

7–8:	Get initial wall clock time beginning of the execution.
11–19:	Simulation cell parameters.
13:	Number of grid points in the x -direction.
14:	Number of grid points in the y -direction.
15:	Total number of grid points in the simulation cell.
17:	Grid spacing between two grid points in x -direction.
18:	Grid spacing between two grid points in y -direction.
20–25:	Time integration parameters.
22:	Number of time integration steps.
23:	Output frequency to write the results to file.
24:	Time increment for the numerical integration.
27–49:	Material-specific parameters.
30–31:	For poly-cell simulations set the desired number of cells and radius of the cells (in grid points).

33–34:	For two-cell simulations set the ncell = 2 and radius of the cells (in grid points).
37–41:	Set the values of the parameters (Eqs. 4.53 and 4.59, and Table 4.3).
45.48:	Pre-calculate the some of the expressions as constants (see Eqs. 4.55, 4.58 and 4.59).
50–55:	Prepare initial microstructure.
60–292:	Time evolution.
62–64:	After 500 time steps, increase the time increment to 0.01.
66–235:	Loop over each cell in the simulation.
68–80:	Assign the cell elasticity parameter depending on its character, determined by ccell.
81–87:	Assign current cell order parameter values from common array phis(Nx,Ny,ncell) to phi (Nx,Ny).
90–138:	Calculate the Laplacian and Cartesian derivative of the current cell with five-point stencil.
141–165:	Calculate the volume integrals that appears in Eqs. 4.58 and 4.60.
150:	Calculate the integral value in Eq. 4.58.
161:	Calculate the volume integral for x -component of the velocity vector in Eq. 4.60.
163:	Calculate the volume integral for y -component of the velocity vector in Eq. 4.60.
171–177:	For the current grid point, calculate the second term of Eq. 4.59.
179–182:	For the current grid point, calculate the third term of Eq. 4.59.
183–201:	Determine the values of the velocity vector of current cell.
185–187:	If time steps ≤ 200 do not include the self-propulsion.
188:	Otherwise take the randomly assigned value of self-propulsion value in function micro_poly_cell.
193:	Cell velocity value in the x -direction.
194:	Cell velocity value in the y -direction.
197–200:	If ncell = 2 take the self propulsion only in the x -direction.
203–206:	Calculate the fourth term in Eq. 4.59, $v_n \cdot \nabla \varphi_n$.
209–213:	Evolve φ with explicit Euler time integration scheme (Eq. 4.59).
217–234:	Return the current values of cell under consideration to common array phis(Nx,Ny,ncell).
220–228:	For small deviations, set the order parameters values to 0.9999 and 0.
237–289:	If print frequency reached, print the results to file.
243–255:	Open an output file and print the results to output file. These lines were commented out,

(continued)

	but they be changed, including the format of the output file.
257–288:	Write the results for contour plot in vtk format to be viewed by Paraview.
260–283:	Adds a small value to the order parameters of the soft cells, in order to distinguish them from the other cells during the contour plotting in Paraview.
296–297:	Calculate the total execution time and print it to screen.

Program

fd_cell_dyna_v2.m

This program solves the phase-field modeling of the cell dynamics with finite difference algorithm. The code is the optimized version of program *fd_cell_dyna_v1.m* for Matlab/Octave.

The program makes calls to the following functions:

- **micro_poly_cell.m**
- **laplacian.m**
- **gradient_mat.m**
- **vec2matx.m**
- **free_energy_v2.m**
- **write_vtk_grid_values.m**

Listing:

```

1  %%%%%%%%%%%%%%%%
2  %
3  % FINITE-DIFFRENCE PHASE-FIELD %
4  %      CODE FOR CELL-DYNAMICS %
5  % (OPTIMIZED FOR MATLAB/OCTAVE) %
6  %
7  %== get intial wall time:
8  time0=clock();
9  format long;
10 %
11 %-- Simulation cell parameters:
12 %
13 Nx = 200;
14 Ny = 200;
15 NxNy= Nx*NY;
16 %
17 dx = 1.0;

18 dy = 1.0;
19 %
20 %--- Time integration parameters:
21 %
22 nstep = 20000;
23 nprint = 100;
24 dtime=5.e-3;
25 %
26 %--- Material specific Parameters:
27 %
28 %-
29 ncell = 80;
30 R = 12.0;
31 %-
32 ncell = 2;
33 R = 25.0;
34 %-
35 %
36 gamma = 5.0;
37 lamda = 7.0;
38 kappa = 60.0;
39 mu = 40.0;
40 kisa = 1.5e3;
41 %
42 %-
43 pix=4.0*atan(1.0);
44 const1 = 30.0/lamda^2;
45 const2 = 2.0*mu/(pix*R^2);
46 const3 = 60.0*kappa/(lamda^2*kisa);

47 %---
48 %prepare microstructure:
49 %
50 %---
51 %
52 [ncell,phis,vac,nccel,cceil] =
53 micro_poly_cell(Nx,Ny,ncell,R);
54 %
55 %--- Get Laplacian templet:
56 [laplacian]=laplacian(Nx,Ny,
57 dx,dy);
58 %
59 %---
60 %- Evolution
61 %---
62 %
63 for istep =1:nstep
64

```

```

65 if(istep >= 500 )
66 dtme = 1.0e-2;
67 end
68
69 for icell =1:ncell
70
71 %--- cell elasticity parameter:
72
73 gamma_cell = gamma;
74 for iccel = 1:ncel
75
76 if(icell == ccell(iccel))
77
78 gamma_cell = 0.5*gamma;
79
80 end
81 end
82
83 %--- Assign cells:
84
85 phi =phis( :, :,icell);
86
87 %---
88 % calculate the laplacian and
89 gradient terms:
90
91 phi2 = reshape(phi',NxNy,1);
92 lap_phi2 = laplacian*phi2;
93 [lap_phi]=vec2matx(lap_phi2,Nx);
94
95
96 %--gradients of phi:
97
98 [phidy,phidx]=gradient_mat(phi,
99 Nx,Ny,dx,dy);
100
101 %--- Second term:
102 %--- derivative of free energy:
103
104 [dfdphi] =free_energy_v2(Nx,Ny,
105 icell,ncell,gamma_cell,kappa,
106 phi,phis);
107
108 %---
109
110 sum_phi =zeros(Nx,Ny) ;
111 for jcell =1:ncell
112
113 if(icell ~= jcell)
114 sum_phi =sum_phi + phis( :, :,jcell).^2;
115
116 end
117 end
118
119 %--- volum integrations:
120
121 dummy = zeros(Nx,Ny) ;
122 dummy = phi.^2;
123 vinteg =sum(sum(dummy));
124
125 %--
126
127 dummy = phi.*phidx.*sum_phi;
128 vintegx = sum(sum(dummy));
129
130
131 dummy = phi.* phidy .*sum_phi;
132 vintegy = sum(sum(dummy));
133
134
135 %--- Third term:
136
137 vinteg =vinteg - pix*R^2;
138
139 term3 =-const2*vinteg*phi;
140
141 %--- cell velocity:
142
143 if(istep <= 200 )
144 vac_cell = 0.0;
145 else
146 vac_cell = vac(icell);
147 end
148
149 %--- cell velocity vector:
150
151 vnx = vac_cell +const3*vintegx;
152
153 vny = vac_cell +const3*vintegy;
154
155 if(ncell == 2)
156 vnx = vac_cell;
157 vny = 0.0;
158 end
159
160 %--- fourth term:
161

```

```

162 vnpos = vnx*phidx + vny*phidy;
163
164 %-- time integration:
165
166 phi = phi + dtim* (gamma_cell *
167   lap_phi + term2 + term3 - vnpos);
168
169 %-- for small deviations:
170 inrange = (phi >= 0.9999);
171 phi(inrange) = 0.9999;
172 inrange = (phi < 0.000);
173 phi(inrange) = 0.0;
174
175 %---
176
177 phis( :, :, icell) = phi;
178
179 end %icell
180
181 %---- print results
182
183 if((mod(istep,nprint) == 0) ||
184   (istep == 1))
185 fprintf('done step: %5d\n', istep);
186
187 %fname1 = sprintf('time_%d.out',
188 istep);
189
190 %for icell =1:ncell
191 %fprintf(out1,'#cell No: %5d\n',
192 icell);
193 %for i=1:Nx
194 %for j=1:Ny
195 %fprintf(out1,'%5d %5d %14.6e\n',
196 i,j,phis(i,j,icell))
197 %end
198 %end
199 %end
200 %--- write vtk file:
201
202 phi1=zeros(Nx,Ny);
203
204 for jcell=1:ncell
205
206 add = 1.0;
207
208 for icel = 1:ncel
209 if(jcell == ccell(icel))
210 add = 1.2;
211 %fprintf('jcell %5d %5d\n', jcell,
212 ccell(icel));
213 end
214 end
215 phi1 = phi1 + (phis( :, :, jcell).^2 *
216 add);
217 end
218 write_vtk_grid_values(Nx,Ny,dx,
219 dy,istep,phi1);
220 end %end if
221
222 end %istep
223
224 %--- calculate compute time:
225
226 compute_time = etime(clock(),
227 time0);
228 fprintf('Compute Time: %10d\n',
229 compute_time);

```

Line numbers:

7–8:	Get initial wall clock time beginning of the execution.
11–19:	Simulation cell parameters.
13:	Number of grid points in the <i>x</i> -direction.
14:	Number of grid points in the <i>y</i> -direction.
15:	Total number of grid points in the simulation cell.
17:	Grid spacing between two grid points in the <i>x</i> -direction.
18:	Grid spacing between two grid points in the <i>y</i> -direction.
20–25:	Time integration parameters.
22:	Number of time integration steps.
23:	Output frequency to write the results to file.
24:	Time increment for the numerical integration.
26–47:	Material-specific parameters.
29–30:	For poly-cell simulations set the desired number of cells and radius of the cells (in grid points).

(continued)

32–33:	For two-cell simulations set the ncell = 2 and radius of the cells (in grid points).	160–163:	Calculate the fourth term in Eq. 4.59, $v_n \cdot \nabla \varphi_n$ for all grid points.
36–40:	Set the values of the parameters (Eqs. 4.53 and 4.59, and Table 4.3).	164–167:	Evolve φ with explicit Euler time integration scheme (Eq. 4.59).
44–46:	Pre-calculate the some of the expressions as constants (see Eqs. 4.55, 4.58, and 4.59).	168–174:	For small deviations, set the order parameters values to 0.9999 and 0.
49–53:	Prepare initial microstructure.	177:	Return the current value the cell under consideration to common array phis.
54–57:	Calculate the finite-difference template for the laplacian.	181–219:	If print frequency reached, print the results to file.
63–222:	Time evolution.	187–199:	Open an output file and print the results to output file. These lines were commented out, but they be changed, including the format of the output file.
65–67:	After 500 time steps, increase the initial time increment to 0.01.	200–219:	Write the results for contour plot in vtk format to be viewed by Paraview.
69–179:	Loop over each cell in the simulation.	202–217:	Adds a small value to the order parameters of the soft cells, in order to distinguish them from the other cells during the contour plotting in Paraview.
71–81:	Assign the cell elasticity parameter depending on its character, determined by ccell.	226–227:	Calculate the total execution time and print it to screen.
83–86:	Assign current cell order parameter values from common array phis(Nx,Ny,ncell) to phi(Nx,Ny).		
88–94:	Calculate the Laplacian of phi.		
91:	Convert phi(Nx,Ny) into one dimensional phi2(Nx*Ny) array.		
92:	Calculate $\nabla^2 \varphi$ for all grid points.		
93:	Convert lap_phi2(Nx*Ny) from one dimension array to lap_phi(Nx,Ny) square matrix.		
96–99:	Calculate the Cartesian derivative of φ , $\nabla \varphi$.		
101–107:	Calculate the second term of Eq. 4.59 for all grid points.		
110–117:	Calculate the summation in Eq. 4.60 for all cells except for the current cell.		
119–133:	Calculate the volume integrals that appear in Eqs. 4.58 and 4.59.		
121–125:	Calculate the volume integral in Eq. 4.58 for all grid points.		
127–128:	Calculate volume integral for the x -component of the velocity vector in Eq. 4.60 for all grid points.		
131–132:	Calculate volume integral for the y -component of the velocity vector in Eq. 4.60 for all grid points		
135–140:	Calculate the third term in Eq. 4.59 for all grid points.		
141–159:	Determine the velocity vector of the current cell for all grid points.		
143–145:	If time steps $<= 200$ do not include the self-propulsion.		
146:	Otherwise take the randomly assigned value of self-propulsion value in function micro_poly_cell.		
151:	Cell velocity value in the x -direction.		
153:	Cell velocity value in the y -direction.		
155–158:	If ncell = 2 take the self propulsion only in the x -direction.		

Function**micro_poly_cell.m**

This function introduces randomly distributed and slightly overlapping cells having radius of R into the simulation cell.

Variable and array list:

Nx:	Number of grid points of the simulation cell in the x -direction.
Ny:	Number of grid points of the simulation cell in the y -direction.
ncell:	Input as desired cell number, output is the actual number of cell generated.
R:	The radius of the cells in terms of number of grid points.
ncel:	Number of soft cells
vac(ncell):	The self-propulsion values of the cells.
ccell(ncel):	Cell number of soft cells.
phis(Nx,Ny, ncell):	Common array containing the order parameter values of the cells.

Listing:

```

1  function [ncell,phis,vac,ncel,
2   ccell] = micro_poly_cell(Nx,Ny,
3   ncell,R)
4
5  format long;

```

```

4
5  %--- initialize:
6
7  %for icell =1:ncell
8  %for i=1:Nx
9  %for j=1:Ny
10 %phis(i,j,icell) = 0.0;
11 %end
12 %end
13 %end
14
15 phis=zeros(Nx,Ny,ncell);
16
17 %---
18
19 if(ncell == 80)
20
21 R2 =2.0*R;
22 Rsq = R*R;
23
24 ncell =0;
25 xmin =0.0;
26 ymin =0.0;
27
28 xmax=Nx;
29 ymax=Ny;
30
31
32 xc =zeros(64);
33 yc= zeros(64);
34
35 for iter =1:500000
36
37 xnc =Nx*rand( );
38 ync =Ny*rand( );
39
40 iflag =1;
41
42 if(((xnc-R) < xmin) || ((xnc+R)
43 > xmax))
44 iflag =0;
45 end
46
47 if(((ync-R) < ymin) || ((ync+R)
48 > ymax))
49 iflag =0;
50 end
51 if(iflag ==1)
52
53 xdist =sqrt((xc(i)-xnc)^2 +(yc(i)-
54 ync)^2);
55 if(xdist <= R*1.6)
56 iflag =0;
57 end
58 end
59
60 end%if
61
62
63 if(iflag ==1)
64
65 ncell =ncell+1;
66 xc(ncell) =xnc;
67 yc(ncell) =ync;
68
69
70 for i=1:Nx
71 for j=1:Ny
72
73 if((i-xnc)^2 +(j-ync)^2 <Rsq)
74 phis(i,j,ncell) =0.999;
75 end
76
77 end
78 end
79
80 end%iflag
81
82
83 if(ncell == 80)
84 break;
85 end
86
87 end% irand
88
89 %--- randomize cell self propulsion
90 %velocities:
91 velc = 0.2;
92
93 for i=1:ncell
94
95 ix =rand( );
96 if(ix <= 0.5)
97 vac(i) = -velc;
98 else
99 vac(i) = velc;

```

```

100 end
101 end
102
103
104 fprintf('iteration done:
105 %d\n',iter);
106 fprintf('number of cell created
107 %d\n',ncell);
108
109 nccel = 5;
110 ccell(1) = 32;
111 ccell(2) = 11;
112 ccell(3) = 16;
113 ccell(4) = 21;
114 ccell(5) = 46;
115
116 end %if ncell=80
117
118 %---
119 %- For two cell simulation:
120 %---
121
122 if(ncell == 2)
123
124 nccel = 0;
125 ccell(1)=10000;
126
127
128 R2 = R*R;
129
130 xc(1,1)=Nx/2-1.25*R;
131 yc(1,1)=Ny/2;
132
133 xc(1,2)=Nx/2+1.25*R;
134 yc(1,2)=Ny/2;
135
136 ncol=1;
137 nrow=2;
138
139 icell=0;
140
141 for icol=1:ncol
142
143 for irow=1:nrow
144
145 icell = icell+1;
146 dx = xc(icol,irow);
147 dy = yc(icol,irow);
148
149
150 for i=1:Nx
151 for j=1:Ny
152
153 if((i-dx)*(i-dx) + (j-dy)*(j-dy)
154 < R2)
155 phis(i,j,icell) = 0.999;
156
157 end
158 end
159
160 end
161 end
162
163 %--- cell self propulsion velocity:
164
165 vac(1) = 0.5;
166 vac(2) = -0.5;
167
168 end %if ncell ==2
169
170 end %end function

```

Line numbers:

5–16:	Initialize the order parameter of the cells.
7–13:	Longhand initialization.
15:	Matlab/Octave initialization for optimization.
19–117:	Generate slightly overlapping cell microstructure, if ncell > 2.
21–22:	Calculate the diameter and the square of cell radius, R.
24:	Initialize the total number cells in the simulation.
25–29:	Determine the minimum and maximum cell dimensions in the x and y directions.
32–33:	Initialize the coordinates of the cell centers.
35–87:	Start randomly introducing cells into the simulation cell.
37–38:	Randomly assign cell center coordinates.
40:	Set iflag = 1. In the following tests, if its value still remains as iflag = 1, a new cell will be generated.
42–48:	Check, if new cell as a whole is in the bounding box of the simulation cell. If not set iflag = 0.
50:	If new cell fits into the simulation cell continue.
51–58:	Calculate the distance to the previously generated cell centers. If distance <= 1.6R Set iflag = 0.

(continued)

63–75:	After two tests above, if iflag = 1, set the order parameter of the new cell to 0.9999.
83:	If the total number of cell equals to 80 exit from the loop irand.
89–101:	Randomly change the sign of self-propulsion values of the cells.
91:	Absolute value of the self-propulsion value of the cells.
104–105:	Print how many cells are generated after 500000 random steps to screen. This number can be set to larger value if more cell needs to be generated.
107–114:	Assign the following cell numbers as soft cells.
128–168:	If ncell = 2, place two cells into the center of the simulation cell with having radius R and assign their self-propulsion values.

```

6
7
8 sum_phi =0.0;
9
10 for jcell =1:ncell
11 if(icell ~= jcell)
12 sum_phi = sum_phi + phis
13 (i,j,jcell)^2;
14 end
15
16 dfdphi = gamma*phi(i,j)*(1.0-
17 phi(i,j))*(1.0-2.0*phi(i,j)) + ...
18 2.0*kappa*phi(i,j)*sum_phi;
19
20 end %end function

```

Function

free_energy_v1.m

This function evaluates the functional derivative of free energy function, Eq. 4.59, for the current cell under consideration at current grid locations given by the main program.

Variable and array list:

i:	Current grid id in the <i>x</i> -direction.
j:	Current grid id in the <i>y</i> -direction.
icell:	Cell id of the current cell being considered in the main program.
ncell:	Total number of cell in the simulation.
gamma:	Parameter in the free energy functional, Eq. 4.59.
kappa:	Parameter in the free energy function, Eq. 4.59.
dfdphi:	Functional derivative of the free energy.
phi(Nx,Ny):	The value of the order parameter at grid point(i,j).
phis(Nx,Ny, ncell):	Common array containing the order parameters of the cells.

Listing:

```

1 function [dfdphi] =free_energy_v1
2   (i,j,icell,ncell,gamma,kappa,
3   phi,phis)
4
5   format long;

```

Line numbers:

8:	Initialize sum_phi.
10–14:	Go over total number of cells in the simulation if jcell counter is not icell add their value to sum_phi.
15–17:	Calculate the functional derivative of the free energy for the current cell, the terms in the first bracket, Eq. 4.59, at the current grid point location.

Function

free_energy_v2.m

This function is the optimized version of the function *free_energy_v1.m*. It evaluates the functional derivative of free energy function Eq. 4.59 at all grid points for the current cell under consideration in the main program.

Variable and array list:

Nx:	Number of grid points in the <i>x</i> -direction.
<b b="" ny:<="">	Number of grid points in the <i>y</i> -direction.
icell:	Cell id of the current cell being considered in the main program.
ncell:	Total number of cell in the simulation.
gamma:	Parameter in the free energy functional, Eq. 4.59.
kappa:	Parameter in the free energy function, Eq. 4.59.
dfdphi(Nx, Ny):	Functional derivative of the free energy.

(continued)

phi(Nx,Ny):	The value of the order parameter at grid point(i,j).
phis(Nx,Ny, ncell):	Common array containing the order parameters of the cells.

Listing:

```

1 function [dfdphi] =free_energy_v2
2 % Nx,Ny,icell,ncell,gamma,
3 % kappa,phi,phis)
4
5 format long;
6
7 sum_phi=zeros(Nx,Ny);
8
9 for jcell=1:ncell
10 if(icell ~= jcell)
11 sum_phi = sum_phi + phis
12 ( :, :,jcell) .^2;
13 end
14 end
15 dfdphi = gamma*phi.* (1.0-phi).* ...
16 (1.0-2.0*phi) + 2.0*kappa*
17 phi.*sum_phi;
18
19 end %end function

```

Line numbers:

5:	Initialize sum_phi for all grid points.
7-11:	Go over total number of cells in the simulation if jcell counter is not equal to icell add their value to sum_phi for all grid points.
12-14:	Calculate the functional derivative of the free energy for the current cell, the terms in the first bracket, Eq. 5.4.7, at all grid points.

References

- Palmieri B, Bresler Y, Wirtz D, Grant M (2015) Multiple scale model for cell migration in monolayers: elastic mismatch between cells enhances motility. *Nat Sci Rep.* doi:[10.1038/srep11745](https://doi.org/10.1038/srep11745)
- Shao D, Levine H, Rappel WJ (2012) Coupling actin flow, adhesion and morphology in a computational cell motility model. *Proc Natl Acad Sci U S A* 109:6851
- Honda H, Tanemura M, Nagai T (2004) A three-dimensional vertex dynamics cell model of space-filling polyhedra simulating cell behavior in cell aggregate. *J Theor Biol* 226:439
- Nagai T, Honda H (2001) A dynamic cell model for the formation of epithelial tissues. *Phil Mag B: Phys Cond Mater* 81:699
- Honda H (1978) Description of cellular patterns by Dirichlet domains: two dimensional case. *J Theor Biol* 72:523
- Honda H, Yamanaka H, Dan-Sohkawa M (1984) A computer simulation of geometrical configurations during cell division. *J Theor Biol* 106:423
- Glazier JA, Graner F (1993) Simulation of the differential adhesion driven rearrangement of biological cells. *Phys Rev E* 47:2128
- Graner F, Glazer JA (1992) Simulation of biological cell sorting using a two dimensional extended Potts model. *Phys Rev Lett* 69:2013.
- Nonomura M (2012) Study on multicellular systems using phase-field model. *PLoS ONE* 7:ee33501. doi:[10.1371/journal.pone.0033501](https://doi.org/10.1371/journal.pone.0033501)
- Shao D, Rappel WJ, Levine H (2010) Computational model for cell morphodynamics. *Phys Rev Lett* 105:108104
- Najem S, Grant M (2013) A phase-field model for neural cell chemotropism. *Europhys Lett* 102:16001
- Najem S, Grand M (2014) The chaser and the chased: a phase-field model of immune response. *J Soft Matter* 10:9715
- Vilanova G, Colominas I, Gomez H (2013) Capillary networks in tumor angiogenesis: from discrete endothelial cells to phase-field averaged descripts via isogeometric analysis. *Int J Num Meth Biomed Eng.* doi:[10.1002/cnm.2552](https://doi.org/10.1002/cnm.2552)
- Lowengrub JS, Ratz A, Voigt A (2009) Phase-field modeling of dynamics of multicomponent vesicles: spinodal decomposition, coarsening, budding and fission. *Phys Rev E Stat Nonlinear Soft Matter Phys* 79:031926
- Yun A, Lee SH, Kim J (2013) A phase-field model for articular cartilage regeneration in degradable scaffolds. *Bull Math Biol* 75:2389
- Sciume G, Shelton S, Gray WG, Miller CT, Hussain F, Ferrari M, Schrefler BA (2013) A multiphase model for three-dimensional tumor growth. *New J Phys* 15:015005
- Ziebert F, Swaminathan S, Aronson I (2012) Model for self-polarization and motility of keratocyte fragments. *J R Soc Interface* 9:1084
- Najem S, Grant M (2013) Phase-field approach to chemotactic driving of neutrophil morphodynamics. *Phys Rev E* 88:034702
- Najem S, Grant M (2015) Coupling actin dynamics to phase-field in modeling neural growth. *Soft Mater* 11:4476
- Takaki T, Nakagawa K, Morita Y, Nakamachi E (2015) Phase-field modeling for axonal extension of nerve cells. *Jpn Soc Mech Eng (JSME)* 2:50063
- Lima EABF, Oden JT, Almehda RC (2014) A hybrid ten-species phase-field model of tumor growth. *Math Models Methods Appl Sci* 24:2569

Solving Phase-Field Models with Fourier Spectral Methods

5.1 Introduction

As described in Chap. 2, finite difference and finite element methods have local character and the unknown functions are interpreted by usually low-order polynomials over small sub-domains. In contrast, spectral methods make use of global representation, usually with high-order polynomials or Fourier series. The rate of convergence of spectral approximations depends only on the smoothness of the solution. They achieve much higher accuracy with much smaller number of sampling points in comparison to other two methods. This fact is known in the literature as “spectral accuracy.” The spectral methods most often are successful with domains in periodic nature, which is the case in most of the phase-field modeling simulations. Again, the application of the Fourier spectral method will be demonstrated to the solution of one-dimensional transient heat conduction in this section. This source code, solving this simple problem given below, forms the foundation of the algorithms that will be developed in this chapter.

There are numerous textbooks and lecture notes available for spectral methods, of which some of them are listed in the reference section, [1–6].

5.2 One-Dimensional Transient Heat Conduction: A Solution with Fourier Spectral Algorithm

Recall Eq. 4.9, for constant thermal conductivity, density, and heat capacity, the one-dimensional heat conduction equation without heat-generating sources was expressed as:

$$\frac{\partial T}{\partial t} = \mu \frac{\partial^2 T}{\partial x^2}, \quad \text{and} \quad \mu = \frac{\lambda}{\rho c_p} \quad (5.1)$$

where ρ is the density, c_p is the heat capacity, λ is the thermal conductivity, T is the temperature, t is the time, and x is the distance. The solution of this equation with Fourier spectral method starts with forward Fourier transform of both sides of the equation

$$\frac{\partial \{T\}_k}{\partial t} = \mu \left\{ \frac{\partial^2 T}{\partial x^2} \right\}_k \quad (5.2)$$

where $\{\cdot\}_k$ is the Fourier transform of the quantity inside the bracket and k is the coefficient of the k th Fourier mode. The general relationship for spatial derivatives in Fourier space is given as

$$\left\{ \frac{\partial^n u}{\partial x^n} \right\}_k = \left((\sqrt{-1})k \right)^n \{u\}_k \quad (5.3)$$

Then, in Fourier space, Eq. 5.2 becomes

$$\frac{\partial \{T\}_k}{\partial t} = -\mu k^2 \{T\}_k \quad (5.4)$$

By taking forward difference for time derivative

$$\frac{\{T\}_k^{n+1} - \{T\}_k^n}{\Delta t} = -\mu k^2 \{T\}_k^n \quad (5.5)$$

in which Δt is the time between the time steps $n + 1$ and n . With forward Euler time marching, for the time step $n + 1$ results in:

$$\{T\}_k^{n+1} = \{T\}_k^n - \mu k^2 \Delta t \{T\}_k^n \quad (5.6)$$

Note that Eq. 5.6 is evaluated at Fourier space, and the results are converted back to real space with an inverse Fourier transformation.

The program *heat_1d_fft.m* given below utilizes the above steps to solve the one-dimensional heat conductivity problem. The program uses the identical parameters as given in *heat_1d_fd.m*. However, in this case the domain is assumed to be periodic without any imposed boundary conditions. The graphical results are shown in Fig. 5.1. As can be seen, while the initial temperature at the center region decays overtime, the temperature rises at the other regions resulting from the thermal conductivity and heat flow.

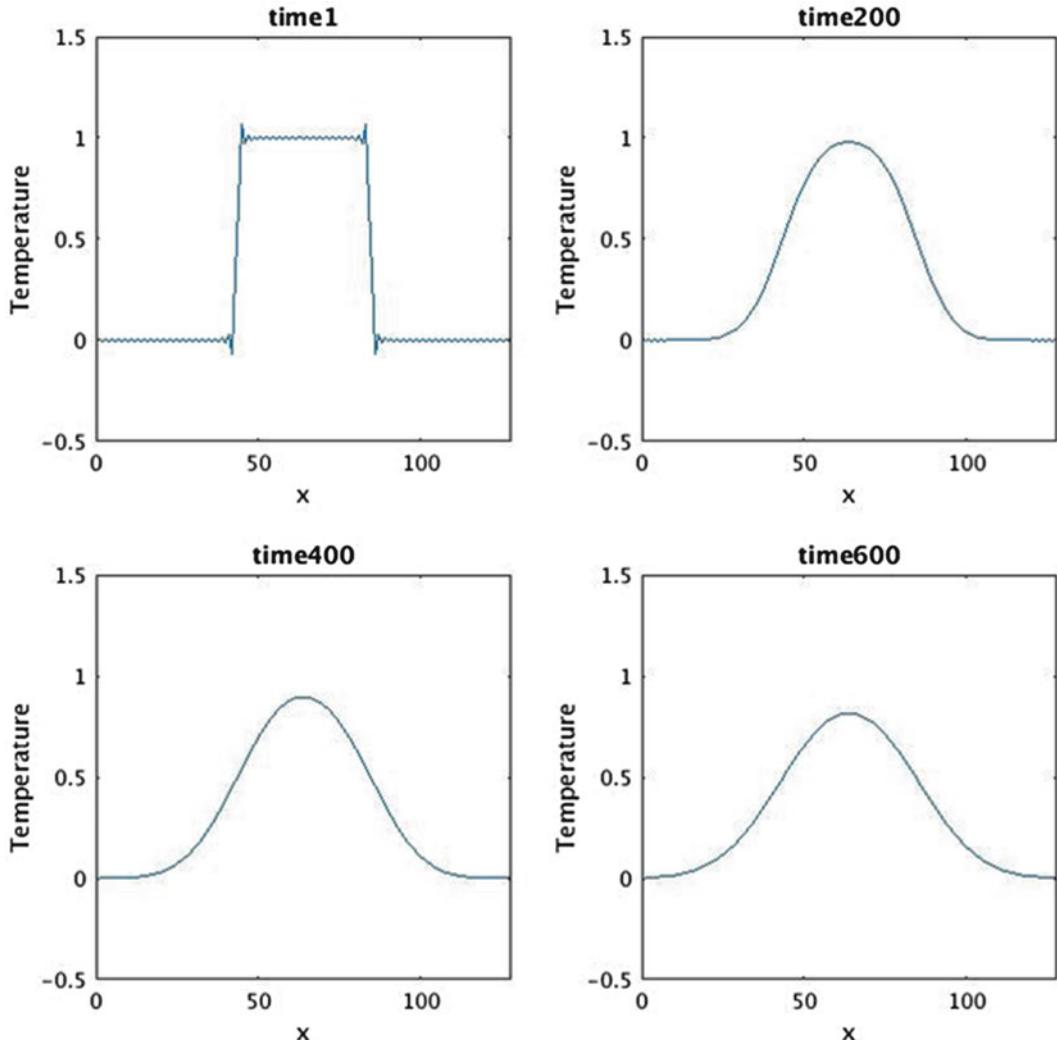


Fig. 5.1 Graphical outputs from the program *heat_1d_fft.m* during the solution of one-dimensional heat conductivity problem

Again, the results seen in the figure agree well with the analytical solutions. However, this is true for only range of dx and Δt values. The use of larger values results in instabilities, oscillations, and unacceptable results owing to the coarse spatial and temporal discretization.

5.3 Source Code

Program

heat_1d_fft.m

This program solves one-dimensional transient heat conductivity problem with Fourier spectral algorithm and explicit Euler time marching scheme.

Listing:

```

1 %%%%%%%%%%%%%%%%
2 % 1D transient heat conduction %
3 % space discretization: Fourier %
4 %           spectral %
5 % time integration: Explicit Euler %
6 %%%%%%
7 %-- get initial wall time:
8 time0=clock();
9
10 %--- Simulation cell parameters:
11
12 Nx=128;
13 dx=1.0;
14
15 %--- Time integration parameters:
16
17 nstep = 600;
18 nprint = 200;
19 dtim = 0.2;
20
21 %-- Initialize temperature field
22 % & grid:
23 for i=1:Nx
24 u0(i)=0.0;
25 x(i)=i*dx;
26
27 if((i >= 44) && (i <= 84))
28 u0(i)=1.0;
29 end
30
31 end
32
33 %-- Set Fourier coefficients:
34
35 Nx21=Nx/2 + 1;
36 Nx2=Nx+2;
37 delkx=(2.0*pi)/(Nx*dx);
38 %
39 for i=1:Nx21
40 fk1=(i-1)*delkx;
41 kx(i)=fk1;
42 kx(Nx2-i)=-fk1;
43 end
44 %
45 for i=1:Nx
46 k2(i)=kx(i)*kx(i);
47 end
48
49 %--
50 %-- Evolve temperature field:
51 %--
52
53 ncount=0;
54 for istep=1:nstep
55
56 fu0=fft(u0);
57
58 for i=1:Nx
59 fu0(i)=fu0(i)-dtim*k2(i)*fu0(i);
60 end
61
62 u0 =real(ifft(fu0));
63
64 %-- Display results:
65 if((mod(istep,nprint) == 0) || (istep
66 ==1))
67 ncount=ncount+1;
68 subplot(2,2,ncount);
69 plot(x,u0);
70 time=sprintf('%d',istep);
71 title(['time' time]);
72 axis([0 Nx -0.5 1.5]);
73 xlabel('x');
```

```

74 ylabel('Temperature');
75
76 end%if
77 end%istep
78
79 compute_time=etime(clock(),time0);
80 fprintf('Compute_time: %5d\n', comp
    utetime);

```

Line numbers:

8:	Get wall clock time beginning of the execution.
10–14:	Simulation cell parameters.
12:	Number of grid points in the x -direction.
13:	Grid spacing between two grid points in the x -direction.
15–20:	Time integration parameters.
17:	Number of time steps.
18:	Print frequency to output the results.
19:	Time increment.
21–31:	Initialize the initial temperature field and grid point coordinates.
33–48:	Set Fourier coefficients.
50–77:	Evolve temperature field.
56:	Take the current temperature field from real space to Fourier space.
58–60:	Evaluate Eq. 5.6 at every grid points in the Fourier space.
62:	Bring back the results from Fourier space to real space with inverse Fourier transformation.
65–76:	If input frequency is reached, output the results as plots.
79–80:	Compute the execution time and print it.

References

1. Krishnamurti TN, Bedi HS, Hardiker VM, Ramaswamy L (2006) An introduction to global spectral modeling. Springer, New York
2. Canuto C, Hussaini MY, Quarteroni A, Zhang TA (2006) Spectral methods, fundamentals in single domains. Springer, Scientific computation, New York
3. Boyd JP (2001) Chebyshev and Fourier spectral methods, 2nd edn. New York, Dover Publications
4. Shen J, Tao T, Wang L (2011) Spectral methods. Springer, New York
5. Trefethen LN (2000) Spectral methods in matlab. Siam, Philadelphia
6. Ben-yu G (1998) Spectral methods and their applications. World Scientific publishing, Singapore

5.4 Case Study-VI**Simulation of spinodal decomposition of a binary alloy with semi-implicit Fourier spectral algorithm****Objectives:**

The objective of this case study is to demonstrate a numerical implementation of the semi-implicit Fourier spectral algorithm for the solution of conserved Cahn–Hilliard equation in phase-field modeling.

5.4.1 Background

The background information regarding this case study has already been described in *Case Study-I* in which the Cahn–Hilliard equation was solved by utilizing finite difference algorithm. For convenience, the phase-field model is briefly summarized in here again.

5.4.2 Phase-Field Model

The evolution equation for the Cahn–Hilliard equation (Chap. 1, Eq. 1.1) can be expressed as:

$$\frac{\partial c}{\partial t} = \nabla^2 M \left(\frac{\delta F}{\delta c} \right) \quad (5.7)$$

Recall Eqs. 4.12 and 4.13, the total free energy, F , in its simplest form taken as:

$$F = \int_V \left[f(c) + \frac{1}{2} \kappa (\nabla c)^2 \right] dv \quad (5.8)$$

and the chemical energy $f(c)$ is described as simple double-well potential:

$$f(c) = Ac^2(1 - c)^2 \quad (5.9)$$

in which κ is the gradient energy coefficient and A is a positive constant and controls the

magnitude of the energy barrier between two equilibrium phases (see Fig. 4.3).

5.4.3 Numerical Implementation

The semi-implicit spectral algorithm implementation of Eq. 5.7 given below follows the work of Chen and Shen [1].

By taking functional derivative in Eq. 5.7, the evolution equation reads

$$\frac{\partial c}{\partial t} = \nabla^2 M \left[\frac{\delta f}{\delta c} - \kappa \nabla^2 c \right] \quad (5.10)$$

By taking Fourier transform of both side of Eq. 5.10 the spatial discretization becomes:

$$\frac{\partial \{c\}_k}{\partial t} = -k^2 M \left[\left\{ \frac{\delta f}{\delta c} \right\}_k + k^2 \kappa \{c\}_k \right] \quad (5.11)$$

where $\{\cdot\}_k$ is the Fourier transform of the quantity inside the bracket and k is the vector in Fourier space, $k = (k_1, k_2)$, with a magnitude $\sqrt{k_1^2 + k_2^2}$. By expending Eq. 5.11

$$\frac{\partial \{c\}_k}{\partial t} = -k^2 M \left\{ \frac{\delta f}{\delta c} \right\}_k - k^4 M \kappa \{c\}_k \quad (5.12)$$

By treating linear and fourth-order operators implicitly and the nonlinear terms explicitly, the semi-implicit form for Eq. 5.12 is:

$$\frac{\{c\}_k^{n+1} - \{c\}_k^n}{\Delta t} = -k^2 M \left\{ \frac{\delta f}{\delta c} \right\}_k^n - k^4 M \kappa \{c\}_k^{n+1} \quad (5.13)$$

where Δt is the time increment between time steps $n + 1$ and n . By rearranging it becomes

$$\{c\}_k^{n+1} = \frac{\{c\}_k^n - \Delta t k^2 M \left\{ \frac{\delta f}{\delta c} \right\}_k^n}{1 + \Delta t k^4 M \kappa} \quad (5.14)$$

and the problem reduces to solving the discretized Eq. 5.14.

The numerical steps to achieve this can be summarized as:

For number of time steps repeat:

1. Evaluate $(\delta f / \delta c)$ using the current time step value of c . Fourier transform $(\delta f / \delta c)$ and c .
2. Calculate the spatial variation of c in Fourier space for the time $t + \Delta t$ using Eq. 5.14.
3. With inverse Fourier transformation bring back the results to real space as the solution at $t + \Delta t$.

5.4.4 Results and Discussion

The simulation was carried out for a square simulation cell having $N_x = 64$ and $N_y = 64$ and with $dx = dy = 1.0$. All the parameters used in these simulations were identical to that used for *Case Study-I*. Time evolution of the microstructure due to phase separation during the course of simulation is summarized in Fig. 5.2. As can be seen at nondimensional time 75 the microstructure is relatively fine and contains a large number of precipitates. As time progresses, coarsening of the second phase through migration of the phase boundaries, dissolution, merging, and breakup can be easily inferred from the figure. As can be seen, the growth mainly occurs with Ostwald ripening process in which the small precipitates dissolve and are absorbed by the larger ones, and as a result the number of the precipitates becomes smaller with time. An animation file summarizing this simulation is given in subdirectory *case_study_6* in the downloadable file.

Figure 5.3 compares the reduction of the total energy during the course of these simulations with the one obtained in *Case Study-I* in which Cahn–Hilliard equation is solved with the explicit Euler time integration and the finite difference algorithm in spatial discretization.

As can be seen, even though there is slight variation in the randomness of the initial microstructures, the results obtained from both solution algorithms agree very well with each other.

As given in [1], the semi-implicit Fourier spectral method enables to use smaller number

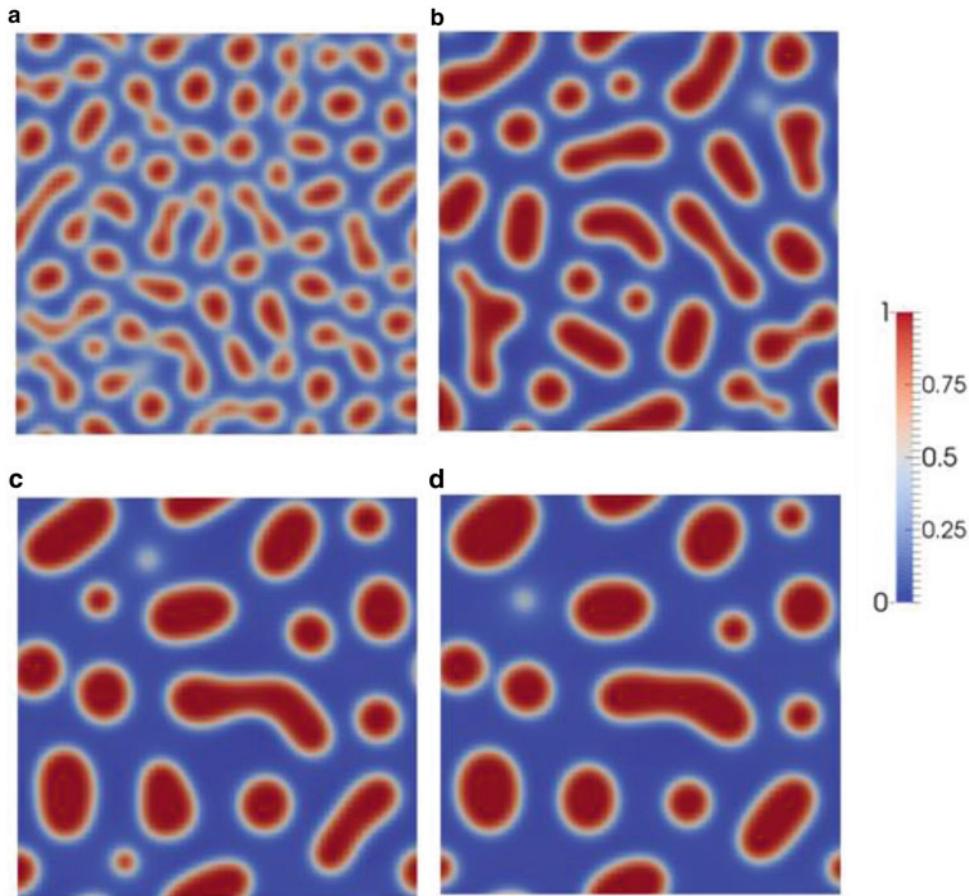


Fig. 5.2 Time evolution of microstructure as a result of phase separation. Nondimensional times are: (a) 75, (b) 100, (c) 150, and (d) 200

of grid points to resolve the spatial discretization owing to the exponential convergence of the Fourier spectral method. In addition, it relaxes the very strict time step size of explicit Euler time marching scheme and permits much larger time step sizes in the solution.

5.4.5 Source Code

Two source codes, *fft_ch_v1.m* in longhand format and corresponding Matlab/Octave optimized version *fft_ch_v2.m* with the associated functions are given below.

Program

fft_ch_v1.m

This program solves Cahn–Hilliard phase-field equation with Fourier spectral method. The time integration is carried out by using semi-implicit time marching scheme. The code is longhand format and is not optimized.

The program makes calls to the following functions:

- **prepare_fft.m**
- **free_energ_ch_v1.m**
- **calculate_energ.m**
- **write_vtk_grid_values.**

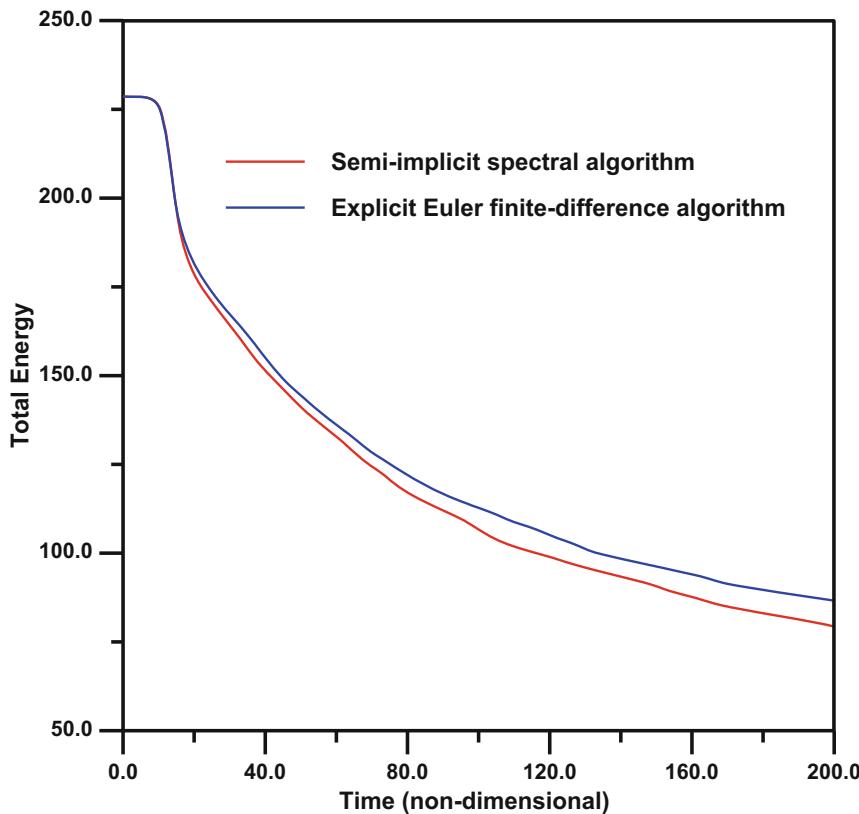


Fig. 5.3 Comparison of the reduction in total energy with time during the phase separation obtained from the semi-implicit spectral and the explicit Euler finite difference algorithms

Listing:

```

1  %%%%%%%%
2  %
3  %      SEMI-IMPLICIT SPECTRAL    %
4  %          PHASE-FIELD CODE       %
5  % FOR SOLVING CAHN-HILLIARD EQUATION %
6  %
7  %==get intial wall time:
8  time0=clock();
9  format long;
10
11
12  out2=fopen('time_energ.out','w');
13
14  %% Simulation cell parameters:
15
16  Nx=64;
17  Ny=64;
18  NxNy =Nx*Ny;
19  dx=1.0;
20  dy=1.0;
21
22  %--- Time integration parameters:
23
24  nstep = 2000;
25  nprint= 50;
26  dtime = 1.0e-2;
27  ttime = 0.0;
28  coefA = 1.0;
29
30  %--- Material specific Parameters:
31
32  c0 =0.40;
33  mobility =1.0;
34  grad_coef=0.5;
35

```

```

36  %--- Prepare microstructure           81
37                                         82  end
38  iflag = 1;                           83  end
39                                         84
40  [con] = micro_ch_pre(Nx,Ny,c0,      85  %-
41    iflag);                           86
42  %-- Prepare fft:                   87  con = real(ifft2(conk));
43                                         88
44  [kx,ky,k2,k4] = prepare_fft(Nx,Ny, 89  %% for small deviations:
45    dx,dy);                           90
46  %%==                                91  for i=1:Nx
47  %%--- Evolve:                      92  for j=1:Ny
48  %%==                                93
49  for istep=1:nstep                  94  if(con(i,j) >= 0.9999);
50                                         95  con(i,j) = 0.9999;
51                                         96  end
52  ttime = ttime + dtime;            97
53                                         98  if(con(i,j) < 0.00001);
54  conk = fft2(con);                99  con(i,j) = 0.00001;
55                                         100 end
56  %%--- derivative of free energy:   101
57                                         102 end
58  for i=1:Nx                      103 end
59  for j=1:Ny                      104 %%=
60                                         105 %%--- print results:
61  [dummy]=free_energ_ch_v1(i,j,con); 106 %%=
62  dfdcon(i,j)=dummy;              107
63  end                               108 if((mod(istep,nprint) == 0) ||
64  end                               109   (istep == 1))
65                                         110 fprintf('done step: %5d\n',istep);
66                                         111
67  %%--                                112 fname1 = sprintf('time_%d.out',
68                                         113   istep);
69  dfdconk = fft2(dfdcon);          114 out1 = fopen(fname1,'w');
70  %%--- Time integration:          115 for i=1:Nx
71  %%for i=1:Nx                    116 for j=1:Ny
72  %%for j=1:Ny                    117 fprintf(out1,'%5d%5d%14.6e\n',i,j,
73  %%numer=dtime*mobility*k2(i,j)* 118 con(i,j));
74  %%dfdconk(i,j);                119 end
75                                         120
76  %%denom = 1.0 + dtime*coefA*mobility* 121 fclose(out1);
77  %%grad_coef*k4(i,j);          122
78                                         123
79                                         124 %%--- write vtk file:
80  %%conk(i,j)=(conk(i,j)-numer)/ 125
81  %%denom;                      126 %%-- calculate total energy
82                                         127

```

```

128 [energ] = calculate_energ(Nx,Ny,con,
grad_coef);
129
130 fprintf(out2,'%14.6e %14.6e\n',
ttime,energ);
131
132 write_vtk_grid_values(Nx,Ny,dx,
dy,istep,con);
133
134 end %end if
135
136 end %istep
137
138 %--- calculate compute time:
139
140 compute_time = etime(clock(),time0);
141 fprintf('Compute Time: %10d\n',
compute_time);
142

```

Line numbers:

8–12:	Get wall clock time beginning of the execution and open output file for printing energy values.
14–21:	Simulation cell parameters.
16:	Number of grid points in the <i>x</i> -direction.
17:	Number of grid points in the <i>y</i> -direction.
18:	Total number of grid points in the simulation cell.
19:	Grid spacing between two grid points in the <i>x</i> -direction.
20:	Grid spacing between two grid points in the <i>y</i> -direction.
22–29:	Time integration parameters.
24:	Number of time integration steps.
25:	Output frequency to write the results to file.
26:	Time increment in the numerical integration.
27:	Total time.
30–35:	Material-specific parameters.
32:	Average composition.
33:	Mobility value.
34:	Gradient energy coefficient.
36–41:	Initialize the concentration field with a random modulation.
42–45:	Get FFT coefficients.
50–136:	Time evolution of the concentration field.
52:	Update the total time.
54:	Take concentration field from real space to Fourier space (forward FFT transformation).
56–65:	Calculate the derivative of the chemical energy for the grid points in the simulation.

69:	Take the derivatives of the chemical energy from real space to Fourier space (forward FFT transformation).
71–83:	Semi-implicit time integration of concentration field at Fourier space, Eq. 5.14.
87:	Bring back concentration field from Fourier space to real space (inverse FFT transformation).
89–103:	If there are small deviations, set the max and min to the limits.
105–134:	If print frequency is reached, print the results to file.
112–119:	Open an output file and print the results. These lines are commented out but they can be changes, including the output format.
126–130:	Calculate the total bulk energy and print the results to “ <i>time-energy.out</i> ” file for 2D plots.
132:	Write the results to vtk file for contour plots to be viewed by using Paraview.
138–141:	Calculate total execution time and print it to screen.

Program**fft_ch_v2.m**

This program solves Cahn–Hilliard phase-field equation with Fourier spectral method. The time integration is carried out by using semi-implicit time marching scheme. The code is optimized for Matlab/Octave.

The program makes calls to the following functions:

- **prepare_fft.m**
- **free_energ_ch_v2.m**
- **calculate_energ.m**
- **write_vtk_grid_values.**

Listing:

```

1 %%%%%%%%%%%%%%%%
2 % %%%%%%%%%%%%%%%
3 % SEMI-IMPLICIT SPECTRAL %
4 % PHASE-FIELD CODE FOR SOLVING %
5 % CAHN-HILLIARD EQUATION %
6 % (OPTIMIZED FOR MATLAB/OCTAVE) %
7 %%%%%%%%%%%%%%%%

```

```

8                               56   conk = fft2(con) ;
9   %% get intial wall time:      57
10  time0=clock();              58   %--- derivative of free energy:
11  format long;                59
12
13  out2 = fopen('time_energ.out', 60   [dfdcon]=free_energ_ch_v2(Nx,Ny,
14  'w');                      con);
15  %% Simulation cell parameters: 61
16
17  Nx=64;                      62   %--
18  Ny=64;                      63
19  NxNy =Nx*Ny;                64   dfdconk = fft2(dfdcon);
20
21  dx=1.0;                     65
22  dy=1.0;                     66   %--- Time integration:
23
24  %% Time integration parameters: 67
25
26  nstep = 20000;               68   numer = dtime*mobility*k2.
27  nprint= 100;                 *dfdconk;
28  dtime = 1.0e-2;              69
29  ttime = 0.0;                 70   denom = 1.0 + dtime*coefA*mobility*
30  coefA = 1.0;                 grad_coef*k4;
31
32  %% Material specific Parameters: 71
33
34  c0 =0.40;                   72   conk=(conk - numer)./denom;
35  mobility =1.0;              73
36  grad_coef=0.5;              74   con = real(ifft2(conk));
37
38  %% Prepare microstructure    75
39
40  iflag = 1;                  76   %% for small deviations:
41
42  [con] = micro_ch_pre(Nx,Ny,c0, 77
43  iflag);                     78   inrange =(con >= 0.9999);
44  %% Prepare fft:             79   con(inrange) = 0.9999;
45
46  [kx,ky,k2,k4] = prepare_fft(Nx,Ny, 80   inrange =(con < 0.00001);
47  dx,dy);                     81   con(inrange) = 0.00001;
48
49  %% Evolve:                  82
50  %%                         83   ==
51
52  for istep=1:nstep           84   %% print results:
53
54  ttime = ttime + dtime;       85   ==
55
56   conk = fft2(con) ;
57
58   %--- derivative of free energy:
59
60   [dfdcon]=free_energ_ch_v2(Nx,Ny,
61   con);
62   %--
63
64   dfdconk = fft2(dfdcon);
65
66   %--- Time integration:
67
68   numer = dtime*mobility*k2.
69   *dfdconk;
70
71   denom = 1.0 + dtime*coefA*mobility*
72   grad_coef*k4;
73
74   conk=(conk - numer)./denom;
75
76   %% for small deviations:
77
78   inrange =(con >= 0.9999);
79   con(inrange) = 0.9999;
80
81   inrange =(con < 0.00001);
82   con(inrange) = 0.00001;
83
84   %% print results:
85   ==
86
87   if((mod(istep,nprint) == 0) ||
88   (istep == 1) )
89
90   fprintf('done step: %5d\n',istep);
91
92   fname1=sprintf('time_%d.out',
93   istep);
94
95   %for i=1:Nx
96   %for j=1:Ny
97   %fprintf(out1,'%5d %5d %14.6e\n',
98   i,j,con(i,j));
99
100  %end
101  %end
102
103  %fclose(out1);

```

```

101
102 %--- write vtk file:
103
104 %-- calculate total energy
105
106 [energ] = calculate_energ(Nx,Ny,
    con,grad_coef);
107
108 fprintf(out2,'%14.6e %14.6e\n',
    ttime,energ);
109
110 write_vtk_grid_values(Nx,Ny,
    dx,dy,istep,con);
111
112 end %end if
113
114 end %istep
115
116 %--- calculate compute time:
117
118 compute_time = etime(clock(),
    time0);
119 fprintf('Compute Time: %10d\n',
    compute_time);

```

Line numbers:

9–13:	Get wall clock time beginning of the execution and open output file for printing energy values.
15–23:	Simulation cell parameters.
17:	Number of grid points in the <i>x</i> -direction.
18:	Number of grid points in the <i>y</i> -direction.
19:	Total number of grid points in the simulation cell.
21:	Grid spacing between two grid points in the <i>x</i> -direction.
22:	Grid spacing between two grid points in the <i>y</i> -direction.
24–31:	Time integration parameters.
26:	Number of time integration steps.
27:	Output frequency to write the results to file.
28:	Time increment for numerical integration.
29:	Total time.
32–37:	Material-specific parameters.
34:	Average composition.
35:	Mobility value.
36:	Gradient energy coefficient.
38–43:	Initialize the concentration field with a random modulation.
44–47:	Get FFT coefficients.

52–114:	Time evolution of concentration field.
54:	Update total time.
56:	Take concentration field from real space to Fourier space (forward FFT transformation).
58–61:	Derivatives of chemical energy at all grid points in the simulation cell.
64:	Take derivatives of chemical energy from real space to Fourier space (forward FFT transformation).
66–73:	Semi-implicit time integration of concentration field at Fourier space, Eq. 5.14.
74:	Bring back concentration field from Fourier space to real space (inverse FFT transformation).
76–82:	If there are small deviations, set the max and min to the limits.
84–112:	If print frequency is reached, print the results to file.
91–100:	Open an output file and print the results. These lines are commented out but they can be changes, including the output format.
104–108:	Calculate the total bulk energy and print the results to “ <i>time-energy.out</i> ” file for 2D plots.
110:	Write the results to vtk file for contour plots to be viewed by using Paraview.
116–119:	Calculate total execution time and print it to screen.

Function**prepare_fft.m**

This function prepares the Fourier transform coefficients. It is used throughout in this chapter.

Variable and array list:

Nx:	Number of grid points in the <i>x</i> -direction.
<td>Number of grid points in the <i>y</i>-direction.</td>	Number of grid points in the <i>y</i> -direction.
dx:	Grid spacing between two grid points in the <i>x</i> -direction.
dy:	Grid spacing between two grid points in the <i>y</i> -direction.
kx(Nx, Ny):	Directional derivative in the <i>x</i> -direction in Fourier space.
ky(Nx, Ny):	Directional derivative in the <i>y</i> -direction in Fourier Space.
k2(Nx, Ny):	Laplacian in Fourier space.
k4(Nx, Ny):	Square of Laplacian in Fourier space.

Listing:

```

1  function [kx,ky,k2,k4] = prepare_
   fft(Nx,Ny,dx,dy)
2
3  format long;
4
5  Nx21 = Nx/2 + 1;
6  Ny21 = Ny/2 + 1;
7
8  Nx2=Nx+2;
9  Ny2=Ny+2;
10
11 %--
12
13 delkx=(2.0*pi)/(Nx*dx);
14 delky=(2.0*pi)/(Ny*dy);
15
16 %--
17
18 for i=1:Nx21
19 fk1=(i-1)*delkx;
20 kx(i)=fk1;
21 kx(Nx2-i)=-fk1;
22 end
23
24 for j=1:Ny21
25 fk2=(j-1)*delky;
26 ky(j)=fk2;
27 ky(Ny2-j)=-fk2;
28 end
29
30 %--
31
32 for i=1:Nx
33 for j=1:Ny
34
35 k2(i,j)=kx(i)^2+ky(j)^2;
36
37 end
38 end
39
40 %%--
41
42 k4 = k2.^2;
43
44 end %endfunction

```

Line numbers:

5–10:	Set grid parameters for the <i>fft</i> frequencies.
13–14:	Calculate the real space between the frequencies.

18–22:	First-order derivatives in the <i>x</i> -direction.
24–28:	First-order derivatives in the <i>y</i> -direction.
32–39:	Calculate the Laplacian.
42:	Square of Laplacian at all grid points.

References

- Chen LQ, Shen J (1998) Applications of semi-implicit Fourier-spectral method to phase-field equations. *Comput Phys Commun* 108:147

5.5 Case Study-VII**Phase-field modeling of grain growth with semi-implicit Fourier spectral algorithm****Objectives:**

The objective of this case study is to demonstrate a numerical implementation of the semi-implicit Fourier spectral algorithm for the solution of multicomponent non-conserved Allen–Cahn equations in phase-field modeling.

5.5.1 Background

The background information regarding this case study has already been described in *Case Study-II* in which the multicomponent non-conserved Allen–Cahn equation was solved by utilizing finite difference algorithm with explicit Euler time marching scheme. For convenience, the phase-field model is briefly summarized in here again.

5.5.2 Phase-Field Model

Again, in the model each grain is described by one order parameter η_i which takes the value of one for a designated grain and the value of zero in other grains. Recall Eq. 4.31, the evolution of the order parameters described by the non-conserved Allen–Cahn equation in the form of:

$$\frac{\partial \eta_i}{\partial t} = -L_i \frac{\delta F}{\delta \eta_i}, \quad i = 1, 2, \dots, N \quad (5.15)$$

where L_i is the mobility coefficient and F is the free energy functional which is given by Eq. 4.32 as:

$$F = \int_V \left[f(\eta_1, \eta_2, \dots, \eta_N) + \sum_i^N \frac{\kappa_i}{2} |\nabla \eta_i|^2 \right] dv \quad (5.16)$$

in which κ_i are the gradient energy coefficients and f is the local free energy density.

The specific form of local free energy which is independent of orientation described by Eq. 4.33 as:

$$f(\eta_1, \eta_2, \dots, \eta_N) = \sum_i^N \left(-\frac{A}{2} \eta_i^2 + \frac{B}{4} \eta_i^4 \right) + \sum_i^N \sum_{i \neq j}^N \eta_i^2 \eta_j^2 \quad (5.17)$$

5.5.3 Numerical Implementation

The semi-implicit spectral algorithm implementation of Eq. 5.15 given below follows the similar lines as given in *Case Study-VI*.

By taking the functional derivative in Eq. 5.15, it becomes:

$$\frac{\partial \eta_i}{\partial t} = -L \frac{\delta f}{\delta \eta_i} + L \kappa \nabla^2 \eta_i \quad (5.18)$$

By taking Fourier transform of both side of Eq. 5.18 the spatial discretization becomes:

$$\frac{\partial \{\eta_i\}_k}{\partial t} = -L \left\{ \frac{\delta f}{\delta \eta_i} \right\}_k - k^2 L \kappa \{\eta_i\}_k \quad (5.19)$$

where $\{\cdot\}_k$ is the Fourier transform of the quantity inside the bracket and k is the vector in Fourier space, $k = (k_1, k_2)$, with a magnitude $\sqrt{k_1^2 + k_2^2}$. By treating linear and second-order operators implicitly and the other terms explicitly, the semi-implicit form of Eq. 5.19 is:

$$\frac{\{\eta_i\}_k^{n+1} - \{\eta_i\}_k^n}{\Delta t} = -L \left\{ \frac{\delta f}{\delta \eta_i} \right\}_k^n - k^2 L \kappa \{\eta_i\}_k^{n+1} \quad (5.20)$$

where Δt is the time increment between time steps $n+1$ and n . By rearranging it becomes

$$\{\eta_i\}_k^{n+1} = \frac{\{\eta_i\}_k^n - \Delta t L \left\{ \frac{\delta f}{\delta \eta_i} \right\}_k^n}{1 + \Delta t L k^2} \quad (5.21)$$

and the problem reduces to solving the discretized Eq. 5.21 for each grain in the simulation.

The numerical steps to achieve this can be summarized as:

For number of steps repeat:

For each grain (i.e., order parameters η_i)

1. Evaluate $(\delta f / \delta \eta_i)$ using the current time step value of η_i . Fourier transform $(\delta f / \delta \eta_i)$ and η_i .
2. Calculate the spatial variation of η_i in Fourier space for the time, $t + \Delta t$, using Eq. 5.21.
3. With inverse Fourier transformation bring back the results to real space for the solution at $t + \Delta t$.

5.5.4 Results and Discussion

The simulation was carried out for a square simulation cell having $N_x = 64$ and $N_y = 64$ and with $dx = dy = 0.5$. All the parameters used in these simulations were identical to that used for *Case Study-II*.

Again, the first set simulations were carried out for the shrinking spherical grain as ideal grain growth. Figure 5.4 shows the time evolution of the spherical grain with initial radius of $D_0 = 14dx$ during the course of simulation. With increasing time, the uniform shrinkage of the spherical grain is apparent from the figure. The variation of the area fraction of the spherical grain with time is compared with the results obtained in *Case Study-II* with explicit Euler finite difference algorithm in Fig. 5.5.

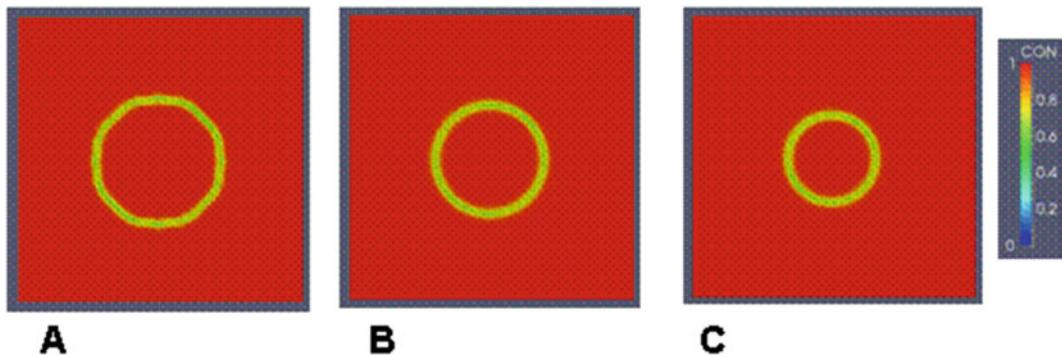
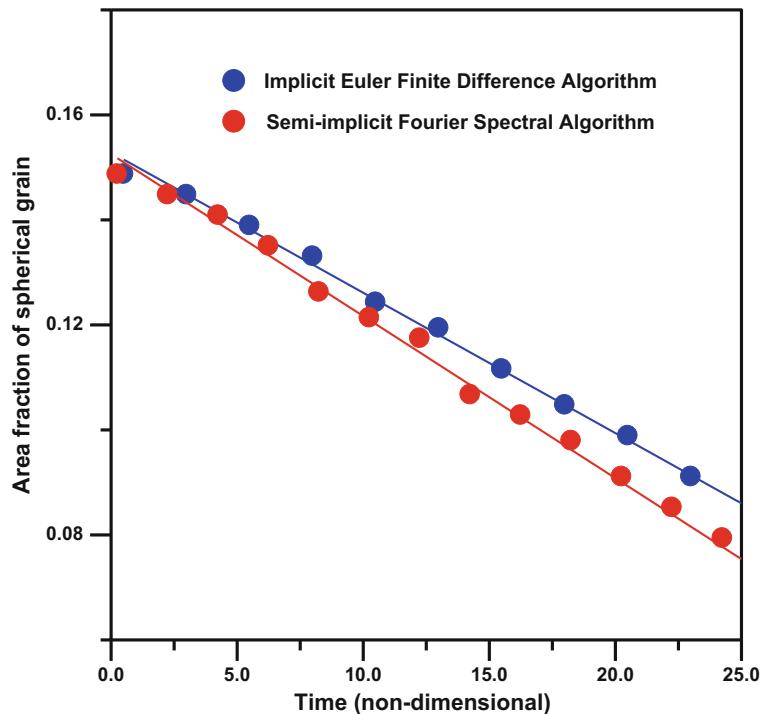


Fig. 5.4 Time evolution of a spherical grain. Nondimensional times are: (a) 0.5, (b) 12.5, and (c) 25

Fig. 5.5 Comparison of the results obtained from semi-implicit Fourier spectral method and explicit Euler finite difference algorithm



In the next simulation, the grain growth is considered, again by using the same input file *grain_25.inp*. The time evolution of the polycrystalline microstructure composed of 25 grains is shown in Fig. 5.6. Similar to that observed in *Case Study-II*, the evolution proceeds as the growth of the larger grains and the disappearance of the smaller ones.

The variation of area fractions of those first five grains in the input list with time are compared in Fig. 5.7. Although, there are slight

variations, however, both algorithms lead to similar evolution kinetics with the identical simulation parameters. In general, the semi-implicit Fourier spectral method enables to use smaller number of grid points to resolve the spatial discretization owing to the exponential convergence of the Fourier spectral method. In addition, it relaxes the very strict time step size of explicit Euler time marching scheme and permits much larger time step sizes in the solution [1].

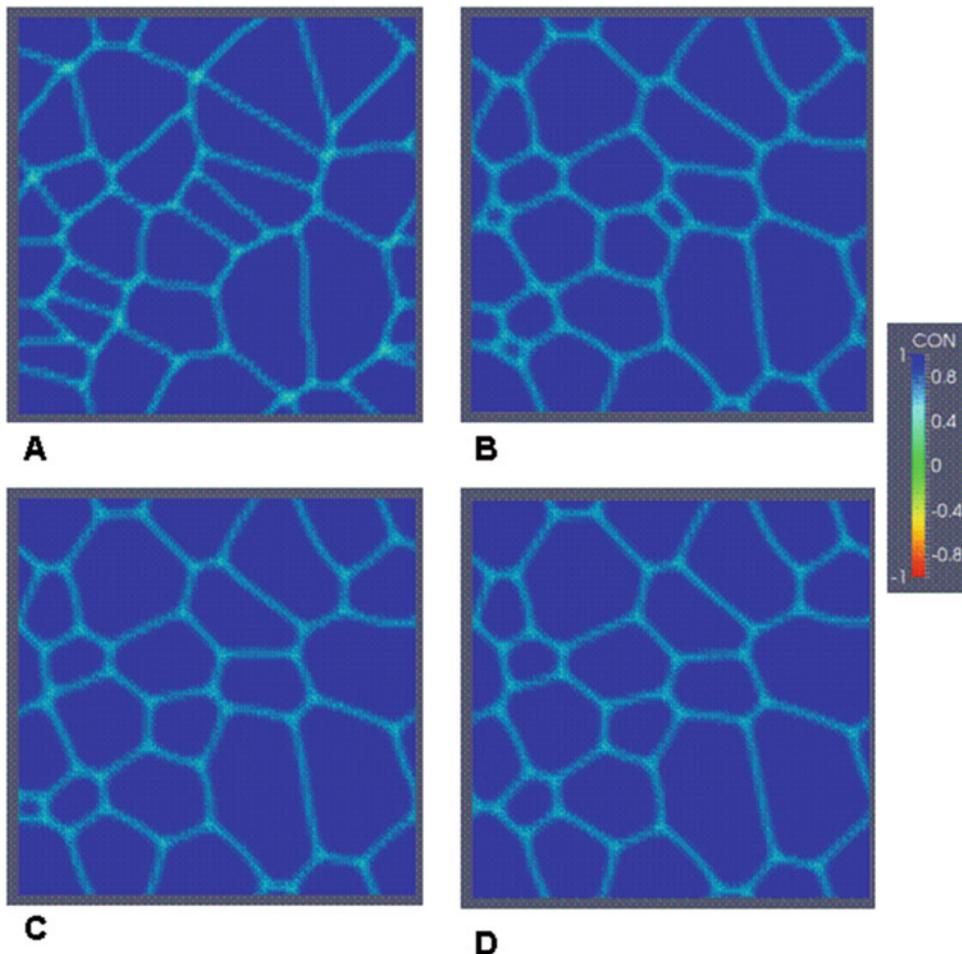


Fig. 5.6 Time evolution of polycrystalline microstructure due to grain growth. Nondimensional times are (a) 0.005, (b) 12.5, (c) 18.75, and (d) 25

The animations resulting from these simulations are given in subdirectory *case_fft_ca_v1.m* in downloadable file.

5.5.5 Source Codes

Two source codes, *fft_ca_v1.m* in longhand format and corresponding Matlab/Octave optimized version *fft_ca_v2.m* with the associated functions, are given below.

Program *fft_ca_v1.m*

This program solves the phase-field equations based on the non-conserved multicomponent Allen–Cahn equation with Fourier spectral method. The time integration is carried out with semi-implicit time marching scheme. It is not optimized and in longhand format.

The program makes calls to the following functions:

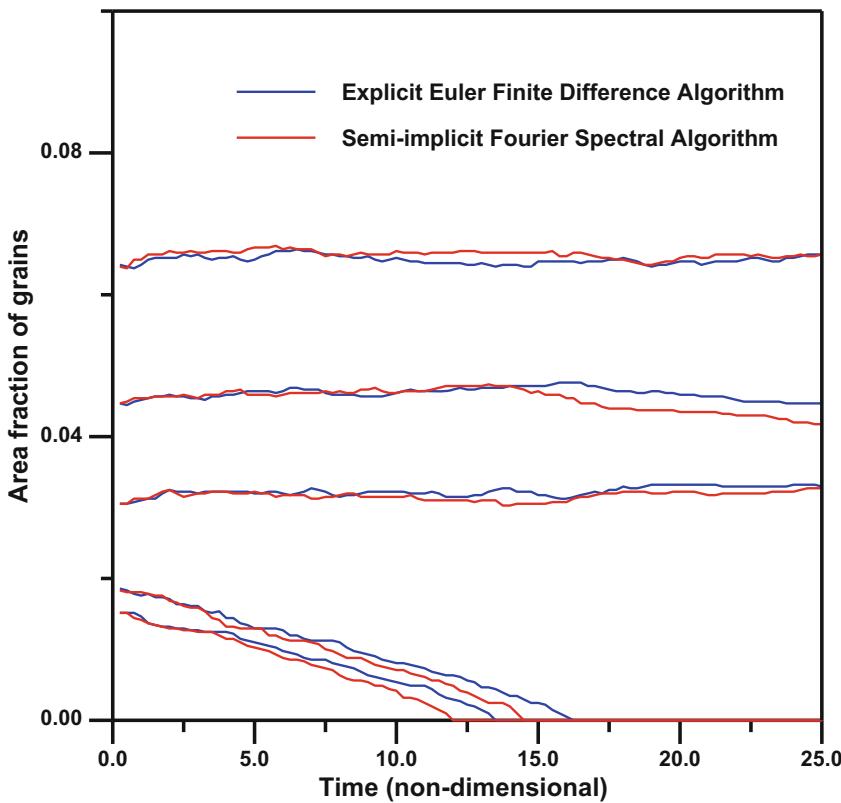


Fig. 5.7 Comparison of the evolution kinetics of five grains in polycrystalline simulations obtained with the semi-implicit Fourier spectral and explicit Euler finite difference algorithms

- `init_grain_micro.m`
- `free_energ_fft_ca_v1.m`
- `prepare_fft.m`
- `write_vtk_grid_values.m`

Listing:

```

1  %%%%%%
2  %
3  % SEMI-IMPLICIT SPECTRAL %
4  % PHASE-FIELD CODE %
5  % FOR SOLVING ALLEN-CAHN EQUATION %
6  %%%%
7  %
8  %% get initial wall time
9  %
10 time0=clock();
11 format long;
12
13 out2=fopen('area_frac.out','w');
14
15 %-- Simulation cell parameters:
16
17 Nx = 64;
18 Ny = 64;
19 NxNy=Nx*Ny;
20 dx = 0.5;
21 dy = 0.5;
22
23 %--- Time integration parameters:
24
25 nstep= 5000;
26 nprint= 50;
27 dtim= 0.005;
28 ttime= 0.0;
29 coefA= 1.0;
30
31 %--- Material Parameters
32
33 mobil= 5.0;

```

```

34 grcoef = 0.1;
35
36 %
37 %--- Generate initial grain_structure
38 %
39 iflag = 1;
40 isolve = 1;
41
42 [etas,ngrain,glist] = init_grain_
micro(Nx,Ny,dx,dy,iflag,isolve);
43
44 %
45 %--- Prepare fft
46 %
47
48 [kx,ky,k2,k4] = prepare_fft(Nx,Ny,
dx,dy);
49
50 %
51 %--- Evolve:
52 %
53
54 for istep=1:nstep
55
56 ttime=ttime+dtime;
57
58 for igrain=1:ngrain
59
60 if(glist(igrain) == 1)
61
62 for i=1:Nx
63 for j=1:Ny
64
65 eta(i,j) = etas(i,j,igrain);
66
67 %
68 %--Derivative of free energy
69 %
70
71 [dummy] = free_energ_fft_ca_v1(i,j,
ngrain,etas,eta,igrain);
72
73 dfdeta(i,j) = dummy;
74
75 end%j
76 end%i
77
78 %%
79
80 etak = fft2(eta);
81 dfdeta_k = fft2(dfdeta);
82
83 %%
84 %--- Time integration:
85 %
86
87 for i=1:Nx
88 for j=1:Ny
89
90 numer = dtime*mobil*dfdeta_k(i,j);
91
92 denom = 1.0 + dtime*coefA*mobil*
grcoef*k2(i,j);
93
94 etak(i,j) = (etak(i,j) - numer) /
denom;
95
96 end%j
97 end%i
98
99 %%
100
101 eta = real(ifft2(etak));
102
103 %%
104 %--
105
106 grain_sum = 0.0;
107 for i=1:Nx
108 for j=1:Ny
109
110 %-- for small deviations:
111
112 if(eta(i,j) >= 0.9999);
113 eta(i,j) = 0.9999;
114 end
115
116 if(eta(i,j) < 0.00001);
117 eta(i,j) = 0.00001;
118 end
119
120 grain_sum = grain_sum + eta(i,j);
121
122 etas(i,j,igrain) = eta(i,j);
123
124 end%j
125 end%i
126
127 %-- Check volume fraction of current
grain:

```

```

128
129 grain_sum=grain_sum/NxNy;
130 if(grain_sum<= 0.001)
131 glist(igrain)=0;
132 fprintf('grain: %5dis eliminated\n',
133 igrain);
134 end
135 %---
136 end%if
137 end%igrain
138
139 if(mod(istep,nprint) == 0)
140
141 fprintf('done step: %5d\n',istep);
142
143 %fname1=sprintf('time_%d.out',
144 istep);
145 %out1=fopen(fname1,'w');
146 %for i=1:Nx
147 %for j=1:Ny
148
149 %gg=0.0;
150 %for igrain=1:ngrain
151 %gg=gg+etas(i,j,igrain)^2;
152 %end
153
154 %fprintf(out1,'%5d %5d %14.6e\n',
155 %i,j, gg);
156 %end
157
158 %fclose(out1);
159
160 %--- write vtk file & calculate area
161 % fraction of grains:
162 eta2=zeros(Nx,Ny);
163
164 fprintf(out2,'%14.6e',ttime);
165
166 for igrain=1:ngrain
167 ncount=0;
168 for i=1:Nx
169 for j=1:Ny
170
171 eta2(i,j) = eta2(i,j) + etas(i,j,
172 igrain)^2;
173 if(etas(i,j,igrain) >= 0.5)
174 ncount=ncount+1;
175 end
176
177 end %j
178 end %i
179 ncount=ncount/(NxNy);
180 fprintf(out2,'%14.6e',ncount);
181 end%igrain
182 fprintf(out2,'\n');
183
184 %--
185
186 write_vtk_grid_values(Nx,Ny,dx,dy,
187 istep,eta2);
188 end%if
189
190 end%istep
191
192 %--- calculate compute time:
193
194 compute_time = etime(clock(),
195 time0);
196 fprintf('Compute Time: %10d\n',compu
te_time);

```

Line numbers:

8–14:	Get wall clock time beginning of the execution and open output file for printing area fraction of each grain.
15–22:	Simulation cell parameters.
17:	Number of grid points in the <i>x</i> -direction.
18:	Number of grid points in the <i>y</i> -direction.
19:	Total number of grid points in the simulation cell.
20:	Grid point spacing between two grid points in the <i>x</i> -direction.
21:	Grid point spacing between two grid points in the <i>y</i> -direction.
23–30:	Time integration parameters.
25:	Number of time integration steps.
26:	Output frequency to write results to file.
27:	Time increment for numerical integration.
28:	Total time.
31–35:	Material parameters.
33:	Mobility coefficient.
34:	Gradient energy coefficient.
37–43:	Generate initial grain microstructure.
39:	iflag = 1 for bicrystal simulation, iflag = 2 for polycrystal simulation.

(continued)

40:	isolve = 1, since the solution uses double arrays as in program <i>fd_ca_v1.m</i> .
45–48:	Get FFT coefficients.
51–190:	Time integration of order parameters.
56:	Update total time.
58–137:	Loop over each grain.
60:	If glist is equal to one, which indicates that the current grain area fraction is greater than 0.001, continue the calculation. Otherwise, the current grain does not exist anymore.
62–77:	Assign order parameters values for the current grain to temporary array eta(Nx,Ny) from common array etas(Nx,Ny,ngrain).
68–74:	Calculate the derivative of free energy for the current grid point and accumulate into array, dfdeta(Nx,Ny).
80:	Take the order parameter values from real space to Fourier space (forward FFT transformation).
81:	Take the derivatives of the order parameter from real space to Fourier space (forward FFT transformation).
84–97:	Semi-implicit time integration of Eq. 5.21 at Fourier space.
101:	Bring back the order parameter from Fourier space to real space (inverse FFT transformation).
106–125:	Calculate area of current grain.
110–118:	If there are small deviations, set the max and min values to the limits.
120:	Calculate area of current grain.
122:	Put back order parameter values of current grain from temporary array eta(Nx,Ny) to common array etas(Nx,Ny,ngrain).
127–134:	Check the area fraction of the current grain. If it is less then 0.001, set its value in glist (ngrain) as zero which indicates that it is extinct. Also, print message “grain # is eliminated” to screen.
139–188:	If print frequency is reached, print the results to file.
143–158:	Open an output file and print the results. These lines are commented out but they can be changed, including the output format.
160–188:	Prepare the data to be written to vtk file and calculate the area fraction of each grain and print them to file <i>area_frac.out</i> .
186:	Write the results to vtk file for contour plots to be viewed by using Paraview.
194–195:	Calculate the total execution time and print it to screen.

Program

fft_ca_v2.m

This program solves the phase-field equations based on the non-conserved multicomponent Allen–Cahn equation with Fourier spectral method. The time integration is carried out with semi-implicit scheme. It is optimized for Matlab/Octave.

The program makes calls to the following functions:

- **init_grain_micro.m**
- **free_energ_fft_ca_v2.m**
- **prepare_fft.m**
- **write_vtk_grid_values.m**

Listing:

```

1 %%%%%%%%%%%%%%%%
2 %
3 % SEMI-IMPLICIT SPECTRAL %
4 % PHASE-FIELDCODE %
5 % FOR SOLVING ALLEN-CAHN EQUATION %
6 % (OPTIMIZED FOR MATLAB/OCTAVE) %
7 %%%%%%%%%%%%%%%%
8
9 %== get initial wall time
10
11 time0=clock();
12 format long;
13
14 out2=fopen('area_frac.out','w');
15
16 %%-- Simulation cell parameters:
17
18 Nx = 64;
19 Ny = 64;
20 NxNy= Nx*Ny;
21 dx = 0.5;
22 dy = 0.5;
23
24 %%-- Time integration parameters:
25
26 nstep = 5000;

```

```

27 nprint= 50;                      73
28 dtme = 0.005;                   74 dfdeta =free_energ_fft_ca_v2(Nx,
29 ttime = 0.0;                   75 Ny,ngrain,etas,eta,igrain);
30 coefA = 1.0;                   76 dfdetak = fft2(dfdeta);
31 %--- Material Parameters       77
32                               78 %
33 mobil = 5.0;                  79 %--- Time integration:
34 grcoef = 0.1;                 80 %
35                               81
36 %                               82 numer = dtme*mobil*dfdetak;
37 %--- Generate initial grain_   83
38 structure                     84 denom = 1.0 + dtme*coefA*mobil*
39 iflag = 2;                     85 grcoef*k2;
40 isolve = 1;                   86 etak = (etak - numer) ./denom ;
41                               87
42 [etas,ngrain,glist] = init_grain_
micro(Nx,Ny,dx,dy,iflag,isolve); 88 %--
43                               89
44 %                               90 eta = real(ifft2(etak));
45 %--- Prepare fft              91
46 %                               92 %-- for small deviations:
47                               93
48 [kx,ky,k2,k4] = prepare_fft(Nx, 94 inrange =(eta >= 0.9999);
Ny,dx,dy);                    95 eta(inrange) = 0.9999;
49 %                               96 inrange =(eta < 0.00001);
50 %                               97 eta(inrange) = 0.00001;
51 %--- Evolve:                 98
52 %                               99 %--
53                               100
54 eta = zeros(Nx,Ny);          101 etas( :, :, igrain) =eta;
55                               102
56 for istep=1:nstep           103 %-- Check volume fraction of current
57 ttime = ttime+dtme;          104 grain:
58 for igrain=1:ngrain          105 grain_sum = 0.0;
59 if(glist(igrain) == 1)        106 grain_sum = sum(sum(eta))/NxNy;
60 eta = etas( :, :, igrain);   107
61                               108 if(grain_sum <= 0.001)
62 if(glist(igrain) == 1)        109 glist(igrain) =0;
63 eta = etas( :, :, igrain);   110 fprintf('grain: %5d is
64                               111 end
65 %                               112 %---
66                               113
67                               114 end %if
68 etak = fft2(eta);            115 end %igrain
69 %                               116
70 %--Derivative of free energy 117 if(mod(istep,nprint) == 0)
71 %                               118
72 %

```

```

118
119 fprintf('done step: %5d\n',istep);
120
121 %fname1=sprintf('time_%d.out',
122 istep);
123
124 %for i=1:Nx
125 %for j=1:Ny
126
127 % gg= 0.0;
128 %for igrain =1:ngrain
129 %gg = gg +etas(i,j,igrain)^2;
130 %end
131
132 %fprintf(out1,'%5d %5d %14.6e\n',
133 i,j,gg);
134 %end
135
136 %fclose(out1);
137
138 %--- write vtk file & calculate area
139 fraction of grains:
140 eta2=zeros(Nx,Ny);
141
142 fprintf(out2,'%14.6e',ttime);
143
144 for igrain=1:ngrain
145 ncount=0;
146 for i=1:Nx
147 for j=1:Ny
148
149 eta2(i,j) = eta2(i,j) + etas(i,j,
150 igrain)^2;
151 if(etas(i,j,igrain) >= 0.5)
152 ncount=ncount+1;
153 end
154
155 end
156 end
157 ncount=ncount/(NxNy);
158 fprintf(out2,'%14.6e',ncount);
159 end
160 fprintf(out2,'\n');
161
162 %--
163
164 write_vtk_grid_values(Nx,Ny,
165 dx,dy,istep,eta2);
166 end %end if
167
168 end %istep
169
170 %--- calculate compute time:
171
172 compute_time = etime(clock(),
173 time0);
174 fprintf('Compute Time: %10d\n',
175 compute_time);

```

Line numbers:

9–14:	Get wall clock time beginning of the execution and open output file for printing area fraction of each grain.
16–23:	Simulation cell parameters.
18:	Number of grid points in the <i>x</i> -direction.
19:	Number of grid points in the <i>y</i> -direction.
20:	The total number of grid points in the simulation cell.
21:	The grid spacing between two grid points in the <i>x</i> -direction.
22:	The grid spacing between two grid points in the <i>y</i> -direction.
24–30:	Time integration parameters.
26:	Number of time integration steps.
27:	Output frequency to write the results to file.
28:	Time increment for numerical integration.
29:	Total time.
31–35:	Material parameters.
33:	Mobility coefficient.
34:	Gradient energy coefficient.
37–43:	Generate the initial microstructure.
39:	iflag = 1 for bicrystal simulation and iflag = 2 for polycrystal simulation.
40:	isolve = 1, since the solution uses double arrays as in program <i>fd_ca_yl.m</i> .
45–48:	Get FFT frequency coefficients.
51–168:	Time evolution.
58:	Update total time.
60–115:	Loop over each grain and evolve each grain one at a time for this time step.
62:	Check current grain is still exists. If it does continue the calculation.
64:	Copy current grain order parameters from common array etas(Nx,Ny,ngrain) to temporary array eta(Nx,Ny).
68:	Take the eta values from real space to Fourier space (forward FFT transformation).

(continued)

71–74:	Calculate the derivative of free energy for this grain at all grid points.
76:	Take the derivative values from real space to Fourier space (forward FFT transformation).
79–86:	Semi-implicit time integration of Eq. 5.21 at all grid points.
90:	Bring back the order parameter values from Fourier space to real space (inverse FFT transformation).
92–97:	If there are small deviations, set the max and min values to the limits.
101:	Put back the order parameter values from temporary array eta(Nx,Ny) to common array etas(Nx,Ny,ngrain).
103–112:	Calculate the area fraction of the current grain. If area fraction equal or less than 0.001, set the flag to zero for this grain in array glist and this grain will not be considered any further. Also print message “grain # is eliminated” to screen.
117–166:	If print frequency is reached, print the results to file.
121–136:	Open an output file and print the results. These lines are commented out but they can be changed, including the output format.
138–165:	Prepare the data to be written to vtk file and calculate the area fraction of each grain and print them to file <i>area_fract.out</i> .
164:	Write the results to vtk file for contour plots to be viewed by using Paraview.
170–173:	Calculate the total execution time and print it to screen.

```

1 function [dfdeta] = free_energ_
2 fft_ca_v1(i,j,ngrain,etas,eta,
3 igrain)
4
5 format long;
6 A=1.0;
7 B=1.0;
8 sum=0.0;
9
10 for jgrain=1:ngrain;
11
12 if(jgrain ~= igrain)
13 sum = sum +etas(i,j,jgrain).^2;
14 end
15
16 end
17
18 dfdeta = A*(2.0*B* eta(i,j)*sum +
19 eta(i,j)^3 - eta(i,j));
20 end %endfunction

```

Line numbers:

5–6:	Parameters in free energy function in Eq. 5.17.
8–16:	Carry out the summation in Eq. 5.17.
18:	Calculate the derivative of the free energy for the current grain under consideration in the main program at the current grid point.

Function

free_energ_fft_ca_v1.m

This function evaluates the derivatives of the free energy, for the grain under consideration, in the main program at current grid point.

Variable and array list:

i:	Index of current grid point in the <i>x</i> -direction.
j:	Index of current grid point in the <i>y</i> -direction.
ngrain:	Number of grains in the solution.
igrain:	Current grain number under consideration in the main program.
dfdeta:	The derivative of the free energy for the current grid point.
etas(Nx, Ny, ngrain):	Common array for order parameters of the grains.
eta(Nx, Ny):	Order parameters of current grain under consideration in the main program.

Variable and array list:

Nx:	Number of grid points in the <i>x</i> -direction.
<b b="" ny:<="">	Number of grid points in the <i>y</i> -direction.
ngrain:	Number of grains in the solution.
igrain:	Current grain number under consideration in the main program.
dfdeta(Nx, Ny):	The derivative of the free energy at all grid points in the simulation cell.
etas(Nx,Ny, ngrain):	Common array for order parameters of the grains.
eta(Nx,Ny):	Order parameters of current grain under consideration in the main program.

Listing:

```

1 function [dfdeta] = free_energ_fft_
2   ca_v2 (Nx,Ny,ngrain,etas,eta,
3   igrain)
4
5 A=1.0;
6 B=1.0;
7
8 sum=zeros(Nx,Ny);
9
10 for jgrain=1:ngrain;
11
12 if(jgrain == igrain)
13 sum = sum +etas(:, :,jgrain) .^2;
14 end
15
16 end
17
18 dfdeta = A*(2.0*B* eta .*sum +eta.^3
- eta);
19
20 end %endfunction

```

Line numbers:

5–6:	Constants in free energy, Eq. 5.17.
8–16:	Calculate sums in free energy, Eq. 5.17
18:	Calculate the derivative of the free energy for the current grain in the main program at all grid points in the simulation.

References

- Chen LQ, Shen J (1998) Applications of semi-implicit Fourier-spectral method to phase-field equations. Comput Phys Commun 108:147

5.6 Case Study-VIII**Phase-field simulation of precipitation behavior of a Fe–Cu–Mn–Ni alloy****Objectives:**

The objective of this case study is to develop a unified phase-field simulation framework for systems in which phase transformation as well as phase separations simultaneously are taking place during the course of microstructure evolution in solid state. The non-conserved Allen–

Cahn formulism, for the phase transformation and Cahn–Hilliard formulism, for the conserved alloy compositions during the phase separation will be utilized in the model. In addition, the thermodynamic database, Thermo-Calc, for a Fe–Cu–Ni–Mn alloy will be used, as an example of how the multicomponent phase-field modeling can be applied to real complex alloying systems.

5.6.1 Background

In addition to being a technologically significant alloying system, Fe–Cu alloys are often utilized as a model system in experimental studies, owing to almost no solid solubility in each other. There are many studies on the precipitation sequence of Cu-rich phases in Fe–Cu alloy systems [1–3]. It is commonly accepted that the initial precipitation starts as a metastable body-centered cubic (*bcc*) phase which is coherent with the Fe matrix [4–6]. During subsequent growth and coarsening, the precipitates undergo through faulted structures of hexagonally ordered phases (9R and 3R), and finally become the equilibrium face-centered cubic (*fcc*) structure of Cu [2]. It is shown that [7, 8] the presence of other alloying elements, such as Ni and Mn, leads to the formation of a ring/core precipitate morphology with Ni–Mn-rich intermetallic B2 rings and a Cu-rich core. The presence of this thermally stable intermetallic phase prevents further diffusional growth, leading to the sluggish coarsening of the Cu-rich precipitates later. In addition, the intermetallic B2 shells act as a buffer layer to relax the lattice mismatch strain between the Cu precipitates and the Fe matrix. Since Fe–Cu alloys are extensively used in industry, several modeling and simulation approaches have been used to quantify precipitation kinetics as overviewed in [9]. At the atomistic level, one class of models relies on cluster dynamics as in [10, 11]. The other models are based on kinetic Monte Carlo algorithm [12, 13]. In general, these atomistic based models and simulation algorithms are limited in spatial and temporal scales. At the mesoscale, the CALPHAD (CALculation of PHAse Diagrams) database has been

used for the thermodynamic description of Fe–Cu–Mn–Ni system in order to quantitatively predict the microstructure evolution at spinodal regime within formulism of phase-field modeling [14, 15].

5.6.2 Phase-Field Model

We will utilize the models developed in two previous case studies for the formulism given in [15] for Fe–Cu–Mn–Ni system. The microstructure evolution of a multicomponent alloy system driven by the free energy reduction governed by the Cahn–Hilliard and Allen–Cahn equations:

$$\frac{\partial c_i(r, t)}{\partial t} = \nabla \cdot M_i \cdot \nabla \frac{\partial F}{\partial c_i(r, t)} + \xi_{ci}(r, t) \quad (5.22)$$

$$\frac{\partial \eta(r, t)}{\partial t} = -L_\eta \frac{\partial F}{\partial (\eta, t)} + \xi_\eta(r, t) \quad (5.23)$$

where $c_i(r, t)$ describes the concentration field (as a function of spatial position r and time t) of the alloying elements (i.e., $i = 1, 2, 3, 4$ for

Fe, Cu, Mn, and Ni, respectively). The phase-field variable $\eta(r, t)$ characterizes the phase distribution of $\alpha(bcc)$ and $\gamma(fcc)$ phases in the Cu precipitates and takes the values of $0 < h(\eta) < 1$, corresponding $h(\eta) = 0$ for $\alpha(bcc)$ phase and $h(\eta) = 1$ for $\gamma(fcc)$ phase, respectively. $\xi_{ci}(r, t)$ and $\xi_\eta(r, t)$ are the usual noise terms to take into account the thermal fluctuations. M_i is the mobility of the alloying components and defined as:

$$M_i(\eta, T) = c_{oi}(1 - c_{oi}) \times \left\{ (1 - \eta) \frac{D_i^\alpha(T)}{RT} + \eta \frac{D_i^\gamma(T)}{RT} \right\} \quad (5.24)$$

where R is the gas constant, T is the temperature, and c_{oi} is the nominal composition of the alloying element i . $D_i^\phi(T)$ is the diffusion constant of alloying element i in the $\alpha(bcc)$ and $\gamma(fcc)$ phases. L_η is the kinetic mobility characterizing the evolution of phase transformation between $\alpha(bcc)$ and $\gamma(fcc)$ phases.

The total free energy, F , including both the short-range interaction owing to compositional and phase inhomogeneities and the long-range elastic interaction can be expressed as [15]:

$$F = \int_v \left\{ [1 - h(\eta)](G_c^\alpha(c_i, T) + YV_m \epsilon_0^2(c_i)) + h(\eta)G_c^\gamma(c_i, T) + Wg^2(\eta) + \sum_{i=1}^4 \frac{1}{2} k_c (\nabla c_i)^2 + \frac{1}{2} k_\eta (\nabla \eta)^2 \right\} dv \quad (5.25)$$

where $G_c^\alpha(c_i, T)$ and $G_c^\gamma(c_i, T)$ are the Gibbs energies of α and γ phases, as a function of composition c_i and temperature T , respectively. They are detailed below; they are directly related to the phase diagram of the Fe–Cu–Ni–Mn system. The quantity $YV_m \epsilon_0^2(c_i)$ is the elastic strain energy induced from the coherent phase separation, where Y is the average stiffness and V_m is the molar volume and $\epsilon_0(c_i)$ is the eigenstrain and taken as:

$$\epsilon_{kl}^0(r, t) = \delta_{kl} \sum_{i=2}^4 \epsilon_i^0 (c_i(r, t) - c_i^0) \quad (5.26)$$

where ϵ_i^0 is the constant determined from the lattice misfits and initial compositions of the i -th alloying component, c_i^0 . The term $Wg(\eta)$ in the integrand represents the energy barrier for the phase transformation between the α and γ phases. The functions $h(\eta)$ and $g(\eta)$ are represented as $h(\eta) = \eta^2(3 - 2\eta)$ and $g(\eta) = \eta(1 - \eta)$. The fourth and fifth terms are the

gradient energies of the composition and the phase fields.

The Gibbs energy function of phase φ phase ($\varphi = \alpha$ or γ) in an Fe–Cu–Mn–Ni quaternary system with magnetic contribution can be approximated by the sub-regular solution [15],

$$G_c^\varphi(c_i, T) = \sum_i G_i^\varphi c_i + {}^E G^\varphi + {}^{mg} G^\varphi + RT \sum_i c_i \ln c_i \quad (5.27)$$

where G_i^φ is the Gibbs energy of the φ phase of the pure element i ($i = 1, 2, 3$, and 4 correspond to Fe, Cu, Mn, and Ni) and its concentration c_i , which is expressed as function of temperature, T .

${}^E G^\varphi$ is the excess energy corresponding to the heat of mixing, and ${}^{mg} G^\varphi$ is the magnetic contribution to the Gibbs energy which will be omitted in here for simplicity. The function ${}^E G^\varphi$ is defined as

$${}^E G^\varphi \equiv \sum_i \sum_{j>i} L_{i,j}^\varphi c_i c_j + \sum_i \sum_{j>i} \sum_{k>j} L_{i,j,k}^\varphi c_i c_j c_k \quad (5.28)$$

Following [15], the values for the parameters in Eqs. 5.27 and 5.28 as a function of temperature and the concentration of alloying elements are:

$${}^0 G_1^\alpha = 0$$

$${}^0 G_2^\alpha = 4017 - 1.255T$$

$${}^0 G_3^\alpha = -3235.3 + 127.85T - 23.7T \ln T - 0.0074271T^2 + 60000/T$$

$${}^0 G_4^\alpha = 8715.084 - 3.556T$$

$${}^0 G_1^\gamma = -1462.4 + 8.282T - 1.15T \ln T + 6.4 \times 10^{-4}T^2$$

$${}^0 G_2^\varphi = 0$$

$${}^0 G_3^\gamma = -3439.3 + 131.884T - 24.5177T \ln T - 0.006T^2 + 69600/T$$

$${}^0 G_4^\gamma = 0$$

$$L_{1,2}^\alpha = 41033.0 - 6.022T$$

$$L_{1,3}^\alpha = -2759.0 + 1.237T$$

$$L_{1,4}^\alpha = -956.63 - 1.28726T + (1789.03 - 1.92912T)(c_1 - c_4)$$

$$L_{2,3}^\alpha = 11190.0 - 6.0T - 9865.0(c_2 - c_3)$$

$$L_{2,4}^\alpha = 8366.0 + 2.80T$$

$$L_{3,4}^\alpha = -51638.31 + 3.64T + 6276.0(c_3 - c_4)$$

$$L_{1,2,3}^\alpha = 30000.0, \quad L_{1,2,4}^\varepsilon = L_{1,3,4}^\alpha = L_{2,3,4}^\alpha = 0$$

$$L_{1,2}^\gamma = 53360 - 12.626T + (11512 - 7.095T)(c_2 - c_1)$$

$$L_{1,3}^\gamma = (-7762 + 3.865T) - 259(c_1 - c_3)$$

$$L_{1,4}^\gamma = -12054.355 + 3.27413T + (11082.1315 - 4.45077T)(c_1 - c_4) - 725.8051(c_1 - c_4)^2$$

$$L_{2,3}^\gamma = 11820 - 2.3T + (-10600 + 3T)(c_2 - c_3) + (-4850 + 3.5T)(c_2 - c_3)^3$$

$$L_{2,4}^\gamma = 8366 + 2.802T + (-4359.6 + 1.812T)(c_2 - c_4)$$

$$L_{3,4}^\gamma = -58158 + 10.878T + 6276(c_3 - c_4)$$

$$L_{1,2,3}^\gamma = -68000 + 50T$$

$$L_{1,2,4}^\gamma = -73272 + 30.9T$$

Table 5.1 Values of the parameters used in the phase-field model

Gradient energy coefficients ($\text{J m}^2 \text{mol}^{-1}$)	$K_c = 5.0 \times 10^{-15}, K_\eta = 1.0 \times 10^{-15}$
Molar volume, V_m ($\text{m}^3 \text{mol}^{-1}$)	7.09×10^{-6}
Energy function of elastic stiffness, $n\text{GPa}$)	214.0
Lattice mismatch	$e_{\text{Cu}}^0 = 3.29 \times 10^{-2}$ $e_{\text{Mn}}^0 = 5.22 \times 10^{-4}$ $e_{\text{Ni}}^0 = 4.75 \times 10^{-4}$
Parameter W (J mol^{-1})	5.0×10^3
$D_i^\varphi(T) = {}^0D_i^\varphi \exp(-Q_i^\varphi/RT)$	${}^0D_{\text{Cu}}^\alpha = 4.7 \times 10^{-5}, Q_{\text{Cu}}^\alpha = 2.44 \times 10^5$
${}^0D_i^\varphi(\text{m}^2 \text{second}^{-1})$	${}^0D_{\text{Cu}}^\gamma = 4.3 \times 10^{-5}, Q_{\text{Cu}}^\gamma = 2.80 \times 10^5$
$Q_i^\varphi (\text{J mol}^{-1})$	${}^0D_{\text{Mn}}^\alpha = 1.49 \times 10^{-4}, Q_{\text{Mn}}^\alpha = 2.33 \times 10^5$
$\varphi = \alpha, \gamma$	${}^0D_{\text{Mn}}^\gamma = 1.78 \times 10^{-5}, Q_{\text{Mn}}^\gamma = 2.64 \times 10^5$
	${}^0D_{\text{Ni}}^\alpha = 1.40 \times 10^{-4}, Q_{\text{Ni}}^\alpha = 2.46 \times 10^5$
	${}^0D_{\text{Ni}}^\gamma = 1.08 \times 10^{-5}, Q_{\text{Ni}}^\gamma = 2.73 \times 10^5$

5.6.3 Numerical Implementation

For the solution of the conserved alloy concentrations, Eq. 5.22, we will follow the steps given by Eqs. 5.10–5.14 in *Case Study-VI* and for the non-conserved phase transformations, Eq. 5.23, we utilize Eqs. 5.18–5.21 in *Case Study-VII*. The quantities having the dimension of energy were normalized with RT , and the time t was normalized with $dx^2/D_{\text{Cu}}^\alpha(T)$, where D_{Cu}^α is the diffusion constant of Cu in the α phase at the temperature T . The parameters used in the simulation are tabulated in Table 5.1.

5.6.4 Results and Discussion

The simulations were carried out in a simulation cell with the number of grid points $N_x = N_y = 128$. The distance between the grid points was taken as $dx = dy = 0.5$. The nominal alloy composition was chosen as 15 at.% Cu, 1 at.% Ni, and 1 at.% Mn, and the annealing temperature was set as 823 °K. The time evolution of the microstructure is given in Fig. 5.8. As can be seen, the Cu-rich precipitates form homogeneously from the supersaturated solid solution first. At this stage, the Ni and Mn appear to be portioned between the solid solution and precipitates and the crystal structure of Cu precipitates is entirely in *bcc* structure. As Cu precipitates coarsen and grow in size, Ni and Mn move to interface region

between the Cu precipitates and the Fe matrix. This segregation more readily occurs around the larger Cu precipitates, as can be seen from the figure. As aging continues, a ring/shell layer forms in which Mn and Ni concentrations almost reach to 10 at.% around the Cu precipitates. As size of the Cu precipitates increases, they also slowly transform into *fcc* Cu which can be deduced from the change in order parameter values. The evolution kinetics seen in here agrees very well with the experiments and simulations given in [15].

As stated earlier, in phase-field models, the microstructure evolution is driven by the energy minimization. Therefore, it is naturally suitable for a second-order phase transitions via spinal decompositions. Phase-field simulations regarding the stability of Cu precipitates and their heterogeneous nucleation characteristics at different lattice defects for dilute concentrations can be found in [16].

The resulting movie files for each concentration field are given in subdirectory *case_study_8* in downloadable file.

5.6.5 Source Codes

Two source codes, *fft_FeCuMnNi_v1.m* in long-hand format and corresponding Matlab/Octave optimized version *fft_FeCuMnNi_v2.m* with the associated functions, are given below.

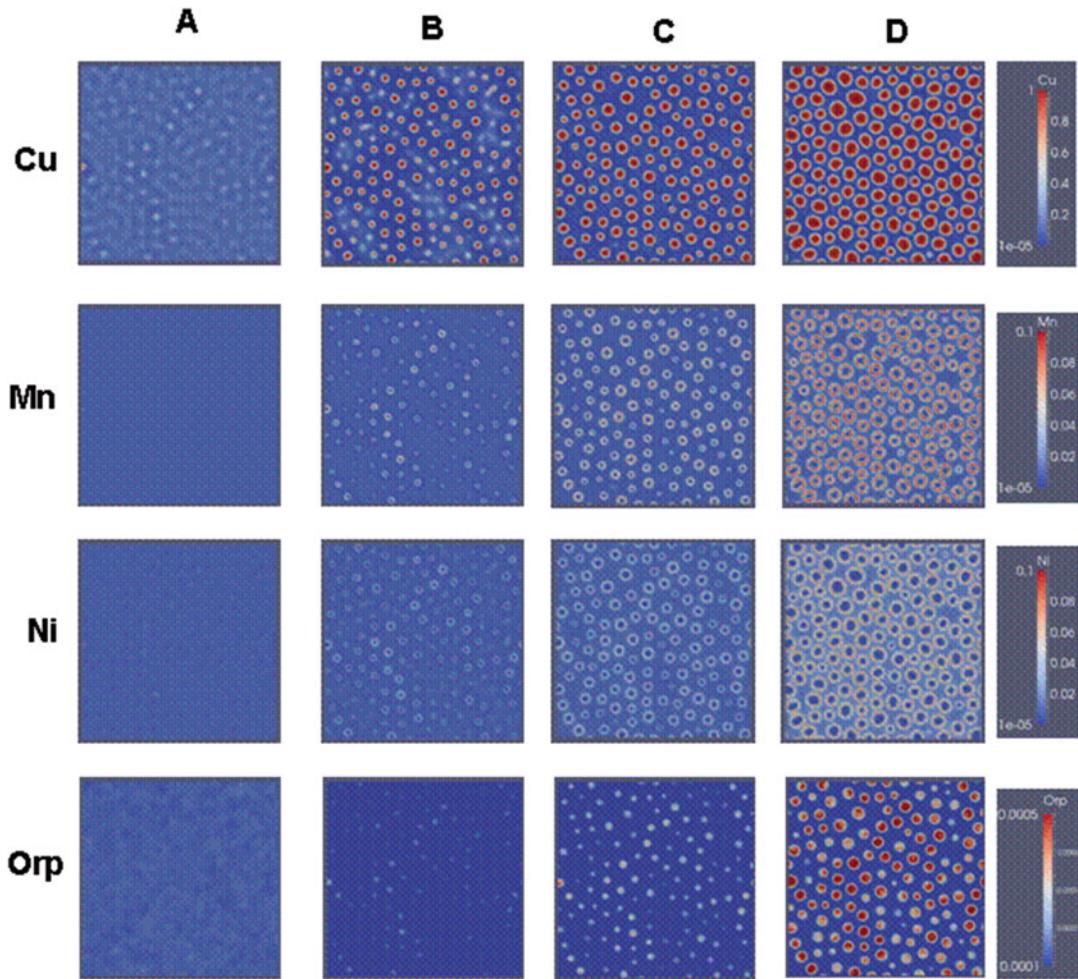


Fig. 5.8 Isothermal phase transformation of 15 at.% Cu–1 at.% Ni–1 at.% Mn alloy at 823 °K. Nondimensional times in columns are: (a) 25.0, (b) 30.0, (c) 37.5, and (d) 50.0

Program

fft_FeCuNiMn_v1.m

This program simulates the precipitation behavior in Fe-Cu-Mn-Ni alloys by semi-implicit Fourier spectral phase-field algorithm. The code is long-hand format and not optimized for Matlab/Octave.

The program makes calls to the following functions.

- **prepare_fft.m**
- **init_FeCuMnNi_micro.m**
- **Fe_Cu_Mn_Ni_free_energy.m**
- **write_vtk_grid_values.m**

Listing:

```

1 %%%%%%%%%%%%%%%%
2 %
3 % SEMI - IMPLICIT SPECTRAL %
4 % PHASE - FIELD CODE %
5 % FOR SOLVING PRECIPITATION IN %
6 % Fe_Cu_Ni_Mn ALLOY %
7 %
8 %== get intial wall time:
9 time0=clock();

```

```

10 format long;
11
12 %-- Simulation cell parameters:
13 Nx= 128;
14 Ny= 128;
15 NxNy =Nx*Ny;
16
17 dx=0.5;
18 dy=0.5;
19
20 %--- Time integration parameters:
21 nstep = 5000;
22 nprint= 50;
23 dtime = 1.0e-2;
24 ttime = 0.0;
25 coefA = 1.0;
26
27 %--- Material specific Parameters:
28 cu0=0.15;
29 mn0=0.01;
30 ni0=0.01;
31 grcoef_cu =0.68884;
32 grcoef_ni =0.68884;
33 grcoef_mn =0.68884;
34 grcoef_or =0.13729;
35
36 %Diffusivity Parameters for mobility:
37 gconst=8.314472;
38 temp=823.0;
39 RT=gconst*temp;
40 QACu=2.44e5;
41 QGCu=2.80e5;
42 D0ACu=4.7e-5;
43 D0GCu=4.3e-5;
44 %QANI=2.46e5;
45 QANI=2.56e5;
46 QGNI=2.73e5;
47 D0ANI=1.4e-4;
48 D0GNI=1.08e-5;
49 %QAMn=2.33e5;
50
51
52
53
54
55
56
57
58
59
60 QAMn=2.63e5;
61 QGMn=2.64e5;
62
63 D0AMn=1.49e-4;
64 D0GMn=2.78e-5;
65
66 DCuA=(D0ACu*exp(-QACu/RT));
67 DCuG=(D0GCu*exp(-QGCu/RT))/DCuA;
68
69 DNIa=(D0ANI*exp(-QANI/RT))/DCuA;
70 DNIg=(D0GNI*exp(-QGNI/RT))/DCuA;
71
72 DMnA=(D0AMn*exp(-QAMn/RT))/DCuA;
73 DMnG=(D0GMn*exp(-QGMn/RT))/DCuA;
74
75 DCuA=1.0;
76
77 %--- Prepare microstructure
78
79 [cu,mn,ni,orp] = init_FeCuNiMn_micro
(Nx,Ny,cu0,mn0,ni0);
80
81 %-- Prepare fft:
82
83 [kx,ky,k2,k4] = prepare_fft(Nx,Ny,
dx,dy);
84
85 ===
86 %--- Evolve:
87 ===
88
89 for istep=1:nstep
90
91 ttime=ttime+dtime;
92
93 cuk=fft2(cu);
94 mnk=fft2(mn);
95 nik=fft2(ni);
96 orpk=fft2(orp);
97
98 %--- derivative of free energy:
99
100 [dgdcu,dgdmn,dgdni,dgdor] = Fe_Cu_
Mn_Ni_free_energy...
101 (Nx,Ny,cu,mn,ni,orp,temp);
102
103
104 %--
105
106 dgdcuk=fft2(dgdcu);

```

```

107 dgdmnk = fft2(dgdmn);
108 dgdnik = fft2(dgdni);
109 dgdkrk = fft2(dgdor);
110
111 for i=1:Nx
112 for j=1:Ny
113
114 %--- Mobilities
115
116 mcoef_cu=cu0*(1.0-cu0)*(DCuA*
117 (1.0-orp(i,j))+DCuG*orp(i,j));
118 mcoef_ni=ni0*(1.0-ni0)*(DNiA*
119 (1.0-orp(i,j))+DNIg*orp(i,j));
120 mcoef_mn=mn0*(1.0-mn0)*(DMnA*
121 (1.0-orp(i,j))+DMnG*orp(i,j));
122 mcoef_orp=0.1;
123
124 %--- Time integration:
125
126
127 cuk(i,j) = (cuk(i,j) - dtim* mcoef_cu*
128 dgdcuk(i,j)*k2(i,j)) / ...
129 (1.0 + dtim*grcoef_cu*k4(i,j)*mcoef_cu);
130 nik(i,j) = (nik(i,j) - dtim* mcoef_ni*dgdn
131 ik(i,j)*k2(i,j)) / ...
132 (1.0 + dtim*grcoef_ni*k4(i,j)*mcoef_ni);
133 mnk(i,j) = (mnk(i,j) - dtim* mcoef_mn*
134 dgdmnk(i,j)*k2(i,j)) / ...
135 (1.0 + dtim*grcoef_mn*k4(i,j)*mcoef_mn);
136 orpk(i,j) = (orpk(i,j) - dtim*dgdork(i,j)
137 *mcoef_orp) / ...
138 (1.0 + dtim*grcoef_or*k2(i,j)*
139 mcoef_orp);
140
141 end
142 %--
143
144 cu = real(ifft2(cuk));
145 ni = real(ifft2(nik));
146 mn = real(ifft2(mnk));
147
148
149
150 orp = real(ifft2(orp));
151
152 %-- for small deviations:
153
154 for i=1:Nx
155 for j=1:Ny
156
157 if(cu(i,j) >= 0.9999);
158 cu(i,j) = 0.9999;
159 end
160 if(cu(i,j) < 0.00001);
161 cu(i,j) = 0.00001;
162 end
163 %
164 if(ni(i,j) >= 0.9999);
165 ni(i,j) = 0.9999;
166 end
167 if(ni(i,j) < 0.00001);
168 ni(i,j) = 0.00001;
169 end
170 %
171 if(mn(i,j) >= 0.9999);
172 mn(i,j) = 0.9999;
173 end
174 if(mn(i,j) < 0.00001);
175 mn(i,j) = 0.00001;
176 end
177
178 end%j
179 end%i
180 ==
181 %--- print results:
182 ==
183
184 if((mod(istep,npri) == 0) ||
185 (istep == 1))
186 fprintf('done step: %d\n',istep);
187
188 fname1=sprintf('time_%d.out',istep);
189 out1 = fopen(fname1,'w');
190
191 %for i=1:Nx
192 %for j=1:Ny
193 %fprintf(out1,'%5d%5d%14.6e%14.6e%14.6e
194 %14.6\n',i,j,cu(i,j),ni(i,j),...
195 mn(i,j),orp(i,j));
196 %end
197
198 fclose(out1);

```

```

199
200 %--- write vtk file:
201
202 write_vtk_grid_values(Nx,Ny,dx,dy,
203 istep,cu,ni,mn,orp);
204 end%endif
205
206 end%istep
207
208 %--- calculate compute time:
209
210 compute_time = etime(clock(),time0);
211 fprintf('Compute Time: %10d\n',
212 compute_time);
213

```

Line numbers:

9:	Get wall clock time at the beginning of execution.
12–20:	Simulation cell parameters.
14:	Number of grid points in the x -direction.
15:	Number of grid points in the y -direction.
16:	Total number of grid points in the simulation cell.
18:	The distance between two grid points in the x -direction.
19:	The distance between two grid points in the y -direction.
21–27:	Time integration parameters.
23:	Number of time steps.
24:	Output frequency to write the results to file.
25:	Time increment for numerical integration.
26:	Total time.
29–39:	Material-specific parameters.
31–33:	Initial concentrations of alloying elements.
40–75:	Diffusivity parameters for the mobility of alloying elements.
77–80:	Prepare initial microstructure.
81–84:	Calculate the coefficients of Fourier transform.
86–206:	Evolve the microstructure.
91:	Update the total time.
93–96:	Transform the alloying elements and the order parameter values from real space to Fourier space (forward FFT transformations).
98–102:	Calculate the derivatives of free energy for each alloying elements and the order parameter.
106–109:	Transform the derivative values from real space to Fourier space (forward FFT transformations).

114–123:	Calculate the mobility of each alloying elements, Eq. 5.24, based on the current values of concentration and order parameter.
124–138:	Semi-implicit time integration of each alloying elements and the order parameter.
143–151:	Bring back the integrated values from Fourier space to real space (inverse FFT transformations).
152–177:	For small deviations from max and min values, reset the limits.
184–204:	If print frequency is reached, output the results to file.
188–198:	Open an output file and write the results. These lines are commented, but they can be changed, including the output format.
202:	Write the results in vtk format for contour plots to be viewed by using Paraview.
208–211:	Calculate the compute time and print it to screen.

Program**fft_FeCuNiMn_v2.m**

This program simulates the precipitation behavior in Fe–Cu–Mn–Ni alloys by semi-implicit Fourier spectral phase-field algorithm. The code is optimized for Matlab/Octave.

The program makes calls to the following functions.

- **prepare_fft.m**
- **init_FeCuMnNi_micro.m**
- **Fe_Cu_Mn_Ni_free_energy.m**
- **write_vtk_grid_values.m**

Listing:

```

1 %%%%%%%%%%%%%%%%
2 %
3 % SEMI-IMPLICIT SPECTRAL %
4 % PHASE-FIELD CODE %
5 % FOR SOLVING PRECIPITATION IN %
6 % Fe_Cu_Ni_Mn ALLOY %
7 % (OPTIMIZED FOR MATLAB/OCTAVE %
8 %
9 %== get intial wall time:
10 time0=clock();
11 format long;
12

```

```

13 %-- Simulation cell parameters:          63
14                                         64 D0AMn=1.49e-4;
15 Nx=128;                                65 D0GMn=2.78e-5;
16 Ny=128;                                66
17 NxNy=Nx*Ny;                            67 DCuA=(D0ACu*exp(-QACu/RT));
18                                         68 DCuG=(D0GCu*exp(-QGCu/RT))/DCuA;
19 dx=0.5;                                 69
20 dy=0.5;                                70 DNiA=(D0ANI*exp(-QANI/RT))/DCuA;
21                                         71 DNiG=(D0GNi*exp(-QGNi/RT))/DCuA;
22 %--- Time integration parameters:      72
23                                         73 DMnA=(D0AMn*exp(-QAMn/RT))/DCuA;
24 nstep= 5000;                            74 DMnG=(D0GMn*exp(-QGMn/RT))/DCuA;
25 nprint= 50;                             75 DCuA=1.0;
26 dtime=1.0e-2;                           76
27 ttime=0.0;                             77
28 coefA=1.0;                            78 %--- Prepare microstructure
29                                         79
30 %--- Material specific Parameters:    80 [cu,mn,ni,orp]=init_FeCuNiMn_micro
31                                         (Nx,Ny,cu0,mn0,ni0);
32 cu0=0.15;                             81
33 mn0=0.01;                            82 %-- Prepare fft:
34 ni0=0.01;                            83
35 grcoef_cu=0.68884;                   84 [kx,ky,k2,k4]=prepare_fft(Nx,Ny,
36 grcoef_ni=0.68884;                   dx,dy);
37 grcoef_mn=0.68884;                   85
38 grcoef_or=0.13729;                   86 %==
39                                         87 %--- Evolve:
40                                         88 %==
41 %Diffusivity Parameters for mobility: 89
42                                         90 for istep=1:nstep
43 gconst=8.314472;                     91
44 tempr=823.0;                         92 ttime=ttime+dtime;
45 RT=gconst*tempr;                     93
46                                         94 cuk=fft2(cu);
47 QACu=2.44e5;                          95 mnk=fft2(mn);
48 QGCu=2.80e5;                          96 nik=fft2(ni);
49                                         97 orpk=fft2(orp);
50 D0ACu=4.7e-5;                         98
51 D0GCu=4.3e-5;                         99 %--- derivative of free energy:
52                                         100
53 %QANI=2.46e5;                         101 [dgdcu,dgdmn,dgdni,dgdor]=Fe_Cu_
54 QANI=2.56e5;                           Mn_Ni_free_energy...
55 QGNi=2.73e5;                           102 (Nx,Ny,cu,mn,ni,orp,tempr);
56                                         103
57 D0ANI=1.4e-4;                          104
58 D0GNi=1.08e-5;                         105 %--
59                                         106
60 %QAMn=2.33e5;                         107 dgdcuk=fft2(dgdcu);
61 QAMn=2.63e5;                           108 dgdmnk=fft2(dgdmn);
62 QGMn=2.64e5;                           109 dgdnik=fft2(dgdni);

```

```

110 dgdk = fft2(dgdk);
111
112 %--- Mobilities
113
114 mcoef_cu=cu0*(1.0-cu0)*(DCuA*
115     (1.0-orp)+DCuG*orp);
116 mcoef_ni=ni0*(1.0-ni0)*(DNiA*
117     (1.0-orp)+DNiG*orp);
118 mcoef_mn=mn0*(1.0-mn0)*(DMnA*
119     (1.0-orp)+DMnG*orp);
120 mcoef_orp=0.1;
121
122 %--- Time integration:
123
124 cuk=(cuk-dtime*mcoef_cu.*  

125     dgdk.*k2)./. . .  

126 (1.0+dtime*grcoef_cu*k4.*  

127     mcoef_cu);
128 nik=(nik-dtime*mcoef_ni.*  

129     dgdnik.*k2)./. . .  

130 (1.0+dtime*grcoef_ni*k4.*mcoef_ni);
131 mnk=(mnk-dtime*mcoef_mn.*  

132     dgdmnk.*k2)./. . .  

133 (1.0+dtime*grcoef_mn*k4.*mcoef_mn);
134 orpk=(orpk-dtime*dgdk.*  

135     mcoef_orp)./. . .  

136 (1.0+dtime*grcoef_or*k2.*  

137     mcoef_orp);
138 cu=real(ifft2(cuk));
139 ni=real(ifft2(nik));
140 mn=real(ifft2(mnk));
141
142 orp=real(ifft2(orpk));
143
144 %-- for small deviations:
145
146 inrange=(cu>=0.9999);
147 cu(inrange)=0.9999;
148
149
150 inrange=(cu<0.00001);
151 cu(inrange)=0.00001;
152
153 %
154 inrange=(ni>=0.9999);
155 ni(inrange)=0.9999;
156 inrange=(ni<0.00001);
157 ni(inrange)=0.00001;
158 %
159 inrange=(mn>=0.9999);
160 mn(inrange)=0.9999;
161 inrange=(mn<0.00001);
162 mn(inrange)=0.00001;
163
164 ====
165 %--- print results:
166 ====
167
168 if((mod(istep,nprint)==0)||  

169     (istep==1))
170 fprintf('done step: %5d\n',istep);
171
172 %fname1=sprintf('time_%d.out',  

173 istep);
174
175 %for i=1:Nx
176 %for j=1:Ny
177 %fprintf(out1,'%5d%5d%14.6e%14.6e  

178 %14.6e%14.6\n',i,j,cu(i,j),  

179     ni(i,j),...
180     mn(i,j),orp(i,j));
181 %end
182 %end
183
184 %--- write vtk file:
185
186 write_vtk_grid_values(Nx,Ny,dx,dy,  

187     istep,cu,ni,mn,orp);
188 end%end if
189
190 end%istep
191
192 %--- calculate compute time:
193

```

```

194 compute_time = etime(clock(), time0);
195 fprintf('ComputeTime: %10d\n',
    compute_time);

```

Line numbers:

10:	Get wall clock time at the beginning of execution.
13–21:	Simulation cell parameters.
15:	Number of grid points in the <i>x</i> -direction.
16:	Number of grid points in the <i>y</i> -direction.
17:	Total number of grid points in the simulation cell.
19:	The distance between two grid points in the <i>x</i> -direction.
20:	The distance between two grid points in the <i>y</i> -direction.
22–29:	Time integration parameters.
24:	Number of time steps.
25:	Output frequency to write the results to file.
26:	Time increment for numerical integration.
27:	Total time.
30–40:	Material-specific parameters.
32–34:	Initial concentrations of alloying elements.
41–77:	Diffusivity parameters for the mobility of alloying elements.
78–81:	Prepare initial microstructure.
82–85:	Calculate the coefficients of Fourier transform.
87–190:	Evolve the microstructure.
92:	Update the total time.
94–97:	Transform the alloying elements and the order parameter values from real space to Fourier space (forward FFT transformations).
99–103:	Calculate the derivatives of free energy for each alloying elements and the order parameter.
107–110:	Transform the derivative values from real space to Fourier space (forward FFT transformations).
112–120:	Calculate the mobility of each alloying elements, Eq. 5.24, based on the current values of concentration and order parameter.
122–135:	Semi-implicit time integration of each alloying elements and the order parameter.
138–145:	Bring back the integrated values from Fourier space to real space (inverse FFT transformations).
146–163:	For small deviations from max and min values, reset the limits.
168–188:	If print frequency is reached, output the results to file.
172–182:	Open an output file and write the results. These lines are commented, but they can be changed, including the output format.

186:	Write the results in vtk format for contour plots to be viewed by using Paraview.
192–195:	Calculate the compute time and print it to screen.

Function**init_FeCuNiMn_micro.m**

This function, for given initial alloy concentration Cu, Mn, and Ni, initializes the modulated concentrations and the order parameter with a noise term at the grid points.

Variable and array list:

Nx:	Number of grid points in the <i>x</i> -direction.
<b b="" ny:<="">	Number of grid points in the <i>y</i> -direction.
<b b="" cu0:<="">	Initial Cu concentration.
<b b="" mn0:<="">	Initial Mn concentration.
<b b="" ni0:<="">	Initial Ni concentration.
<b b="" cu(nx,ny):<="">	Modulated Cu values at the grid points.
<b b="" mn(nx,ny):<="">	Modulated Mn values at the grid points.
<b b="" ni(nx,ny):<="">	Modulated Ni values at the grid points.
<b b="" ny):<="" orp(nx,="">	Modulated order parameter values at the grid points.

Listing:

```

1 function [cu,mn,ni,orp] = init_
  FeCuNiMn_micro(Nx,Ny,cu0,mn0,ni0)
2
3 format long;
4
5
6 noise = 0.001;
7
8 for i=1:Nx
9 for j=1:Ny
10
11 cu(i,j)=cu0+noise*(0.5-rand());
12 mn(i,j)=mn0+noise*(0.5-rand());
13 ni(i,j)=ni0+noise*(0.5-rand());
14
15 orp(i,j)=0.001+noise*(0.5-rand());
16
17 end
18 end
19
20 end %endfunction

```

<i>Line numbers:</i>	
6:	The value of the noise term.
8–19:	Assign the modulated alloy concentrations and the order parameter values to the grid points.

Function

Fe_Cu_Mn_Ni_free_energy.m

This function evaluates the functional derivatives of Eq. 5.25 for each alloying element and order parameter at all grid points.

<i>Variable and array list:</i>	
Nx:	Number of grid points in the x -direction.
<b b="" ny:<="">	Number of grid points in the y -direction.
tempr:	Temperature.
cu(Nx,Ny):	Cu concentration values at the grid points.
mn(Nx,Ny):	Mn concentration values at the grid points.
ni(Nx,Ny):	Ni concentration values at the grid points.
orp(Nx,Ny):	Non-conserved order parameter value at the grid points.
dgdcu(Nx,Ny):	Functional derivative with respect to Cu
dgdmn(Nx,Ny):	Functional derivative with respect to Mn.
dgdni(Nx,Ny):	Functional derivative with respect to Ni.
dgdor(Nx,Ny):	Functional derivative with respect to non-conserved order parameter.

Listing:

```

55      +30000.0*c3*c1+36076.894 ...
56      *c1+6842.810456*(log(c2) ...
57      -log(c1))+(6252.0-9865.0*
58      (c2-c3))*c3 ...
59      -39865.0*c2*c3+1740.949*
60      c3-36076.894*c2+2984.135);
61      dgmnna=1.4613878411949395E-4*
62      (-201.364240 ...
63      *c1*c4+(6276.0*(c3-c4)
64      -48642.59) ...
65      *c4-(201.364240*(-2*c4-c3-
66      c2+1.0) ...
67      -20135.0*c2*c3+1740.949*c3+c2*
68      (6252.0-9865.0...
69      * (c2-c3))-36076.894*c2-33919.
70      89130169578);
71      dgnia=1.4613878411949395E-4*
72      (6842.810456 ...
73      *(log(c4)-log(c1))-402.
74      7284800 ...
75      *c1*c4-(201.364240* ...
76      (-2*c4-c3-c2+1.0)-2016.
77      04498)*c4-6276.0*c3*c4 ...
78      +(201.3642400*(-2*c4-c3-c2
79      +1.0)-2016.04498)* ...
80      c1+c3*(6276.0*(c3-c4)
81      -48642.59)-...
82      30000.0*c2*c3+1740.949*
83      c3-25404.848*c2 ...
84      +5788.49600);
85      %== Gamma phase:
86      dgcuug=1.4613878411949395E-4*
87      (c1* ...
88      (5672.81500*(c4+c3+2*c2-1.0)+
89      42968.802) ...
90      c2*(5672.8150*(c4+c3+2*c2-
91      1.0)+42968.802) ...
92      +(1451.610348*(-2*c4-c3-
93      c2+1.0)-...
94      7419.147789)*c1*c4-47841.3 ...
95      *c1*c4+(10672.046-2868.
96      3240* ...
97      (c2-c4))*c4-(-725.805174*
98      (-2*c4-c3-c2+1.0)^2 ...
99      +7419.147789*(-2*c4-c3-
100     -c2+1.0)-...
101     9359.746009*c4+44972.976*c2*
102     c4-26591.0* ...
103     c3*c1+11345.63*c2*c1 ...
104     -c3*(-259.0*(-c4-2*c3-c2
105     +1.0)-4581.105)+...
106     6842.810456*(log(c2)-
107     log(c1))+ ...
108     -1969.5*(c2-c3)^3-8131.0*
109     (c2-c3)+9927.1)*c3 ...
110     +c2*(-5908.5*(c2-c3)^2
111     -8131.0)*c3+26850.0*c2* ...
112     c3+566.3008361308123);

```

```

113 (c2-c3)^2+8131.0)*c3+26850.0* 144 *c4+c3*(6276.0*(c3-c4)
c2*c3+c2* ... -48642.59)*c4+10672.046 ...
114 (-1969.5*(c2-c3)^3-8131.0* 145 *c2*c4+5788.496000000001*c4
(c2-c3)+9927.1) ... +30000.0*c2*c3 ...
115 -33766.35316921635); 146 *c1-1740.949*c3*c1+ ...
116 147 36076.894*c2*c1+c2*(6252.0-...
117 148 9865.0*(c2-c3))*c3-33919.
118 89130169578*c3+...
119 dgning=1.4613878411949395E-4* 149 2984.135*c2);
(6842.810456*(log(c4) ...
120 log(c1))-c2*(5672. 151
815000000001 ...
121 *(c4+c3+2*c2-1.0)+42968.802) 152 gcuning=1.4613878411949395E-4*
+(2903.220696* ... 153 (6842.810456*(c4 ...
122 (-2*c4-c3-c2+1.0)-14838. 154 *log(c4)+log(c1)*c1 ...
29558)*c1 ...
123 *c4-(-725.805174*(-2*c4-c3-c2 155 *(5672.815000000001*
+1.0)^2+... (c4+c3+2*c2-1.0)+...
124 7419.147789999999*(-2*c4-c3- 156 42968.802)+(-725.805174*
c2+1.0)-... (-2*c4-c3-c2+1.0)^2 ...
125 9359.746009999999)*c4- 157 +7419.147789999999*(-2*c4
6276.0*c3*c4+... -c3-c2+1.0)-...
126 50709.624*c2*c4+(-725. 158 9359.746009999999)*c1*c4
805174* ... -47841.3 ...
127 (-2*c4-c3-c2+1.0)^2 159 *c2*c1*c4+c3*(6276.0*
+7419.147789999999* ... (c3-c4)-...
128 (-2*c4-c3-c2+1.0)-9359. 160 49205.406)*c4+c2*(10672.
746009999999)*... 046-2868.324000000001 ...
129 c1+259.0*c3*c1... 161 *(c2-c4))*c4+c3*(-259.0*
130 -42168.485*c2*c1+ 162 (-c4-2*c3-c2+1.0) ...
c3*(6276.0* ... -4581.105)*c1-26850.0*
131 (c3-c4)-49205.406) 163 c2*c3*...
-c3*(-259.0* ... 164 c1-566.3008361308123* ...
132 (-c4-2*c3-c2+1.0)-4581.105) 165 c1+c2*(-1969.5*(c2-c3)^3-...
+166 8131.0*(c2-c3)+9927.1)*c3-...
133 -2868.324000000001*(c2-c4)) 167 34332.65400534716*c3);
+26850.0*c2*c3+...
134 566.3008361308123); 168 %--
135 169
136 170 %== Cu:
137 171
138 %free energy for alfa gamma: 172 dgdcu(i,j)=(1.0-funch)*(dgcua
139 173 +delascu)+funch*dgcug;
140 gcunia=1.4613878411949395E-4* 174 %== Mn:
(6842.810456*(c4 ...
141 *log(c4)+log(c1)*c1 ... 175
142 +c3*log(c3)+c2*log(c2)) 176 dgdmn(i,j)=(1.0-funch)*(dgmna
+(201.364240000001*... +delasmn)+funch*dgmng;
143 (-2*c4-c3-c2+1.0) 177
-2016.04498)*c1 ...
144 178 %== Ni:

```

```

179
180 dgdni(i,j)=(1.0-funch)*(dgnia
+delasni)+funch*dgnig;
181
182 %== order parameter:
183
184 dg dor(i,j)=(gcunia+elaste)*(2.0*
orp(i,j)^2-2.0*(3.0...
185 -2.0*orp(i,j))*orp(i,j))
-2.0*constw*(1.0...
186 -orp(i,j))*orp(i,j)^2-2.0*
gcunig*orp(i,j)^2+...
187 2.0*constw*(1.0-orp(i,j))^2*
orp(i,j)+2.0*gcunig...
188 *(3.0-2*orp(i,j))*orp(i,j);
189
190
191 end%if
192
193 if(c1 >=0.9995)
194
195 funch=(3.0-2.0*orp(i,j))**
orp(i,j)^2;
196
197 funcg=(1.0-orp(i,j))*orp(i,j);
198
199
200 elaste=conste*((c4-c04)*eta4
+(c3-c03)*eta3...
201 +(c2-c02)*eta2)^2;
202
203 delascu=2.0*conste*eta2*((c4-c04)
*eta4+(c3-c03)...
204 *eta3+(c2-c02)*eta2);
205
206 delasmn=2.0*conste*eta3*((c4-c04)
*eta4+(c3-c03)...
207 *eta3+(c2-c02)*eta2);
208
209 delasni=2.0*conste*eta4*((c4-c04)
*eta4+(c3-c03)...
210 *eta3+(c2-c02)*eta2);
211
212
213 gcunia=Coef*((1.0-c2-c3-c4)
-0.9995)^2;
214 gcunig=Coef*((1.0-c2-c3-c4)
-0.9995)^2
215 dg cua=-2.0*Coef*((1.0-c2-c3-c4)
-0.9995);
216
217 dg cuug=-2.0*Coef*((1.0-c2-c3-c4)
-0.9995)
218
219 dg mna=-2.0*Coef*((1.0-c2-c3-c4)
-0.9995);
220 dg mnng=-2.0*Coef*((1.0-c2-c3-c4)
-0.9995);
221
222
223 dg nia=-2.0*Coef*((1.0-c2-c3-c4)
-0.9995);
224 dg nig=-2.0*Coef*((1.0-c2-c3-c4)
-0.9995);
225
226
227 dg dcu(i,j)=(1.0-funch)*(dg cua
+delascu)...
228 +funch*dg cuug;
229
230 dg dm n(i,j)=(1.0-funch)*(dg mna
+delasmn)...
231 +funch*dg mnng;
232
233 dg dn i(i,j)=(1.0-funch)*(dg nia
+delasni)...
234 +funch*dg nig;
235
236 dg dor(i,j)=(gcunia+elaste)*(6.0
*orp(i,j)^2...
237 -6.0*orp(i,j))+constw*(2.0
*orp(i,j)-6.0...
238 *orp(i,j)^2+4.0*orp(i,j)^3)
+gcunig...
239 *(6.0*orp(i,j)-6.0
*orp(i,j)^2);
240
241 end%if
242
243 if(c1 <=0.0005)
244
245 funch=(3.0-2.0*orp(i,j))
*orp(i,j)^2;
246
247 funcg=(1.0-orp(i,j))*orp(i,j);
248
249
250 elaste=conste*((c4-c04)*eta4
+(c3-c03)*eta3...
251 +(c2-c02)*eta2)^2;

```

```

252
253 delascu=2.0*conste*eta2*((c4-c04)
*eta4+(c3-c03) ...
254     *eta3+(c2-c02)*eta2);
255
256 delasmn=2.0*conste*eta3*((c4-c04)
*eta4+(c3-c03) ...
257     *eta3+(c2-c02)*eta2);
258
259 delasni=2.0*conste*eta4*((c4-c04)
*eta4+(c3-c03) ...
260     *eta3+(c2-c02)*eta2);
261
262 gcunia=Coef*((1.0-c2-c3-c4)
-0.0005)^2;
263 gcunig=Coef*((1.0-c2-c3-c4)
-0.0005)^2;
264
265 dgcuu=-2.0*Coef*((1.0-c2-c3-c4)
-0.0005);
266 dgcuu=-2.0*Coef*((1.0-c2-c3-c4)
-0.0005);
267
268 dgmnna=-2.0*Coef*((1.0-c2-c3-c4)
-0.0005);
269 dgmnng=-2.0*Coef*((1.0-c2-c3-c4)
-0.0005);
270
271 dggnia=-2.0*Coef*((1.0-c2-c3-c4)
-0.0005);
272 dggnig=-2.0*Coef*((1.0-c2-c3-c4)
-0.0005);
273
274 dgdcu(i,j)=(1.0-funch)*(dgcuu
+delascu) ...
275     +funch*dgcuu;
276
277 dgdmnn(i,j)=(1.0-funch)*(dgmnna
+delasmn) ...
278     +funch*dgmnng;
279
280 dgdnii(i,j)=(1.0-funch)*(dggnia
+delasni) ...
281     +funch*dggnig;
282
283 dgddor(i,j)=(gcunia+elaste)
*(6.0*orp(i,j)^2...
284     -6.0*orp(i,j))+constw*(2.0*orp
(i,j)-6.0...
285     *orp(i,j)^2+4.0*orp(i,j)^3)
+gcunig...
286     *(6.0*orp(i,j)-6.0*orp(i,j)^2);
287
288 end %if
289
290
291
292 end %i
293 end %j
294
295 end %endfunction
296

```

Line numbers:

6:	The value of gas constant.
9:	The value W in Eq. 5.25.
10:	The normalized value of Y in Eq. 5.25.
12–14:	Misfit strain values in Eq. 5.26 for each alloying element.
16–18:	Initial concentrations of alloying elements.
20:	Value of coefficient which will be used as a penalty parameter later.
25–28:	Current values of the alloying elements at the current grid point.
31:	If Cu concentration is within the limits continue
33:	Evaluate the value of function h .
35:	Evaluate the value of function g .
38:	Calculate the elastic energy.
41–48:	Derivative of elastic energy with respect to each alloying element.
50–80:	Calculate the functional derivatives for α phase.
52–60:	Functional derivative with respect to Cu concentration.
61–70:	Functional derivative with respect to Mn concentration.
71–80:	Functional derivative with respect to Ni concentration.
81–137:	Calculate the functional derivatives for γ phase.
83–98:	Functional derivative with respect to Cu concentration.
101–116:	Functional derivative with respect to Mn concentration.
119–135:	Functional derivative with respect to Ni concentration.
138–168:	Calculate the derivatives of free energy which will be used later for the evaluation of derivative with respect to order parameter.
171–173:	Derivative of free energy based on the function h for Cu concentration.

(continued)

174–177:	Derivative of free energy based on the function h for Mn concentration.
178–181:	Derivative of free energy based on the function h for Ni concentration.
182–189:	Derivative of free energy with respect to order parameter.
193–288:	if Cu concentration is greater than 0.9995 or less than 0.0005 apply a penalty term. This is a numerical trick to avoid the overflows because of presence of log terms in free energy functional Eq. 5.26.
195:	Evaluate the value of function h .
197:	Evaluate the value of function g .
200–210:	Evaluate the elastic energy and its derivatives for each alloying element.
213–225:	Calculate the derivatives of penalty term for each alloying elements.
227–241:	Derivatives of free energy based on function h for each alloying elements.
243–288:	Repeat the same, however, this time if Cu concentration is less than 0.0005.

9. Becquart CS, Domain C (2011) Modeling microstructure and irradiation effects. *Metall Trans A* 42:852
10. Mathon MH, Barbu A, Dunstetter F, Maury F, Lorenzelli N, de Novion CH (1997) Experimental study and modeling of copper precipitation under electron irradiation in dilute FeCu binary alloy. *J Nucl Mater* 245:224
11. Duparc AH, Moingeon C, Smetniansky-de Grande N, Barbu A (2002) Microstructure modeling of ferritic alloys under high flux 1MeV electron irradiation. *J Nucl Mater* 302:143
12. Domain C, Becquart CS, Malerba L (2004) Simulation of radiation damage in Fe alloys: an object kinetic Monte Carlo approach. *J Nucl Mater* 335:121
13. Vincent E, Becquart CS, Pareige C, Pareige P, Domain C (2008) Precipitation of FeCu system A critical review of atomic kinetic Monte Carlo simulations. *J Nucl Mater* 373:387
14. Koyoma T, Onodera H (2005) Computer simulation of phase decomposition in Fe-Cu-Mn-Ni quaternary alloy based on the phase-field method. *Mater Trans* 46:1187
15. Koyoma T, Hashimoto K, Onodera H (2006) Phase-field simulations of phase-transformations in Fe-Cu-Mn-Ni quaternary alloy. *Mater Trans* 47:2765
16. Biner SB, Weifeng R, Yonfeng Z (2016) The stability of precipitates and the role of lattice defects in Fe-1at%Cu-1at%Ni-1At%Mn alloy: a phase-field model study. *J Nucl Mater* 468:9

References

1. Miller MK, Russell KF (2007) Embrittlement of RPV steel: an atom probe tomography perspective. *J Nucl Mater* 371:145
2. Monzen R, Iguchi M, Jenkins ML (2000) Structural changes of 9R copper precipitates in an aged Fe-Cu alloy. *Philos Mag Lett* 80:137
3. Othen PJ, Jenkins ML, Smith GDW (1994) High-resolution electron microscopy studies of the structure of Cu precipitates in α -Fe. *Philos Mag A* 70:1
4. Goodman SR, Brenner SS, Low JR (1973) An FIM-atom probe study of precipitation of copper from iron-1.4 pct copper. *Metall Trans A* 4:2363
5. Pareige PJ, Russell KF, Miller MK (1996) APFIM studies of the phase transformations in thermally aged ferritic FeCuNi alloys comparison with aging under neutron irradiation. *Appl Surf Sci* 94:362
6. Nakamichi H, Yamada K (2010) Sub-nanometre elemental analysis of Cu cluster in Fe-Cu-Ni alloy using aberration corrected STEM-EDS. *J Microscopy* 242:1
7. Wen YR, Hirata A, Zhang ZW, Fujita T, Liu CT, Jiang JH, Chen MW (2013) Microstructure characterization of Cu-rich nanoprecipitates in Fe-2.5Cu-1.5Mn-4.0Ni-1.0Al multicomponent ferritic alloy. *Acta Mater* 61:2133
8. Miller MK, Burke MG (1992) An atom probe field ion microscopy study of neutron-irradiated pressure vessel steel. *J Nucl Mater* 195:68

5.7 Case Study-IX

The role of elastic inhomogeneities and applied stresses on the phase separation behavior of a binary alloy

Objective:

The objective of this case study is to introduce an efficient Fourier spectral method to account the mechanical equilibrium resulting from the mismatch of mechanical properties of evolving phases and applied stress fields in phase-field modeling of microstructure evolution.

5.7.1 Background

During the microstructure evolution, stress-strain fields develop owing to the differences in the lattice parameters of the phases. In addition to

these intrinsic stress-strain fields, externally applied stresses may also be present. These stress-strain fields contribute to the total energy in form of elastic energy and have profound effect on the morphology of evolving microstructures; therefore, they should be included into the simulations for these cases. The examples can be seen in nickel-based supper alloys [1–3], precipitation behavior of hydrides in zirconium-alloys [4, 5], and instabilities in thin films and fiber reinforced composites [6]. The preferential coarsening of misfitting precipitates with respect to applied stress direction is termed as rafting [7, 8] and studied at the atomistic level [9, 10] and at the meso-scale with the phase-field models [11–15].

5.7.2 Phase-Field Model

For simplicity, a binary alloy undergoing spinodal decomposition as studied in *Case Studies-I* and *VI* is considered again. For this case, the total free energy is assumed to be additive and composed of chemical and elastic energies. Then, the standard evolution equation for Cahn–Hilliard equation can be expressed as:

$$\frac{\partial c}{\partial t} = \nabla \cdot M \nabla \left(\frac{\delta F^{CH}}{\delta c} + \frac{\delta F^{EL}}{\delta c} \right) \quad (5.29)$$

F^{CH} is expressed, as in previous cases, as

$$F^{CH} = \int_V \left[\frac{\delta f(c)}{\delta c} + \frac{1}{2} \kappa |\nabla c|^2 \right] dv \quad (5.30)$$

again, a simple double-well potential for $f(c)$ is assumed.

$$f(c) = Ac^2(1 - c)^2 \quad (5.31)$$

in which κ is the gradient energy coefficient and A is a positive constant and controls the magnitude of the energy barrier between two equilibrium phases (see Fig. 4.3).

The elastic energy contributing to total free energy is taken as:

$$F^{EL} = \frac{1}{2} \int_V \sigma_{ij} \epsilon_{ij}^{el} dv \quad (5.32)$$

in which σ_{ij} and ϵ_{ij}^{el} are the stresses and elastic strains, respectively, and elastic strains expressed as

$$\epsilon_{ij}^{el} = \epsilon_{ij} - \epsilon_{ij}^0 \quad (5.33)$$

where ϵ_{ij}^0 is the position- and composition-dependent eigenstrains, ϵ_{ij} is the total strain and given by:

$$\epsilon_{ij} = \frac{1}{2} \left[\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right] \quad (5.34)$$

in which u and x are the displacement and position vectors, respectively.

Assuming that all the phases behaves linear elastic and obeys the Hooks law, the stresses are obtained from,

$$\sigma_{ij} = C_{ijkl} \epsilon_{ij}^{el} \quad (5.35)$$

where C_{ijkl} is position-dependent (also composition-dependent) elastic modulus tensor.

Because of dependency of the elastic modulus also the dependency of eigenstrains to composition and to position, the evolution equation, Eq. 5.29, becomes fully coupled system of equations requiring the solution of the mechanical equilibrium,

$$\frac{\partial \sigma_{ij}}{\partial x_i} = 0 \text{ in } V \quad (5.36)$$

5.7.3 Solving the Equation of Mechanical Equilibrium

To solve the equation of mechanical equilibrium with the Fourier spectral method, the algorithm given by Moulinec and Suquet [16–19] is adopted in here; although, there are some others also available in the literature [15, 20, 21]. The reason for this choice is its simplicity in implementation and also it can be easily adapted to cases where material behaves nonlinearly (i.e., elastic-plastic, elastic-viscoplastic, and

crystal plasticity) as given in [18, 22, 23]. In addition, it allows the solutions to be obtained both under applied stresses and strains to the system.

For given a microstructure that is composed of different phases, the algorithm first considers an auxiliary problem of a homogeneous material with stiffness C_{ijkl}^0 subject to a periodic polarization field $\tau(x)$ of which solution can be found in textbooks [e.g., 24]. The solution can be expressed in terms of the Fourier transform of the polarization field by means of the Fourier transform of the Green's operator in the form of following equations:

In real space:

$$\begin{aligned}\sigma_{ij} &= C_{ijkl}^0 \epsilon(u^*(x)) + \tau_{ij}(x) \\ \operatorname{div} \sigma_{ij}(0), \quad \sigma_{ij} n_j - &\neq, \quad u^*(x) \neq\end{aligned}\quad (5.37)$$

In Fourier space, these equations take the form:

$$\begin{aligned}\hat{\sigma}_{ij}(\xi) &= iC_{ijkl}^0 \xi_l \hat{u}_k(\xi) + \hat{\tau}_{ij}(\xi) \\ i\hat{\sigma}_{ij}(\xi) \xi_j &= 0\end{aligned}\quad (5.38)$$

in which ξ is the Fourier frequencies or vectors and $i = \sqrt{-1}$. Eliminating $\hat{\sigma}_{ij}(\xi)$ between two equations gives

$$K_{ik}^0(\xi) \cdot u_k^* = \hat{\tau}_{ij}(\xi) \xi_j \quad (5.39)$$

where $K^0(\xi)$ is the acoustic tensor of the homogeneous material which takes the form:

$$K_{ik}^0(\xi) = C_{ijkl}^0 \xi_l \xi_k \quad (5.40)$$

Then, the displacement field can be expressed as:

$$\begin{aligned}\hat{u}_k^*(\xi) &= iN_{ki}^0(\xi) \hat{\tau}_{ij}(\xi) \xi_j \\ &= \frac{i}{2} \left(N_{ki}^0(\xi) \xi_j + N_{kj}^0(\xi) \xi_i \right) \hat{\tau}_{ij}(\xi)\end{aligned}\quad (5.41)$$

in which the symmetry of τ is used. The $N^0(\xi)$ denotes the cofactors of inverse of matrix of $K^0(\xi)$, i.e.,

$$N^0(\xi) = [K^0(\xi)]^{-1} \quad (5.42)$$

The resulting strain field, then, can be expressed as

$$\begin{aligned}\hat{\epsilon}_{kh}(u^*) &= \frac{i}{2} (\xi_h \hat{u}_k(\xi) + \xi_k \hat{u}_h(\xi)) \\ &= \Gamma_{khij}^0(\xi) \hat{\tau}_{ij}(\xi)\end{aligned}\quad (5.43)$$

with

$$\begin{aligned}\Gamma_{khij}^0 &= \frac{1}{4} \left[N_{hi}^0(\xi) \xi_j \xi_k + N_{ki}^0(\xi) \xi_j \xi_h + N_{hj}^0(\xi) \xi_i \xi_k \right. \\ &\quad \left. + N_{kj}^0(\xi) \xi_i \xi_h \right] \quad (5.44)\end{aligned}$$

and

$$\hat{\tau}_{ij}(\xi) = \langle \tau_{ij}(x) e^{-i\xi \cdot x} \rangle \quad (5.45)$$

The strain field induced at each grid point in the periodic simulation cell with an initial stress τ can be determined through Eqs. 5.43, 5.44, and 5.45 iteratively.

The numerical implementation of the algorithm follows the following steps:

Initialization:

$$\begin{aligned}\epsilon^0(x) &= E \\ \sigma^0(x) &= C(x) : \epsilon^0(x)\end{aligned}$$

Iterate $i + 1$ ϵ^i and σ^i are known at every grid point x

- (a) $\hat{\sigma}^i = \text{FFT}(\sigma^i)$ and $\hat{\epsilon}^i = \text{FFT}(\epsilon^i)$
- (b) $\hat{\epsilon}^{i+1}(\xi) = \hat{\epsilon}^i(\xi) - \hat{\Gamma}(\xi) : \hat{\sigma}^i(\xi)$
- (c) $\epsilon^{i+1}(x) = \text{FFT}^{-1}(\hat{\epsilon}^{i+1}(\xi))$
- (d) $\sigma^{i+1}(x) = C(x) : \epsilon^{i+1}(x)$
- (e) Check convergence

(5.46)

In the above, FFT is the forward Fourier transformations of the terms inside the parenthesis, and similarly, FFT^{-1} is the inverse Fourier transforms. As discussed in detail in [19], the convergence rate is proportional to the differences in the elastic properties of the phases; however, except very specific cases, usually the convergence rate is fast. The elastic constants of the reference medium, C^0 , is usually taken as the arithmetic averages of the elastic properties of the phases.

The detailed expressions of Green's tensor Γ^0 for different anisotropy of the reference material can be found in [24]. As can be seen from Eqs. 5.40, 5.42, and 5.44, the calculation of Green's tensor is computationally little demanding. However, after algebraic manipulations and collection of the terms together, it can be put into simple tensor form for cubic anisotropy, which is given in function *green_tensor.m* for the 2D case. The 3D version which takes into account the six components of the strains and stresses is given in *Appendix-B* for those who wish to expand their analysis into the 3D.

5.7.4 Numerical Implementation

After, solving stress and strain values at every grid points with the algorithm discussed above, the solution of Eq. 5.29 follows the same steps as given in *Case Study-VI*. Specifically, by taking Fourier transform of both side of Eq. 5.29 the spatial discretization becomes:

$$\frac{\partial \{c\}_k}{\partial t} = -k^2 M \left[\left\{ \frac{\delta f}{\delta c} \right\}_k + k^2 \kappa \{c\}_k + \left\{ \frac{\delta F^{EL}}{\delta c} \right\}_k \right] \quad (5.47)$$

where $\{\cdot\}_k$ is the Fourier transform of the quantity inside the bracket and k is the vector in Fourier space. Again, expending Eq. 5.47

$$\begin{aligned} \frac{\partial \{c\}_k}{\partial t} &= -k^2 M \left\{ \frac{\delta f}{\delta c} \right\}_k - k^4 M \kappa \{c\}_k \\ &\quad - k^2 M \left\{ \frac{\delta F^{EL}}{\delta c} \right\}_k \end{aligned} \quad (5.48)$$

Similarly, by treating linear and fourth-order operators implicitly and the nonlinear terms explicitly, the semi-implicit form for Eq. 5.29 is

$$\begin{aligned} \frac{\{c\}_k^{n+1} - \{c\}_k^n}{\Delta t} &= -k^2 M \left\{ \frac{\delta f}{\delta c} \right\}_k^n - k^2 M \left\{ \frac{\delta F^{EL}}{\delta c} \right\}_k^n \\ &\quad - k^4 M \kappa \{c\}_k^{n+1} \end{aligned} \quad (5.49)$$

Where Δt is the time increment between time steps $n+1$ and n . By rearranging it becomes

$$\{c\}_k^{n+1} = \frac{\{c\}_k^n - \Delta t k^2 M \left[\left\{ \frac{\delta f}{\delta c} \right\}_k^n + \left\{ \frac{\delta F^{EL}}{\delta c} \right\}_k^n \right]}{1 + \Delta t k^4 M \kappa} \quad (5.50)$$

and the problem again reduces to solving the discretized Eq. 5.50.

The numerical steps to achieve this:

For number of time steps repeat:

1. Using the current time step value of c , solve stress and strain values and evaluate $(\delta F^{EL}/\delta c)$ and $(\delta f/\delta c)$. Fourier transform $(\delta F^{EL}/\delta c)$, $(\delta f/\delta c)$, and c .
2. Calculate the spatial variation of c in Fourier space for the time $t + \Delta t$ using Eq. 5.50.
3. With inverse Fourier transformation bring back the results to real space for the solution at $t + \Delta t$.

5.7.5 Results and Discussion

Before the analysis of phase separation behavior due to the elastic inhomogeneity and the applied loads, the algorithm that calculates the stress and strain distribution in the simulation cell is tested first. For this purpose, a simulation cell having number of grid points $N_x = N_y = 256$ with grid spacing $dx = dy = 1.0$ was utilized. Either a spherical hole or a spherical hard particle with a radius of $20dx$ was placed in the center of the simulation cell. In the case of spherical hole the system is strained in the axial direction with an applied strain of $\epsilon_{22} = 0.01$. In the case of hard particle, there was no applied load and stresses were generated due to the dilatational misfit strain of the particle. The ratio of the elastic constants of the matrix and particle was taken as $\frac{C_p}{C_m} = 2.0$ and the misfit strains (eigenstrains) were assumed to be $\epsilon_{ij}^0 = \pm 0.01 \delta_{ij}$, in which δ_{ij} is the Kronecker delta function. These results are summarized in Fig. 5.9. In the figure, the variation of the axial stress σ_{22} along the dashed line in Fig. 5.9a is shown for all three cases. As can be seen, in the case of the hole the stress values

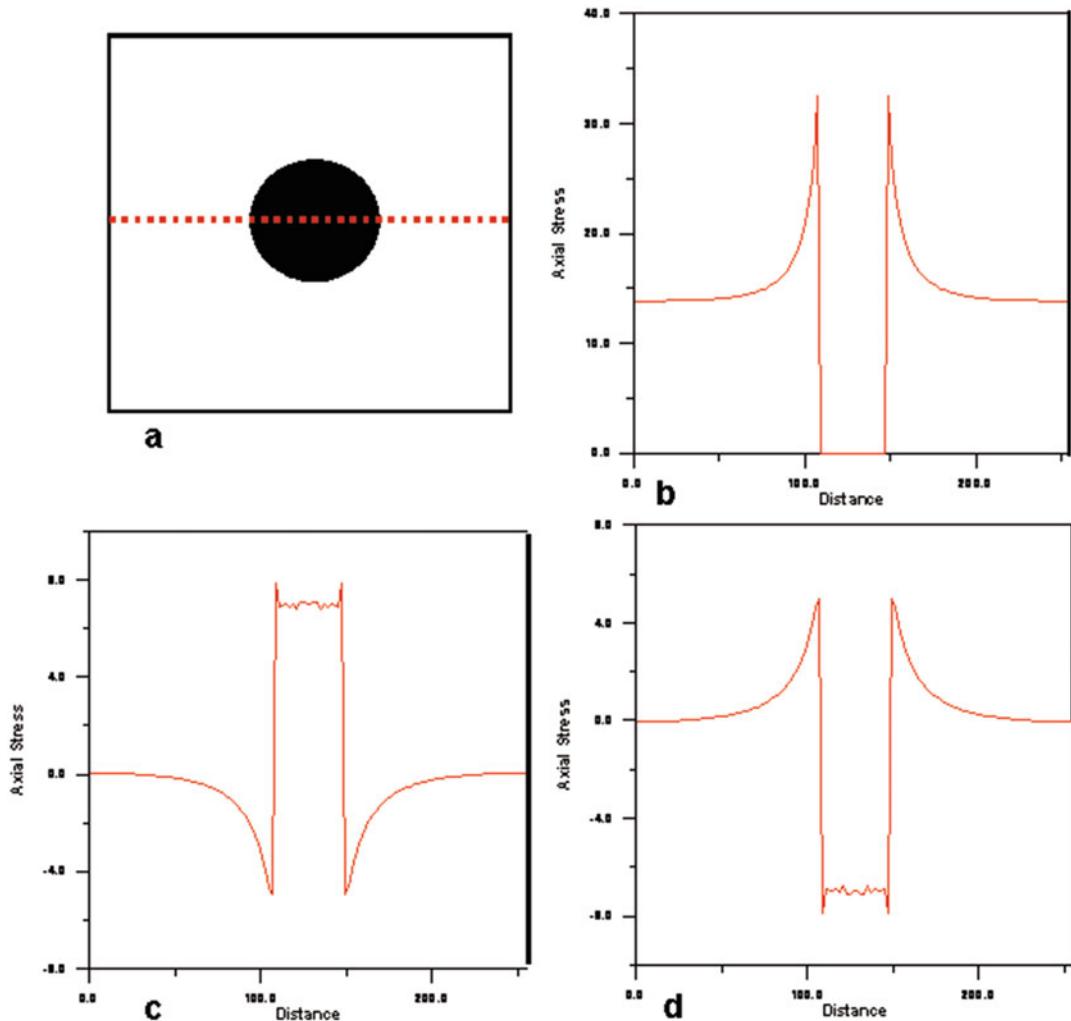


Fig. 5.9 (a) Schematic drawing of the simulation cell, the back circular region represents either a hole or a hard particle, (b-d) shows the variation of the axial stress

along the dashed line. (b) For hole with applied strain of 0.01, (c) for particle with a positive misfit strain, and (d) particle with a negative misfit strain

increase to near the theoretical value of 3.0 near the hole. In the case of hard particle, the resulting stress field depends on the sign of the misfit strains. For a particle having negative misfit strains, large tensile stresses develop at the particle and it puts the nearby regions into the compression. Opposite develops for particle having positive misfit strains. For both cases, the stress values approach to zero towards the edges of the simulation cell as predicted from the analysis of Eshelby [25].

Next, the results obtained from set of simulation of phase separation of a binary alloy are presented. During the simulations, the elastic

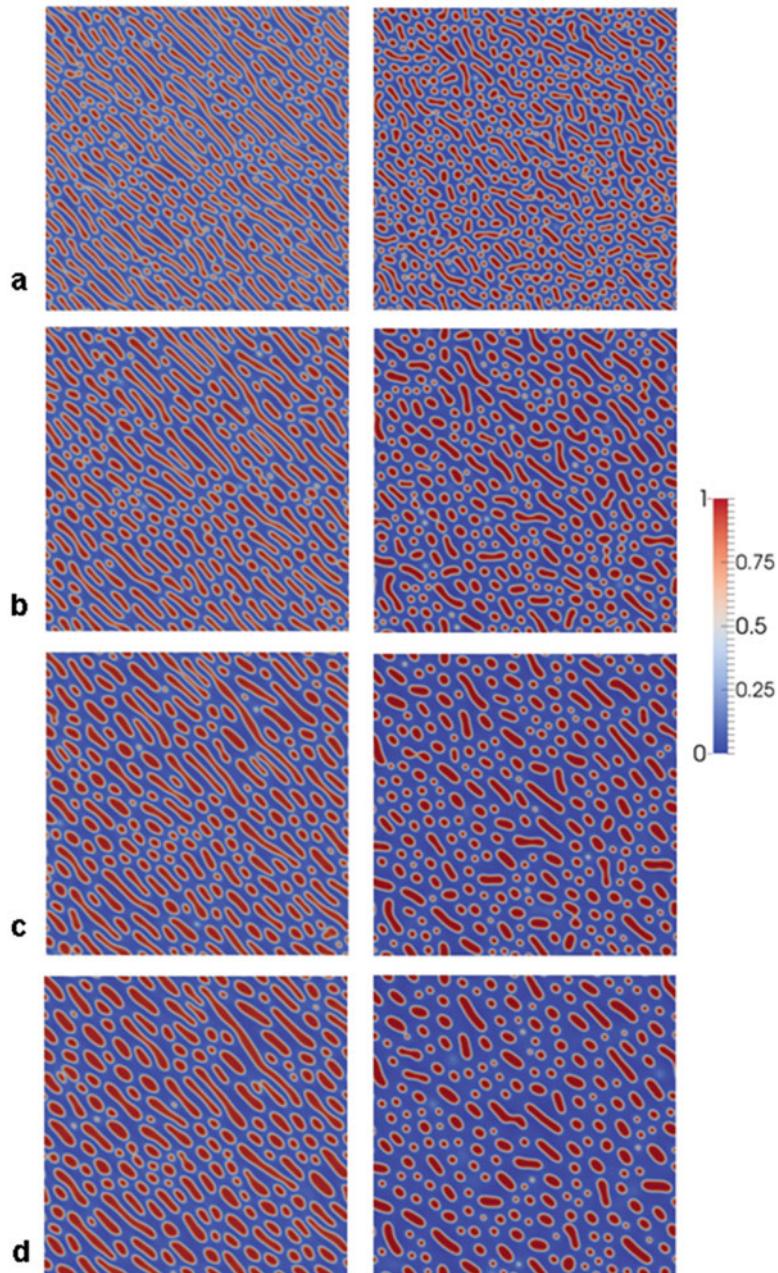
inhomogeneity is introduced as the differences between the elastic constants of the matrix C^M and precipitation phase C^P . For the harder precipitation phase, the ratio of elastic constants C^P/C^M was taken as 2 and for the soft precipitates this ratio was taken as 0.5. For both, matrix and precipitation phases are assumed to be elastically isotropic; although, the code allows full cubic anisotropy. The eigenstrains of precipitates (for both hard and soft) are assumed to be positive without the shear component. The parameters used in the simulations, in nondimensional form, are given in Table 5.2. The number of grid points in the simulations was $Nx = Ny = 256$.

Table 5.2 The nondimensional parameters used in the simulations

M	κ	C_{11}^M	C_{12}^M	C_{44}^M	ϵ_i^0	Δt	$dx = dy$
1.0	0.5	1400	600	400	0.01	0.05	1.0

In the first set of simulations, just the role of elastic inhomogeneity is elucidated for alloy concentration 0.4. Without the elastic inhomogeneity, the results would be the same to that seen in *Case Studies-I* and *VI*. The time evolution behavior of hard and soft phases is summarized in Fig. 5.10. As can be seen, the elastic

Fig. 5.10 The evolution of hard (left column) and soft (right column) particles. (a) 1250, (b) 2500, (c) 3750, and (d) 5000 time steps, respectively

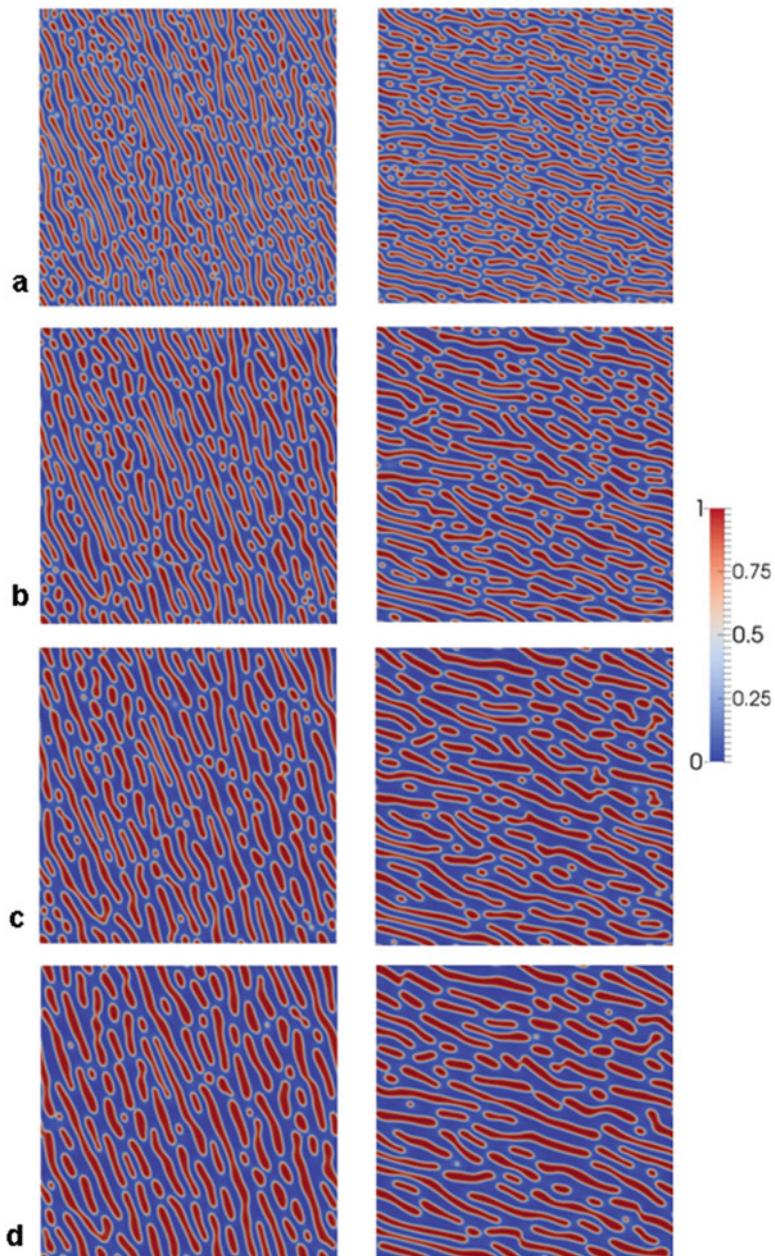


inhomogeneity has profound effect on the microstructure evolution. While the hard particles seem to be mostly aligned along $<11>$ direction, there is no preferential alignment for the soft particles. In addition, the differences in the size and the density of the particles between the two cases can also be discerned from the figure.

In the next set simulations, axial strain of 0.01 is applied to the simulation cell. For this case, the time evolution of particles is summarized in Fig. 5.11.

As can be seen, the hard particles now align parallel to applied stress direction, while soft particles align normal to the applied stress

Fig. 5.11 The evolution of hard (left column) and soft (right column) particles under applied tensile strain of 0.01. (a) 1250, (b) 2500, (c) 3750, and (d) 5000 time steps, respectively



direction. These results are consistent with the theoretical study [26] and with the phase-field model given in [15].

The simulations presented in here are for isotropic elasticity. The role of the elastic anisotropy, the sign of misfit strains, and other applied stress configurations on rafting can be easily explored with the code provided.

The movie files resulting from these simulations can be found in subdirectory *case_study_9* in downloadable file.

5.7.6 Source Codes

Two source codes, *fft_raft_v1.m* in longhand format and corresponding Matlab/Octave optimized version *fft_raft_v2.m* with the associated functions, are given below.

Program

fft_raft_v1.m

This program solves Cahn–Hilliard phase-field equation with semi-implicit Fourier spectral method by taking into account the effects of elastic inhomogeneities and applied stresses based on solution of stress–strain fields with Green’s tensor and Fourier transformations. The time integration is carried out by using semi-implicit time marching scheme. The code is longhand format and it is not optimized.

The program makes calls to the following functions:

- **prepare_fft.m**
- **free_energ_ch_v1.m**
- **green_tensor.m**
- **solve_elasticity_v1.m**
- **micro_ch_pre.m**
- **write_vtk_grid_values.**

Listing:

```

1 %%%%%%%%%%%%%%%%
2 %
3 % SEMI-IMPLICIT SPECTRAL %
4 % PHASE-FIELD %
5 % FOR SOLVING PRECIPITATION %
6 % UNDER STRESS %
7 %
8 %% get intial wall time:
9 time0=clock();
10 format long;
11
12 %%-- Simulation cell parameters:
13
14 Nx=256;
15 Ny=256;
16 NxNy =Nx*Ny;
17
18 dx=1.0;
19 dy=1.0;
20
21 %--- Time integration parameters:
22
23 nstep = 5000;
24 nprint= 25;
25 dtime = 5.0e-2;
26 ttime = 0.0;
27 coefA = 1.0;
28
29 %--- Material specific Parameters:
30
31 c0 =0.40;
32 mobility =1.0;
33 grad_coef=0.5;
34
35 %%-- elastic constants:
36
37 cm11 = 1400.0;
38 cm12 = 600.0;
39 cm44 = 400.0;
40 %
41 cp11 = 2.0*cm11;
42 cp12 = 2.0*cm12;
```

```

43 cp44 = 2.0*cm44;
44
45 %--eigen strains:
46 ei0= 0.01;
47
48 %-- Applied strains:
49
50 ea(1)=0.0;
51 ea(2)=0.01;
52 ea(3)=0.0;
53
54 %--Initialize stress & strain
55 %componetes
56
57 for i=1:Nx
58 for j=1:Ny
59
60 s11(i,j)=0.0;
61 s22(i,j)=0.0;
62 s12(i,j)=0.0;
63 %
64 e11(i,j)=0.0;
65 e22(i,j)=0.0;
66 e12(i,j)=0.0;
67
68 ed11(i,j)=0.0;
69 ed22(i,j)=0.0;
70 ed12(i,j)=0.0;
71
72 end
73 end
74 %--- Prepare microstructure
75
76 iflag = 1;
77
78 [con] = micro_ch_pre(Nx,
79 Ny,c0,iflag);
80
81 %-- Prepare fft:
82
83 [kx,ky,k2,k4] = prepare_fft(Nx,
84 Ny,dx,dy);
85
86 %---- Greens tensor
87
88 [tmatx] = green_tensor(Nx,Ny,kx,ky,
89 cm11,cm12,cm44, ...
90 cp11,cp12,cp44);
91
92 %==
93
94 ttime = ttime + dtim;
95
96 %--- derivative of free energy:
97 for i=1:Nx
98 for j=1:Ny
99
100 [dummy]=free_energ_ch_v1
101 (i,j,con);
102 dfdcon(i,j) =dummy;
103
104 end
105
106 %---derivative of elastic energy:
107
108 [delsdc ] = solve_elasticity_v1(Nx,
109 Ny,tmatx,kx,ky,s11,s22,s12, ...
110 e11,e22,e12,ed11,ed22,ed12, ...
111 cm11,cm12,cm44,cp11,cp12, ...
112 cp44,ea,ei0,con);
113
114 %--
115 conk = fft2(con);
116 dfdconk = fft2(dfdcon);
117
118 delsdck = fft2(delsdc);
119
120 %--- Time integration:
121
122 for i=1:Nx
123 for j=1:Ny
124
125 numer = dtim*mobility*k2(i,j)*
126 (dfdconk(i,j) + delsdck(i,j));
127 denom = 1.0 + dtim*coefA*mobility*
128 grad_coef*k4(i,j);
129 conk(i,j) =(conk(i,j) - numer) ./ ...
130 denom;
131
132 end
133
134
135 con = real(ifft2(conk));

```

```

136
137 %-- for small deviations:
138
139 for i=1:Nx
140 for j=1:Ny
141
142 if(con(i,j) >= 0.9999);
143 con(i,j) = 0.9999;
144 end
145
146 if(con(i,j) < 0.00001);
147 con(i,j) = 0.00001;
148 end
149
150 end
151 end
152
153 ====
154 %--- print results:
155 ====
156
157 if((mod(istep,nprint) == 0) ||
158 (istep == 1))
159 fprintf('done step: %5d\n',istep);
160
161 %fname1 = sprintf('time_%d.out',
162 %istep);
163 %out1 = fopen(fname1,'w');
164 %for i=1:Nx
165 %for j=1:Ny
166 %fprintf(out1,'%5d %5d %14.6e\n',i,
167 %j,con(i,j));
168 %end
169 %end
170 %fclose(out1);
171
172 %--- write vtk file:
173
174 write_vtk_grid_values(Nx,Ny,dx,
175 dy,istep,con);
176 end %end if
177
178 end %istep
179
180 %--- calculate compute time:
181

```

```

182 compute_time = etime(clock(),
183 time0);
184 fprintf('Compute Time: %10d\n',
185 compute_time);
186

```

Line numbers:

8:	Get initial wall clock time beginning of the execution.
12–20:	Simulation cell parameters.
14:	Number of grid points in the <i>x</i> -direction.
15:	Number of grid points in the <i>y</i> -direction.
16:	Total number of grid points in the simulation cell.
18:	The distance between two grid points in the <i>x</i> -direction.
19:	The distance between two grid points in the <i>y</i> -direction.
21–28:	Time integration parameters.
23:	Number of time steps.
24:	Print frequency to write the results to file.
25:	Time increment for numerical integration.
26:	Total time
29–34:	Material-specific parameters.
31:	Initial concentration.
32:	The value of mobility coefficient.
33:	The value of gradient energy coefficient.
35–44:	Elastic constants.
37–39:	Elastic constants of matrix phase.
41–43:	Elastic constants of second phase.
45–46:	Magnitude of eigenstrains.
48–52:	Magnitude of applied strains.
54–73:	Initialize stress and strain components.
67–69:	Strain components due to lattice defects.
74–79:	Initialize microstructure.
80–83:	Calculate coefficients of Fourier transformation.
84–87:	Calculate Green's tensor.
89–178:	Time evolution of microstructure
94:	Update total time.
96–105:	Calculate derivative of free energy.
106–112:	Calculate the derivative of elastic energy.
113–119:	Take the values of concentration, derivative of free energy and derivative of elastic energy from real space to Fourier space (forward FFT transformations).
120–133:	Semi-implicit time integration of concentration field at Fourier space (Eq. 5.50).
135:	Take concentration field from Fourier space back to real space (inverse FFT transformation).
137–152:	For small deviations from max and min values, reset the limits.

(continued)

157–176:	If print frequency is reached, write the results to file.
161–170:	Open an output file and print results. These lines are commented out but they can be changed, including the output format.
174:	Write results in vtk format for contour plots to be viewed by using Paraview.
180–183:	Calculate the execution time and print it to screen.

Program

fft_raft_v2.m

This program solves Cahn–Hilliard phase-field equation with semi-implicit Fourier spectral method by taking into account the effects of elastic inhomogeneities and applied stresses based on solution of stress–strain fields with Green’s tensor and Fourier transformations. The time integration is carried out by using semi-implicit time marching scheme. The code is optimized for Matlab/Octave.

The program makes calls to the following functions:

- **prepare_fft.m**
- **free_energ_ch_v2.m**
- **green_tensor.m**
- **solve_elasticity_v2.m**
- **micro_ch_pre.m**
- **write_vtk_grid_values.**

Listing:

```

1 %%%%%%%%%%%%%%%%
2 % % %
3 % SEMI-IMPLICIT SPECTRAL PHASE- %
4 % FIELD CODE %
5 % FOR SOLVING PRECIPITATION UNDER %
6 % STRESS %
7 % (OPTIMIZED FOR MATLAB/OCTAVE) %
8 % %
9 %% get intial wall time:
10 time0=clock();
11 format long;
12 %-- Simulation cell parameters:
13 Nx=256;
14 Ny=256;
15 NxNy =Nx*Ny;
16 dx=1.0;
17 dy=1.0;
18
19 %--- Time integration parameters:
20 nstep = 5000;
21 nprint= 50;
22 dtime = 5.0e-2;
23 ttime = 0.0;
24 coefA = 1.0;
25
26 %--- Material specific Parameters:
27 c0 =0.40;
28 mobility =1.0;
29 grad_coef=0.5;
30
31 %-- elastic constants:
32 cm11 = 1400.0;
33 cm12 = 600.0;
34 cm44 = 400.0;
35
36 %-
37 cp11 = 2.0*cm11;
38 cp12 = 2.0*cm12;
39 cp44 = 2.0*cm44;
40
41 %--eigen strains:
42 ei0= 0.01;
43 ed11=zeros(Nx,Ny);
44 ed22=zeros(Nx,Ny);
45 ed12=zeros(Nx,Ny);
46
47 %-- Applied strains:
48 ea(1)=0.0;
49 ea(2)=0.01;
50 ea(3)=0.0;
51
52 %--Initialize stress & strain
53 %componetes
54 s11=zeros(Nx,Ny);
55 s22=zeros(Nx,Ny);
56
57
58
59
60

```

```

61 s12=zeros(Nx,Ny);           105 delsdck = fft2(delsdc);
62 e11=zeros(Nx,Ny);           106
63 e22=zeros(Nx,Ny);           107
64 e12=zeros(Nx,Ny);           108 %--- Time integration:
65 %--- Prepare microstructure 109
66 iflag = 1;                  110 numer = dtimes*mobility*k2.*;
67 [con] = micro_ch_pre(Nx,Ny,   * (dfdconk + delsdck);
68 c0,iflag);                  111
69 %-- Prepare fft:           112 denom = 1.0 + dtimes*coefA*mobility*
70 [kx,ky,k2,k4] = prepare_fft  grad_coef*k4;
71 (Nx,Ny,dx,dy);             113
72 %---- Greens tensor         114 conk=(conk - numer)./denom;
73 [tmatx] = green_tensor(Nx,Ny,kx, 115
74 ky,cm11,cm12,cm44,...,        116 con=real(ifft2(conk));
75 cp11,cp12,cp44);             117
76 %==                         118 %-- for small deviations:
77 %--- Evolve:                119
78 %==                         120 inrange=(con >= 0.9999);
79 %==                         121 con(inrange) = 0.9999;
80 %==                         122 inrange=(con < 0.00001);
81 %==                         123 con(inrange) = 0.00001;
82 %==                         124
83 %==                         125 %==
84 for istep=1:nstep            126 %--- print results:
85 ttime=ttime+dtimes;          127 %==
86                         128
87 %--- derivative of free energy: 129 if((mod(istep,nprint)==0) ||
88 [dfdcon]=free_energ_ch_v2(Nx, 130 (istep==1))
89 Ny,con);                   131 fprintf('done step: %5d\n',
90 %---derivative of elastic energy: 132 istep);
91 [delsdc ] = solve_elasticity_v2 133 %fname1=sprintf('time_%d.out',
92 (Nx,Ny,tmatx,kx,ky,s11,s22,s12,... istep);
93 e11,e22,e12,ed11,ed22,ed12,... 134 %out1=fopen(fname1,'w');
94 cm11,cm12,cm44,cp11,cp12,cp44,... 135
95 ea,ei0,con);                 136 %for i=1:Nx
96 %--                           137 %for j=1:Ny
97 delsdck = solve_elasticity_v2 138 %fprintf(out1,'%5d %5d %14.6e\n',i,
98 (Nx,Ny,tmatx,kx,ky,s11,s22,s12,... j,con(i,j));
99 e11,e22,e12,ed11,ed22,ed12,... 139 %end
100 cm11,cm12,cm44,cp11,cp12,cp44,... 140 %end
101 ea,ei0,con);                 141
102 %--                           142 %fclose(out1);
103 dfdconk = fft2(dfdcon);      143
104 %--- write vtk file:        144 %--- write vtk file:
105                               145
106                               146 write_vtk_grid_values(Nx,Ny,
107 dx,dy,istep,con);
108

```

```

147
148 end %end if
149
150 end %istep
151
152 %--- calculate compute time:
153
154 compute_time = etime(clock(), time0);
155 fprintf('Compute Time: %10d\n',
           compute_time);
156

```

Line numbers:

10:	Get initial wall clock time beginning of the execution.
13–21:	Simulation cell parameters.
15:	Number of grid points in the x -direction.
16:	Number of grid points in the y -direction.
17:	Total number of grid points in the simulation cell.
19:	The distance between two grid points in the x -direction.
20:	The distance between two grid points in the y -direction.
22–29:	Time integration parameters.
24:	Number of time steps.
25:	Print frequency to write the results to file.
26:	Time increment for numerical integration.
27:	Total time.
30–35:	Material-specific parameters.
32:	Initial concentration.
33:	The value of mobility coefficient.
34:	The value of gradient energy coefficient.
36–44:	Elastic constants.
37–39:	Elastic constants of matrix phase.
41–43:	Elastic constants of second phase.
46:	Magnitude of eigenstrains.
47–49:	Strain components due to lattice defects.
51–55:	Magnitude of applied strains.
57–66:	Initialize stress and strain components.
67–72:	Initialize microstructure.
73–76:	Calculate coefficients of Fourier transformation.
77–80:	Calculate Green's tensor.
82–150:	Time evolution of microstructure
87:	Update total time.
89–92:	Calculate derivative of free energy at all grid points.
93–98:	Calculate the derivative of elastic energy at all grid points.
101–106:	Take the values of concentration, derivative of free energy and derivative of elastic

108–114:	energy from real space to Fourier space (forward FFT transformations).
116:	Semi-implicit time integration of concentration field at Fourier space (Eq. 5.50).
118–124:	Take concentration field from Fourier space back to real space (inverse FFT transformation).
129–148:	For small deviations from max and min values, reset the limits.
133–142:	If print frequency is reached, write the results to file.
146:	Open an output file and print results. These lines are commented out but they can be changed, including the output format.
152–155:	Write results in vtk format for contour plots to be viewed by using Paraview.
	Calculate the execution time and print it to screen.

Function**green_tensor.m**

This function evaluates the value of Green's tensor given by Eqs. 5.40, 5.42, and 5.44 at all grid points of the simulation cell. It is in compact form after some algebraic manipulations and collecting terms together. In here, it is given for 2D case. The 3D version which takes into account the six components of the strains and stresses is given in *Appendix-B* for those who wish to expand their analysis into the 3D.

Variable and array list:

Nx:	Number of grid points in the x -direction.
Ny:	Number of grid points in the y -direction.
cm11:	C11 component of elasticity matrix for matrix material.
cm12:	C12 component of elasticity matrix for matrix material.
cm44:	C44 component of elasticity matrix for matrix material.
cp11:	C11 component of elasticity matrix for second phase.
cp12:	C12 component of elasticity matrix for second phase.
cp44:	C44 component of elasticity matrix for second phase.
kx(Nx):	Fourier vector component in x -direction.
ky(Ny):	Fourier vector component in y -direction.
tmatx(Nx, Ny, 2, 2, 2, 2):	Values of Green's tensor at all grid points.

Listing:

```

1  function [tmatx] = green_tensor(Nx,
2      Ny,kx,ky,cm11,cm12,cm44, ...
3      cp11,cp12,cp44)
4  format long;
5
6  c11 = 0.5*(cm11+cp11);
7  c12 = 0.5*(cm12+cp12);
8  c44 = 0.5*(cm44+cp44);
9
10
11
12  chi=(c11-c12-2.0*c44)/c44;
13
14  for i=1:Nx
15  for j=1:Ny
16
17
18  rr=kx(i)^2+ky(j)^2;
19
20  d0=c11*rr^3+chi*(c11+c12)*rr*
    (kx(i)^2*ky(j)^2);
21
22  if(rr < 1.0e-8)
23  d0=1.0;
24 end
25
26  omeg11(i,j)=(c44*rr^2+(c11-c44)
    *rr*ky(j)^2)/(c44*d0);
27
28  omeg22(i,j)=(c44*rr^2+(c11-c44)
    *rr*kx(i)^2)/(c44*d0);
29
30  omeg12(i,j)=-(c12+c44)*kx(i)
    *ky(j)*rr/(c44*d0);
31
32 end
33 end
34
35
36
37 for i=1:Nx
38 for j=1:Ny
39
40
41 %-- Greens tensor:
42
43 gmatx(1,1)=omeg11(i,j);
44 gmatx(1,2)=omeg12(i,j);
45 gmatx(2,1)=omeg12(i,j);
46 gmatx(2,2)=omeg22(i,j);
47
48 %% position vector:
49
50 dvect(1)=kx(i);
51 dvect(2)=ky(j);
52
53 %% Green operator:
54
55 for kk=1:2
56 for ll=1:2
57 for ii=1:2
58 for jj=1:2
59
60     tmatx(i,j,kk,ll,ii,jj)=0.25*
        (gmatx(ll,ii)*dvect(jj) ...
        *dvect(kk)+gmatx(kk,ii)*dvect
        (jj) ...
        *dvect(ll)+gmatx(ll,jj)*dvect
        (ii) ...
        *dvect(kk)+gmatx(kk,jj)*dvect
        (ii)*dvect(ll));
61
62
63
64
65 end
66 end
67 end
68 end
69
70
71 end %j
72 end %i
73
74 end %endfunction

```

Line numbers:

6–8:	Elastic constants of the homogenous material, C^0 , which are arithmetic averages of the elastic constants of the phases.
14–72:	Evaluate Green's tensor by using Eqs. 5.40, 5.42, and 5.44 for all grid points. The form given in here is after extensive algebraic manipulations and collecting terms together.

Function

solve_elasticity_v1.m

This function evaluates the derivative of elastic energy with respect to concentration. First, stress and strain values are solved with the iterative algorithm described earlier, then the derivative

of elastic energy is evaluated for all grid points. The function is longhand format and not optimized for Matlab/Octave.

Variable and array list:

Nx:	Number of grid points in the x -direction.
Ny:	Number of grid points in the y -direction.
cm11:	C11 component of elasticity matrix for matrix material.
cm12:	C12 component of elasticity matrix for matrix material.
cm44:	C44 component of elasticity matrix for matrix material.
cp11:	C11 component of elasticity matrix for second phase.
cp12:	C12 component of elasticity matrix for second phase.
cp44:	C44 component of elasticity matrix for second phase.
ed11:	Strain component of lattice defects.
ed22:	Strain component of lattice defects.
ed12:	Strain component of lattice defects.
ei0:	Magnitude of eigenstrains.
ea(3):	Applied strains.
con(Nx,Ny):	Concentration.
s11(Nx,Ny):	Component of stress.
s22(Nx,Ny):	Component of stress.
s12(Nx,Ny):	Component of stress.
e11(Nx,Ny):	Component of strain.
e22(Nx,Ny):	Component of strain.
e12(Nx,Ny):	Component of strain.
delsdc(Nx,Ny):	Functional derivative of elastic energy.
tmatx(Nx, Ny,2,2,2,2):	Green's tensor.

Listing:

```

1  function [delsdc] = solve_
   elasticity_v1(Nx,Ny,tmatx,
   kx,ky, ...
2   s11,s22,s12,e11,e22,e12, ...
3   ed11,ed22,ed12,cm11,cm12,cm44, ...
4   cp11,cp12,cp44,ea,ei0,con)
5
6  format long;
7
8  niter=10;
9  tolerance=0.001;
10
11 for i=1:Nx
12 for j=1:Ny
13
14 %--- eigenstrains:
15
16 ei11(i,j) = ei0*con(i,j);
17 ei22(i,j) = ei0*con(i,j);
18 ei33(i,j) = ei0*con(i,j);
19 ei12(i,j) = 0.0*con(i,j);
20
21 % calculate effective elastic
22 % constants
23 % using Vergards law
24 c11(i,j) = con(i,j)*cp11 +(1.0-con
25 (i,j))*cm11;
26 c12(i,j) = con(i,j)*cp12 +(1.0-con
27 (i,j))*cm12;
28 c44(i,j) = con(i,j)*cp44 +(1.0-con
29 (i,j))*cm44;
30
31 end
32 end
33
34 for iter=1:niter
35
36 %--- take the stresses & strains to
37 Fourier space
38 e11k = fft2(e11);
39 e22k = fft2(e22);
40 e12k = fft2(e12);
41
42 %
43 s11k = fft2(s11);
44 s22k = fft2(s22);
45 s12k = fft2(s12);
46
47 %
48 %-- form the stress & strain tensor:
49 for i=1:Nx
50 for j=1:Ny
51
52 smatx(i,j,1,1)=s11k(i,j);
53 smatx(i,j,1,2)=s12k(i,j);
54 smatx(i,j,2,1)=s12k(i,j);
55 smatx(i,j,2,2)=s22k(i,j);
56 %
57 ematx(i,j,1,1)=e11k(i,j);

```

```

58 ematx(i,j,1,2)=e12k(i,j);           +e11(i,j)-ei11(i,j)-ed11(i,j)+...
59 ematx(i,j,2,1)=e12k(i,j);           107      c12(i,j)*(ea(2)+e22(i,j)-ei22
60 ematx(i,j,2,2)=e22k(i,j);           (i,j)-ed22(i,j));
61
62 end
63 end
64 %% Green operator:
65
66 for i=1:Nx
67 for j=1:Ny
68
69 for ii=1:2
70 for jj=1:2
71 for kk=1:2
72 for ll=1:2
73     ematx(i,j,ii,jj)=ematx(i,j,ii,
74         jj)-tmatx(i,j,ii,jj,kk,ll) ...
75     .*smatx(i,j,kk,ll);
76 end
77 end
78 end
79
80 end %j
81 end %i
82 %---
83
84 for i=1:Nx
85 for j=1:Ny
86
87 e11k(i,j)=ematx(i,j,1,1);
88 e22k(i,j)=ematx(i,j,2,2);
89 e12k(i,j)=ematx(i,j,1,2);
90
91 end
92 end
93
94 %---
95 % From Fourier space to real space:
96
97 e11 = real(ifft2(e11k));
98 e22 = real(ifft2(e22k));
99 e12 = real(ifft2(e12k));
100
101 %--- Calculate stresses:
102
103 for i=1:Nx
104 for j=1:Ny
105
106 s11(i,j)=c11(i,j)*(ea(1)
107
108
109 s22(i,j)=c11(i,j)*(ea(2)+e22(i,j)
110 -ei22(i,j)-ed22(i,j))+...
111      c12(i,j)*(ea(1)+e11(i,j)-ei11
112 (i,j)-ed11(i,j));
113
114 end
115 end
116 %---check convergence:
117
118 for i=1:Nx
119 for j=1:Ny
120 sum_stres(i,j)=0.0;
121 end
122 end
123
124 for i=1:Nx
125 for j=1:Nx
126 sum_stres(i,j) = s11(i,j)+s22(i,j)
127 +s12(i,j);
128 end
129
130 normF = norm(sum_stres,2);
131 if(iter ~= 1)
132 conver= abs((normF-old_norm) /
133 old_norm);
134 if(conver <= tolerance)
135 break
136 end
137 old_norm=normF;
138 %
139
140 end %iter
141
142 %--- strain energy:
143
144 % sum strain components:
145
146 for i=1:Nx
147 for j=1:Ny
148
149 et11=ea(1)+e11(i,j)-ei11(i,j)

```

```

-ed11(i,j);
150 et22 =ea(2)+e22(i,j)-ei22(i,j)
-ed22(i,j);
151 et12 =ea(3)+e12(i,j)-ei12(i,j)
-ed12(i,j);
152
153
154 delsdc(i,j) = 0.5*(et11*((cp12
-cm12)*et22+(cp11-cm11)*et11
-c12(i,j)*ei0 - ...
155     c11(i,j)*ei0)-ei0*(c12(i,j)
*et22+c11(i,j)*et11) + ...
156     ((cp11-cm11)*et22+(cp12
-cm12)*et11-c12(i,j) * ...
157     ei0-c11(i,j)*ei0)*et22-ei0*
(c11(i,j)*et22+c12(i,j)
*et11) ...
158     + 2.0*(cp44-cm44)*et12.^2-4.0
*ei0*c44(i,j)*et12);
159 end
160 end
161
162 end %endfunction
163

```

Line numbers:

8:	Maximum number of iteration steps.
9:	Tolerance value of convergence tests.
14–20:	Calculate the eigenstrains.
21–27:	Calculate the effective elastic constants at the grid points based on the composition and using Vegard's law.
34–140:	Solve stress and strain field with iterative algorithm given in the text.
36–46:	Take stress and strain components from real space to Fourier space (forward FFT transformations). Step-a.
48–63:	Form stress and strain tensors to be used in Eq. 5.46, Step-b.
66–78:	Calculate strain tensor, Eq. 5.46, Step-b.
84–92:	Rearrange strain components using symmetry of strain tensor.
95–100:	Take strain components from Fourier space back to real space (inverse FFT transformations), Step-c.
103–115:	Calculate the stress components using linear elasticity. Step-d.
116–138:	Check convergence. Step-e.
142–160:	Calculate functional derivative of elastic energy.
149–151:	Calculate strain components.
154–158:	Functional derivative of the elastic energy with respect to composition.

Function**solve_elasticity_v2.m**

This function evaluates the derivative of elastic energy with respect to concentration. First, stress and strain values are solved with the iterative algorithm described earlier, then derivative of elastic energy is evaluated for all grid points. The function is optimized for Matlab/Octave.

Variable and array list:

Nx:	Number of grid points in the x-direction.
<b b="" ny:<="">	Number of grid points in the y-direction.
<b b="" cm11:<="">	C11 component of elasticity matrix for matrix material.
<b b="" cm12:<="">	C12 component of elasticity matrix for matrix material.
<b b="" cm44:<="">	C44 component of elasticity matrix for matrix material.
<b b="" cp11:<="">	C11 component of elasticity matrix for second phase.
<b b="" cp12:<="">	C12 component of elasticity matrix for second phase.
<b b="" cp44:<="">	C44 component of elasticity matrix for second phase.
<b b="" ed11:<="">	Strain component of lattice defects.
<b b="" ed22:<="">	Strain component of lattice defects.
<b b="" ed12:<="">	Strain component of lattice defects.
<b b="" ei0:<="">	Magnitude of eigenstrains.
<b b="" ea(3):<="">	Applied strains.
<b b="" con(nx,ny):<="">	Concentration.
<b b="" s11(nx,ny):<="">	Component of stress.
<b b="" s22(nx,ny):<="">	Component of stress.
<b b="" s12(nx,ny):<="">	Component of stress.
<b b="" e11(nx,ny):<="">	Component of strain.
<b b="" e22(nx,ny):<="">	Component of strain.
<b b="" e12(nx,ny):<="">	Component of strain.
<b delsdc(nx,<br=""> Ny):	Functional derivative of elastic energy.
<b tmatx(nx,<br=""> Ny,2,2,2,2):	Green's tensor.

Listing:

```

1 function [delsdc ] = solve_
elasticity_v2(Nx,Ny,tmatx,
kx,ky, ...
2 s11,s22,s12,e11,e22,e12, ...
3 ed11,ed22,ed12,cm11,cm12,cm44, ...
4 cp11,cp12,cp44,ea,ei0,con)
5

```

```

6  format long;
7
8  niter =10;
9  tolerance=0.001;
10
11 %--- eigenstrains:
12
13 ei11 = ei0*con;
14 ei22 = ei0*con;
15 ei33 = ei0*con;
16 ei12 = 0.0*con;
17
18
19 % calculate effective elastic
20 % constants
21 % using Vergards law
22 c11 = con*cp11 +(1.0-con)*cm11;
23 c12 = con*cp12 +(1.0-con)*cm12;
24 c44 = con*cp44 +(1.0-con)*cm44;
25
26
27 for iter=1:niter
28
29 %--- take the stresses & strains to
30 % Fourier space
31 e11k = fft2(e11);
32 e22k = fft2(e22);
33 e12k = fft2(e12);
34
35 %
36 s11k = fft2(s11);
37 s22k = fft2(s22);
38 s12k = fft2(s12);
39
40
41 %-- form the stress & strain tensor:
42
43 smatx( :, :,1,1)=s11k;
44 smatx( :, :,1,2)=s12k;
45 smatx( :, :,2,1)=s12k;
46 smatx( :, :,2,2)=s22k;
47 %
48 ematx( :, :,1,1)=e11k;
49 ematx( :, :,1,2)=e12k;
50 ematx( :, :,2,1)=e12k;
51 ematx( :, :,2,2)=e22k;
52
53 %-- Green operator:
54
55 for ii=1:2
56 for jj=1:2
57 for kk=1:2
58 for ll=1:2
59     ematx( :, :,ii,jj)=ematx( :, :,
60                                ii,jj)-tmatx( :, :,ii,jj,
61                                kk,ll) ...
62                                .*smatx( :, :,kk,ll);
63 end
64 end
65
66 %---
67
68 e11k=ematx( :, :,1,1);
69 e22k=ematx( :, :,2,2);
70 e12k=ematx( :, :,1,2);
71
72
73 %---
74 % From Fourier space to real space:
75
76 e11 = real(ifft2(e11k));
77 e22 = real(ifft2(e22k));
78 e12 = real(ifft2(e12k));
79
80 %--- Calculate stresses:
81
82
83 s11=c11.* (ea(1)+e11-ei11-ed11)
84 +c12.* (ea(2)+e22-ei22-ed22);
85 s22=c11.* (ea(2)+e22-ei22-ed22)
86 +c12.* (ea(1)+e11-ei11-ed11);
87 s12=2.0*c44.* (ea(3)+e12-ei12
88 -ed12);
89
90 %---check convergence:
91 sum_stres = s11+s22+s12;
92 normF = norm(sum_stres,2);
93 if(iter ~= 1)
94 conver= abs((normF-old_norm)/
95 old_norm);
96 if(conver <= tolerance)
97 break
98 end
99 end %iter

```

```

100
101 %--- strain energy:
102
103 % sum strain components:
104
105 et11 =ea(1)+e11-ei11-ed11;
106 et22 =ea(2)+e22-ei22-ed22;
107 et12 =ea(3)+e12-ei12-ed12;
108
109
110 delsdc = 0.5*(et11.*((cp12-cm12)
*et22+(cp11-cm11)*et11-c12*ei0
-c11*ei0)- ei0*(c12.*et22...
111 +c11.*et11) + ((cp11-cm11)*et22
+(cp12-cm12)*et11-c12*ei0
-c11*ei0).*et22...
112 -ei0*(c11.*et22+c12.*et11)
+ 2.0*(cp44-cm44)*et12.^2-4.0
*ei0*c44.*et12);
113
114
115 end %endfunction
116

```

Line numbers:

8:	Maximum number of iteration steps.
9:	Tolerance value of convergence tests.
13–16:	Calculate the eigenstrains.
19–25:	Calculate the effective elastic constants at the grid points based on the composition and using Vegard's law.
27–99:	Solve stress and strain field with iterative algorithm given in the text.
29–39:	Take stress and strain components from real space to Fourier space (forward FFT transformations). Step-a.
41–51:	Form stress and strain tensors to be used in Eq. 5.46, Step-b.
55–65:	Calculate strain tensor, Eq. 5.46, Step-b.
68–70:	Rearrange strain components using symmetry of strain tensor.
74–78:	Take strain components from Fourier space back to real space (inverse FFT transformations), Step-c.
80–86:	Calculate the stress components using linear elasticity, Step-d.
87–97:	Check convergence, Step-e.
101–114:	Calculate functional derivative of elastic energy.
105–107:	Calculate strain components.
110–112:	Functional derivative of the elastic energy with respect to composition.

References

- Tien JK, Copley SM (1971) The effect of uniaxial stress on the periodic morphology of coherent gamma prime precipitates in nickel-base superalloys. *Phys Rev B* 43:13649
- Pollock TM, Argon AS (1994) Directional coarsening in nickel-base single crystals with high volume fractions of coherent precipitates. *Acta Metall Mater* 42:1859
- Veron M, Brechet Y, Louchet F (1996) Strain induced directional coarsening in Ni based alloys. *Scr Mater* 34:1883
- Carpenter GJC (1973) The dilatational misfit of zirconium hydrides precipitated in zirconium. *J Nucl Mater* 48:264
- Zanellato O, Preuss M, Buffiere JY, Ribeiro F, Steuwer A, Desquines J, Andrieux J, Krebs B (2012) Synchrotron diffraction study of dissolution and precipitation kinetics of hydrides in Zircalloy-4. *J Nucl Mater* 420:537
- Sridhar N, Richman JM, Srolovitz DJ (1997) Microstructural stability of stressed lammellar and fiber composites. *Acta Mater* 45:2715
- Chang JC, Allen SM (1991) Elastic energy changes accompanying gamma prime rafting in nickel-base superalloys. *J Mater Res* 6:1843
- Nabarro FRN, Cress CM, Kotschy P (1996) The thermodynamic driving force for rafting in superalloys. *Acta Mater* 44:3189
- Lee JK (1997) Studying stress-induced morphological evolution with the discrete atom method. *JOM* 49:37
- Gupta H, Weinkamer R, Fratzl P, Lebowitz JL (2001) Microscopic computer simulations of directional coarsening in face-centered cubic alloys. *Acta Mater* 49:53
- Dy L, Chen LQ (1997) Computer simulation of morphological evolution and rafting of gamma prime particles in Ni-base superalloys under applied stress. *Scr Mater* 37:1271
- Dy L, Chen LQ (1999) Shape evolution and splitting of coherent particles under applied stresses. *Acta Mater* 47:247
- Zhu J, Chen LQ, Shen J (2001) Morphological evolution during phase separation and coarsening with strong inhomogeneous elasticity. *Model Simul Mater Sci Eng* 9:499
- Leo PH, Lowengrub JS, Nie Q (2001) On elastically induced splitting instability. *Acta Mater* 49:2761
- Gururajan MP, Abinandanan TA (2007) Phase-field study of precipitate rafting under a uniaxial stress. *Acta Mater* 55:5015
- Moulinec H, Suquet P (1994) A fast numerical method for computing the linear and nonlinear properties of composites. *Comptes Renud de l' Academic des Sciences, Paris II* 318:1417

17. Moulinec H, Suquet P (1998) A numerical method for computing the overall response of nonlinear composites with complex microstructures. *Comput Methods Appl Mech Eng* 157:69
18. Micheal JC, Moulinec H, Suquet P (1999) Effective properties of composite materials with periodic microstructure: a computational approach. *Comput Methods Appl Mech Eng* 172:109
19. Michel JC, Moulinec H, Suquet P (2001) A computational scheme for linear and nonlinear composites with arbitrary phase contrast. *Int J Numer Methods Eng* 52:139
20. Jin YM, Wang YU, Khachaturyan AG (2001) Three dimensional phase-field microelasticity theory and modeling of multiple cracks and voids. *Appl Phys Lett* 79:3071
21. Shen Y, Li Y, Li Z, Wan H, Nie P (2009) An improvement on the three dimensional phase-field microelasticity theory for elastically and structurally inhomogeneous solids. *Scr Mater* 60:901
22. Biner SB, Hu SY (2009) Simulation of damage evolution in discontinuously reinforced metal matrix composites: a phase-field model. *Int J Frac* 158:99
23. Lebensohn RA, Rollett AD, Suquet P (2011) Fast Fourier transform-base modeling for the determination of micromechanical fields in polycrystals. *JOM* 63:13
24. Mura T (1987) *Micromechanics of defects in solids*. Martinus Nijhoff, Dordrecht
25. Eshelby JD (1957) The determination of the elastic field of an ellipsoidal inclusion and related problems. *Proc R Soc Lond A* 241:376
26. Schmit I, Gross D (1999) Directional coarsening in Ni-base superalloys: analytical results for an elasticity based model. *Proc R Soc Lond A* 455:3085

diffusion processes and phase transformations in solids. The interaction of solute atoms with the stress field of dislocations was recognized in a study as early as 1949 which showed how the interaction between solute atoms and a dislocation results in solute segregation and depletion, leading to the formation of the so-called “Cottrell atmosphere” [1]. The nucleation of new phases around dislocations and grain boundaries is often observed in experiments. Cahn [2] first studied the nucleation of a second-phase precipitate around a dislocation with a theoretical model that shows that additional driving forces develop for nucleation around the dislocation than the bulk. The solute–dislocation interactions have been studied at atomistic level [3–7] as well as phase-field modeling approaches at the mesoscale [8–13] with varying degree of sophistications.

The Fe–Cr system is the basis for a large class of important engineering materials, such as stainless steels. They exhibit usually excellent corrosion resistance and mechanical properties. However, they are susceptible to the so-called “475 °C embrittlement.” This embrittlement, large loss of fracture toughness, originates from a phase separation reaction, where the ferrite decomposes into Fe-rich α and Cr-rich α' phases. This phase separation is studied in detail in both experiments and simulations in a series of articles in [14–16].

5.8 Case Study-X

The role of lattice defects on spinodal decomposition of a Fe–Cr alloy

Objectives:

The objective of this case study is to develop a semi-implicit Fourier spectral phase-field algorithm in which, in addition to elastic inhomogeneities, the lattice defects such as dislocation and grain boundaries are incorporated into the formulism.

5.8.1 Background

The lattice defects, such as dislocations and grain boundaries, play an important role in

5.8.2 Phase-Field Model

The phase-field model closely follows that is given in *Case Study-IX* in which the modified Cahn–Hilliard equation was expressed as:

$$\frac{\partial c}{\partial t} = \nabla^2 \cdot M \left(\frac{\delta F^{CH}}{\delta c} + \frac{\delta F^{EL}}{\delta c} \right) \quad (5.51)$$

in which, F^{CH} , the chemical energy, is taken as

$$F^{CH} = \int_V \left[\frac{\delta f(c)}{\delta c} + \frac{1}{2} \kappa |\nabla c|^2 \right] dv \quad (5.52)$$

The $f(c)$ for Fe–Cr system, based on the regular solution approximation, is given in [17] as:

$$f(c) = \frac{1}{V_m} \left\{ (1-c)G_{\text{Fe}}^0 + cG_{\text{Cr}}^0 + (20500 - 9.68T)c(1-c) + \right\} \quad (5.53)$$

where c is the Cr concentration, V_M is the molar volume, G_{Fe}^0 and G_{Cr}^0 are the standard molar Gibbs energy of pure Fe and Cr, respectively, and they are taken as zero, as reference energy level. R is the gas constant and T is the temperature.

The contribution of dislocations to elastic energy, F^{EL} , is taken into account through their eigenstrains which is expressed as:

$$\varepsilon_{ij}^0 = b \otimes n = (b_i n_j + b_j n_i)/2d \quad (5.54)$$

where b and n are the Burgers vector and normal vector to the slip plane, respectively, and d is the inter-planar distance of the slip planes.

5.8.3 Numerical Implementation

After, solving stress and strain values, by considering the elastic inhomogeneity owing to the differences in the elastic properties between Fe and Cr phases, misfit strains of Cr precipitates and the eigenstrains due to presence of dislocations, the solution of Eq. 5.51 follows the same steps given in *Case Study-IX*. Therefore, they are not repeated in here again.

5.8.4 Results and Discussion

Before, the simulation of phase separation in a Fe–Cr alloy, the accuracy of the algorithm in calculating the stress fields of dislocations is considered first. For this purpose a dislocation dipole (composed of two edge dislocations) with Burger's vector $b = a_0/2[111]$ was introduced on $(1\bar{1}0)$ slip plane, where a_0 is the lattice constant. The lattice rotated to a coordinate system with x -axis $[1\bar{1}0]$, y -axis $[111]$ and z -axis $[\bar{1}12]$. For this case, the only nonzero eigenstrain terms (Eq. 5.54) is the shear term. The simulation cell had $N_x = N_y = 256$ grid points with grid spacing $dx = dy = 0.5$. The width of the dipole was 60 grid spacing and the

elastic properties were assumed to be isotropic. The resulting stress fields are shown in Fig. 5.12. The results agree very well with the analytical solutions, based on the isotropic linear elasticity,

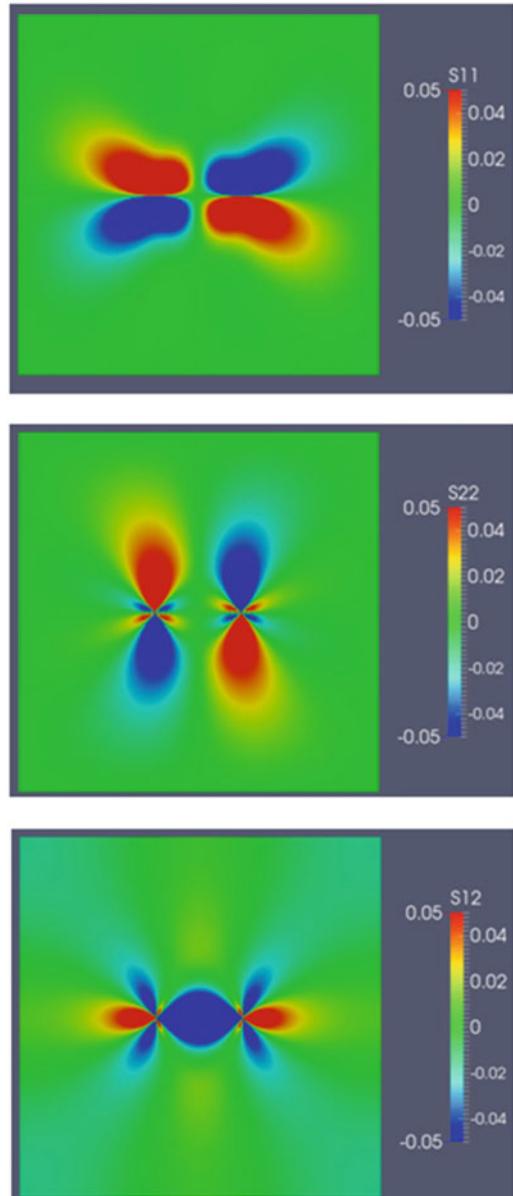


Fig. 5.12 Calculated stress fields of edge dislocation dipole with eigenstrain approach

presented in [18]. As can be seen, while the magnitude of the stresses remains the same, there is a change of sign between the positive and negative edge dislocations.

Next, the precipitation behavior of a 20%Cr containing Fe–Cr alloy at annealing temperature 535 °K is studied with a simulation cell having number of grid points $N_x = N_y = 128$ and grid spacing $dx = dy = 1.0$. The parameters used in these simulations are summarized in Table 5.3. As can be seen, there is an elastic inhomogeneity through the differences in their elastic properties of Fe and Cr phases. In addition, the both phases

exhibit cubic anisotropy, with the anisotropy ratios $\text{Fe} \approx 3.6$ and $\text{Cr} \approx 0.71$. The misfit strains for Cr phase are dilatational and taken as $\varepsilon_{ij}^0 = 0.006\delta_{ij}$, where δ_{ij} is the Kronecker delta function. The quantities having the dimension of distance were normalized with the magnitude of the Burger's vector, the quantities having the dimension of energy were normalized with RT , and the time t was normalized with $M/(dx)^2$. The initial concentration was modulated by setting the noise term to 0.001 in function *micro_ch_pre.m*. The evolution of microstructure due to phase separation is shown in Fig. 5.13.

Table 5.3 The values of the parameters used in the simulations

C_{11}^{Fe}	C_{12}^{Fe}	C_{44}^{Fe}	C_{11}^{Cr}	C_{12}^{Cr}	C_{44}^{Cr}	M	κ	T
$233.1e^3$	$135.44e^3$	$178.3e^3$	$350.0e^3$	$67.8e^3$	$100.8e^3$	1	0.5	535 °K

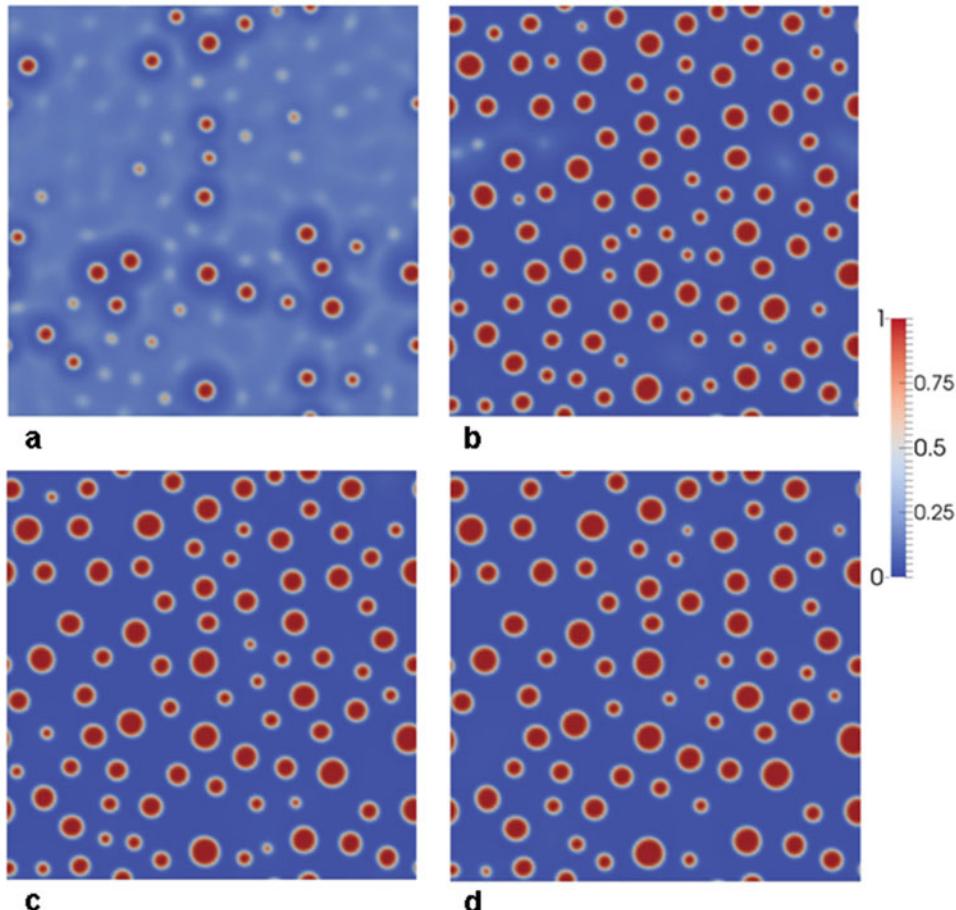


Fig. 5.13 Phase separation of 20%Cr Fe–Cr alloy at 535 °K without lattice defects. Nondimensional times are (a) 62.5, (b) 75.0, (c) 87.5, and (d) 100.0

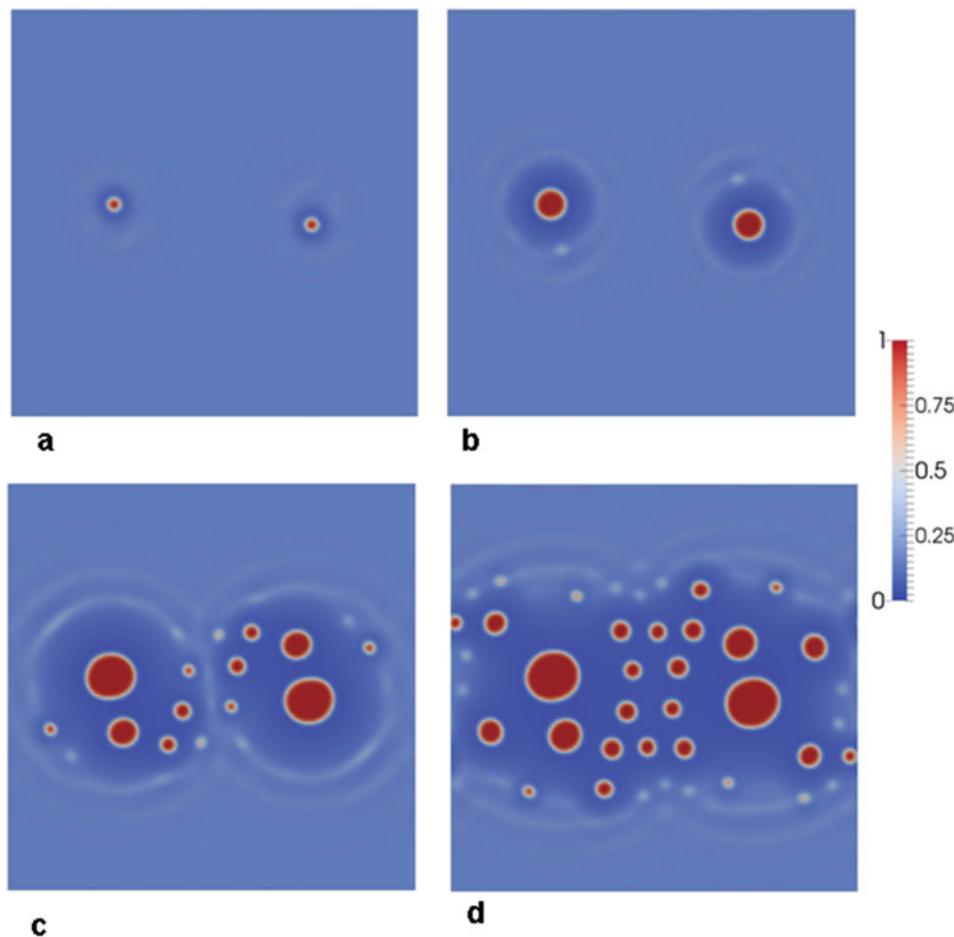


Fig. 5.14 The precipitation behavior at the dislocation dipole. Nondimensional times are: (a) 50.0, (b) 62.5, (c) 87.5, and (d) 100.0

As seen in earlier case studies for spinodal decomposition, the microstructure is relatively fine and contains a large number of precipitates initially. Owing to use of very small noise term, the start of precipitation required almost half of the simulation time. As aging time progresses, coarsening of the second phase through migration of the phase boundaries, dissolution, merging, and breakup can be seen from the figure.

In the second set of simulations, the dislocation dipole was introduced into the center of the simulation cell. The character of the dislocation dipole and the simulation parameters were the same as given earlier. However, there was no modulation of the concentration, the noise term in function `micro_ch_pre.m` was set to be zero for

this case. Figure 5.14 shows the time evolution of the microstructure for this case. Even there was no noise term to promote the nucleation; two precipitates nucleate, one above the dislocation and one below, owing to the elastic strain energy induced to the system. Since Cr atoms are oversized, the precipitation follows the stress field of dislocation dipole shown in Fig. 5.12 and takes place at the regions where the positive hydrostatic stress is the highest. In addition, the first noticeable appearance of the precipitates was earlier than the one seen in previous simulation. The initial depletion of the Cr concentration around the dislocation can also be clearly seen in the figure. With increasing aging time and with the flux of concentration driven by the stress field, as

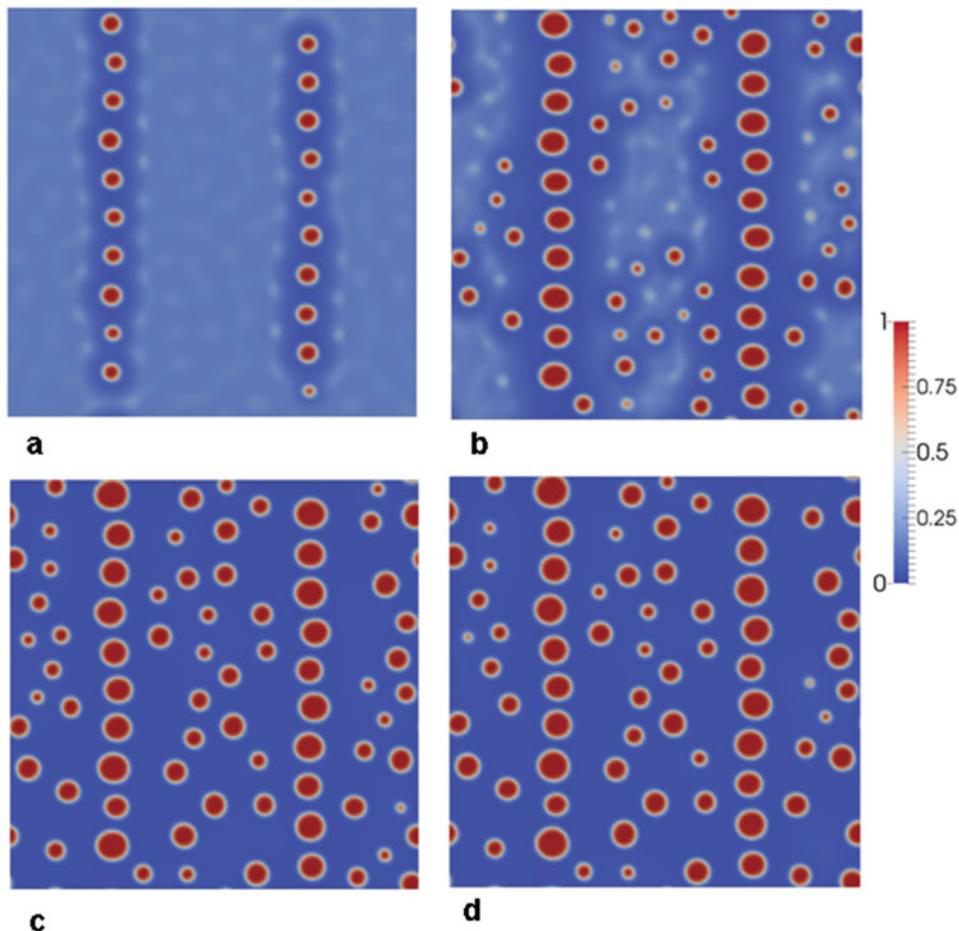


Fig. 5.15 Precipitation behavior in a bicrystal. The grain boundaries are modeled with array of edge dislocations. Nondimensional times are: (a) 50, (b) 62.5, (c) 87.5, and (d) 100.0

first nucleated precipitates coarsen, the other also forms. At this stage, the elastic inhomogeneity induced by the coarsening of precipitates also becomes a contributing factor.

Next, the precipitation behavior due the presence of low-angle grain boundaries was elucidated. The representation of grain boundaries with array of edge dislocations was first introduced by Read and Schocley [19] for low-angle grain boundaries and later was modified to include the high-angle boundaries in [20]. To represent a bicrystal microstructure, array of edge dislocation dipoles, 60 grid-spacing

in width and 12 grid-spacing apart in y -direction was introduced into the system. The initial concentration field was again modulated with noise value 0.001 as in the first case. The evolution of the microstructure is shown in Fig. 5.15 for this case. Again, owing to the presence of dislocations, the first noticeable appearance of the precipitates was earlier, and the nucleation of row precipitates at the grain boundaries can be discerned from the figure. As the aging proceeds, the evolution of denuded zones, similar to that observed in Fig. 5.14, also can be clearly seen from the figure.

5.8.5 Source Codes

Program

fft_FeCr_v1.m

This program solves conserved phase-field equation with Fourier spectral method by taking into account the effects of elastic inhomogeneities and lattice defects, based on solution of stress-strain fields with Green's tensor and Fourier transformations. The time integration is carried out by using semi-implicit time marching scheme. The code is in longhand format and is not optimized.

The program makes calls to the following functions:

- **dislo_strain.m**
- **FeCr_chem_poten_v1.m**
- **green_tensor.m**
- **prepare_fft.m**
- **solve_elasticity_v1.m**
- **micro_ch_pre.m**
- **write_vtk_grid_values.m**

Listing:

```

1  %%%%%%%%%%%%%%%%
2  %
3  % SEMI-IMPLICIT SPECTRAL PHASE- %
4  % FIELD CODE %
5  % FOR SOLVING PRECIPITATION IN %
6  % FeCr ALLOY %
7  %
8  %% get intial wall time:
9  time0=clock();
10 format long;
11
12 %-- Simulation cell parameters:
13
14 Nx=128;
15 Ny=128;
16 NxNy =Nx*Ny;
17
18 dx=1.0;

19 dy=1.0;
20
21 %--- Time integration parameters:
22
23 nstep = 10000;
24 nprint= 50;
25 dtime = 1.0e-2;
26 ttime = 0.0;
27 coefA = 2.0;
28
29 %--- Material specific Parameters:
30
31 c0 =0.20;
32 mobility =1.0;
33 grad_coef=0.5;
34 temp = 535.0;
35 RT = 8.314462*temp;
36
37 %-- elastic constants:
38 cm11 = 233.10e3;
39 cm12 = 135.44e3;
40 cm44 = 178.30e3;
41 %-
42 cp11 = 350.00e3;
43 cp12 = 67.80e3;
44 cp44 = 100.80e3;
45
46 %--eigen strains:
47 ei0= 0.006;
48
49 %-- dislocation eigen strain
50
51 idislo = 1;
52 [ed11,ed22,ed12]=dislo_strain(Nx,
Ny,idislo);
53
54 %-- Applied strains:
55
56 ea(1)=0.0;
57 ea(2)=0.0;
58 ea(3)=0.0;
59
60 %--Initialize stress & strain
componetes
61
62 for i=1:Nx
63 for j=1:Ny
64
65 s11(i,j)=0.0;
66 s22(i,j)=0.0;

```

```

67 s12(i,j)=0.0;           112
68                               113 dfdcr(i,j) = dummy;
69 e11(i,j)=0.0;           114 end
70 e22(i,j)=0.0;           115 end
71 e12(i,j)=0.0;           116
72                               117 %-
73 end                         118
74 end                         119 crk = fft2(cr);
75                               120
76 %--- Prepare microstructure 121 dfdcrk = fft2(dfdcr);
77                               122
78 iflag = 1;                 123 delsdck = fft2(delsdc);
79                               124
80 [cr] = micro_ch_pre(Nx,Ny, 125 %--- Time integration:
c0,iflag);                  126
81                               127
82 %-- Prepare fft:          128 for i=1:Nx
83                               129 for j=1:Ny
84 [kx,ky,k2,k4] = prepare_fft(Nx, 130
Ny,dx,dy);                  131 numer = dtim*mobility*k2(i,j)*
                               (dfdcrk(i,j) + delsdck(i,j));
85                               132
86 %---- Greens tensor        133 denom = 1.0 + dtim*coefA*mobility
87                               134 *grad_coef*k4(i,j);
88 [tmatx] = green_tensor(Nx,Ny,kx,ky, 135 crk(i,j) = (crk(i,j) - numer)./
cm11,cm12,cm44,...           denom;
89     cp11,cp12,cp44);         136
90 %==                         137 end
91 %--- Evolve:               138 end
92 %==                         139
93                               140 %---
94 for istep=1:nstep          141
95                               142 cr = real(ifft2(crk));
96 ttime = ttime + dtim;       143
97                               144 %-- for small deviations:
98                               145
99 %---derivative of elastic energy: 146 for i=1:Nx
100                               147 for j=1:Ny
101 [delsdc ] = solve_elasticity_v1(Nx, 148
Ny,tmatx,kx,ky,s11,s22,s12,...   149 if(cr(i,j) >= 0.9999);
102 e11,e22,e12,ed11,ed22,ed12,... 150 cr(i,j) = 0.9999;
103 cm11,cm12,cm44,cp11,cp12,cp44, 151 end
ea,ei0,cr);                  152 if(cr(i,j) < 0.00001);
104 %--- derivative of free energy: 153 cr(i,j) = 0.00001;
105                               154 end
106 for i=1:Nx                155
107 for j=1:Ny                156 end
108                               157 end
109 delsdc(i,j) = delsdc(i,j)/RT; 158
110                               159 %==
111 [dummy] =FeCr_chem_potent_v1(i,j, 159 %==
cr,temp);

```

```

160 %--- print results:
161 ====
162
163 if((mod(istep,nprint) == 0) ||
164 (istep == 1))
165 fprintf('done step: %d\n',istep);
166
167 %fname1 =sprintf('time_%d.out',
168 istep);
169 %out1 = fopen(fname1,'w');
170 %for i=1:Nx
171 %for j=1:Ny
172 %fprintf(out1,'%5d %5d %14.6e\n',
173 i,j,cr(i,j));
174 %end
175 %end
176 %fclose(out1);
177
178 %--- write vtk file:
179
180 write_vtk_grid_values(Nx,Ny,
181 dx,dy,istep,cr);
182 end %end if
183
184 end %istep
185
186 %--- calculate compute time:
187
188 compute_time = etime(clock(),
189 time0);
190 fprintf('Compute Time: %10d\n',
compute_time);

```

Line numbers:

9:	Get initial wall clock time beginning of the execution.
12–20:	Simulation cell parameters.
14:	Number of grid points in the x -direction.
15:	Number of grid points in the y -direction.
16:	Total number of grid points in the simulation cell.
18:	Grid spacing between two grid points in the x -direction.
19:	Grid spacing between two grid points in the y -direction.

21–28:	Time integration parameters.
23:	Number of time steps.
24:	Print frequency to write the results to file.
25:	Time increment for numerical integration.
26:	Total time.
27:	The value of coefficient A.
29–45:	Material-specific parameters.
31:	Alloy concentration.
32:	Value of the mobility parameter.
33:	Value of gradient energy coefficient.
34:	Value of annealing temperature.
35:	Gas constant x temperature.
37–45:	Elastic constants of phases.
38–40:	Elastic constants of Fe-rich phase.
42–44:	Elastic constants of Cr-rich phase.
47:	The value of eigenstrains for Cr-rich phase.
49–53:	Introduce eigenstrains of dislocations into the simulation cell.
51:	idislo = 1 for dislocation dipole, idislo = 2 for dislocation array.
54–59:	The components of applied strains.
66–75:	Initialize the stress and strain field.
76–81:	Initialize microstructure.
82–85:	Calculate coefficients of Fourier transformation.
86–89:	Calculate the Green's tensor.
91–184:	Time evolution of microstructure.
96:	Update total time.
99–103:	Calculate the derivative of elastic energy.
106–115:	Calculate the derivative of chemical energy.
109:	Normalize the derivative elastic energy with RT .
118–124:	Take the values of concentration, derivative of elastic energy and derivative of chemical energy from real space to Fourier space (forward FFT transformations).
126–135:	Semi-implicit time integration of Cr concentration field at Fourier space.
142:	Take the concentration field from Fourier space back to real space (inverse FFT transformation).
144–157:	For small deviations from max and min values, reset the limits.
160–182:	If print frequency is reached, write the results to file.
167–176:	Open an output file and print the results. These lines are commented out but they can be changed, including the output format.
180:	Write the results in vtk format for contour plots to be viewed by using Paraview.
186–189:	Calculate the execution time and print it to screen.

Program

fft_FeCr_v2.m

This program solves conserved phase-field equation with Fourier spectral method by taking into account the effects of elastic inhomogeneities and lattice defects, based on solution of stress-strain fields with Green's tensor and Fourier transformations. The time integration is carried out by using semi-implicit time marching scheme. The code is optimized for Matlab/Octave.

The program makes calls to the following functions:

- **dislo_strain.m**
- **FeCr_chem_poten_v2.m**
- **green_tensor.m**
- **prepare_fft.m**
- **solve_elasticity_v2.m**
- **micro_ch_pre.m**
- **write_vtk_grid_values.m**

Listing:

```

1    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2    %
3    % SEMI-IMPLICIT SPECTRAL PHASE- %
4    % FIELD CODE %
5    % FOR SOLVING PRECIPITATION IN FeCr %
6    % ALLOY %
7    % (OPTIMIZED FOR MATLAB/OCTAVE) %
8    %
9    %% get intial wall time:
10   time0=clock();
11   format long;
12
13   %% Simulation cell parameters:
14
15   Nx=128;
16   Ny=128;
17   NxNy =Nx*Ny;
18
19   dx=1.0;
20   dy=1.0;

21
22   %--- Time integration parameters:
23
24   nstep = 10000;
25   nprint= 50;
26   dtime=1.0e-2;
27   ttime= 0.0;
28   coefA= 2.0;
29
30   %--- Material specific Parameters:
31
32   c0 =0.20;
33   mobility=1.0;
34   grad_coef=0.5;
35   tempR = 535.0;
36   RT = 8.314462*tempR;

37
38   %% elastic constants:
39   cm11 = 233.10e3;
40   cm12 = 135.44e3;
41   cm44 = 178.30e3;
42   %
43   cp11 = 350.00e3;
44   cp12 = 67.80e3;
45   cp44 = 100.80e3;
46
47   %%eigen strains:
48   ei0= 0.006;
49
50   %% dislocation eigen strain
51
52   idislo = 1;
53   [ed11,ed22,ed12] = dislo_strain(Nx,Ny,
54   idislo);
55
56   %% Applied strains:
57   ea(1)=0.0;
58   ea(2)=0.0;
59   ea(3)=0.0;
60
61   %%Initialize stress & strain componentes
62
63   s11=zeros (Nx,Ny);
64   s22=zeros (Nx,Ny);
65   s12=zeros (Nx,Ny);
66
67   e11=zeros (Nx,Ny);
68   e22=zeros (Nx,Ny);

```

```

69 e12=zeros(Nx,Ny);
70
71 %--- Prepare microstructure
72
73 iflag = 1;
74
75 [cr] = micro_ch_pre(Nx,Ny,c0,iflag);
76
77 %-- Prepare fft:
78
79 [kx,ky,k2,k4] = prepare_fft(Nx,Ny,dx,
dy);
80
81 %---- Greens tensor
82
83 [tmatx] = green_tensor(Nx,Ny,kx,ky,
cm11,cm12,cm44,...,
84 cp11,cp12,cp44);
85 %==
86 %--- Evolve:
87 %==
88
89 for istep=1:nstep
90
91 ttime = ttime + dttime;
92
93 %--- derivative of free energy:
94
95 [dfdcr] = FeCr_chem_potent_v2(cr,
temp);
96
97 %---derivative of elastic energy:
98
99 [delsdc] = solve_elasticity_v2(Nx,Ny,
tmatx,ky,s11,s22,s12,...,
100 e11,e22,e12,ed11,ed22,ed12,...,
101 cm11,cm12,cm44,cp11,cp12,cp44,
ea,ei0,cr);
102 delsdc = delsdc/RT;
103 %--
104
105 crk = fft2(cr);
106
107 dfdcrk = fft2(dfdcr);
108
109 delsdck = fft2(delsdc);
110
111 %--- Time integration:
112
113
114 numer = dttime*mobility*k2.* (dfdcrk
+ delsdck);
115
116 denom = 1.0 + dttime*coefA*mobility*
grad_coef*k4;
117
118 crk = (crk - numer) ./denom;
119
120 cr = real(ifft2(crk));
121
122 %-- for small deviations:
123
124 inrange = (cr >= 0.9999);
125 cr(inrange) = 0.9999;
126 inrange = (cr < 0.00001);
127 cr(inrange) = 0.00001;
128
129 %==
130 %--- print results:
131 %==
132
133 if((mod(istep,nprint) == 0) ||
(istep == 1))
134
135 fprintf('done step: %5d\n',istep);
136
137 %fname1 = sprintf('time_%d.out',
istep);
138 %out1 = fopen(fname1,'w');
139
140 %for i=1:Nx
141 %for j=1:Ny
142 %fprintf(out1,'%5d %5d %14.6e\n',i,
j,cr(i,j));
143 %end
144 %end
145
146 %fclose(out1);
147
148 %--- write vtk file:
149
150 write_vtk_grid_values(Nx,Ny,dx,
dy,istep,cr);
151
152 end%endif
153
154 end%istep
155
156 %--- calculate compute time:
157

```

```

158 compute_time = etime(clock(),time0);
159 fprintf('ComputeTime: %10d\n',
           compute_time);
160
Line numbers:
10:   Get initial wall clock time beginning of the
      execution.
13–21: Simulation cell parameters.
15:   Number of grid points in the  $x$ -direction.
16:   Number of grid points in the  $y$ -direction.
17:   Total number of grid points in the
      simulation cell.
19:   Grid spacing between two grid points in the
       $x$ -direction.
20:   Grid spacing between two grid points in the
       $y$ -direction.
22–29: Time integration parameters.
24:   Number of time steps.
25:   Print frequency to write the results to file.
26:   Time increment for numerical integration.
27:   Total time.
28:   The value of coefficient  $A$ .
30–46: Material-specific parameters.
32:   Alloy concentration.
33:   Value of mobility parameter.
34:   Value of gradient energy coefficient.
35:   Value of annealing temperature.
36:   Gas constant  $x$  temperature.
38–46: Elastic constants of phases.
39–41: Elastic constants of Fe-rich phase.
43–45: Elastic constants of Cr-rich phase.
48:   The value of eigenstrains for Cr-rich phase.
50–54: Introduce eigenstrains of dislocations into
      the simulation cell.
52:   idislo = 1 for dislocation dipole, idislo = 2
      for dislocation array.
55–60: The components of applied strains.
61–70: Initialize the stress and strain field.
71–76: Initialize microstructure.
77–80: Calculate coefficients of Fourier
      transformation.
81–84: Calculate the Green's tensor.
86–154: Time evolution of microstructure.
91:   Update total time.
93–96: Calculate the derivative of chemical energy.
97–101: Calculate the derivative of elastic energy.
102:   Normalize the derivative elastic energy with
       $RT$ .
105–109: Take the values of concentration, derivative
      of elastic energy and derivative of chemical
      energy from real space to Fourier space
      (forward FFT transformations).

```

112–118:	Semi-implicit time integration of Cr concentration field at Fourier space.
120:	Take the concentration field from Fourier space back to real space (inverse FFT transformation).
122–128:	For small deviations from max and min values, reset the limits.
130–152:	If print frequency is reached, write the results to file.
137–146:	Open an output file and print the results. These lines are commented out but they can be changed, including the output format.
150:	Write the results in vtk format for contour plots to be viewed by using Paraview.
156–159:	Calculate the execution time and print it to screen.

Function

dislo_strain.m

This function introduces the eigenstrain distribution of dislocations to the simulation cell. $\text{idislo} = 1$ is for a dipole and $\text{idislo} = 2$ is for dislocation array.

Variable and array list:

Nx:	Number of grid points in the x -direction.
<b b="" ny:<="">	Number of grid points in the y -direction.
<b b="" idislo:<="">	$\text{idislo} = 1$ is for dislocation dipole, and $\text{idislo} = 2$ for dislocation array.
<b b="" ed11(nx,="" ny):<="">	Eigenstrain component of dislocations (see Eq. 5.54).
<b b="" ed22(nx,="" ny):<="">	Eigenstrain component of dislocations.
<b b="" ed12(nx,="" ny):<="">	Eigenstrain component of dislocations.

Listing:

```

1  function [ed11,ed22,ed12] = dislo_
   strain(Nx,Ny,idislo)
2
3  format long;
4
5  ed11 = zeros (Nx,Ny) ;
6  ed22 = zeros (Nx,Ny) ;
7  ed12 = zeros (Nx,Ny) ;
8
9  Ny2 = Ny/2 ;
10

```

```

11 ndipoles=34;
12 ndipolee=94;
13
14 if(idislo == 1)
15
16 for i=1:Nx
17 if( i >=34 && i <= 94)
18 %ed12(i,Ny2-1) = 5.0e-3
19 ed12(i,Ny2) = 5.0e-3;
20
21 end
22 end
23 end %if
24 %
25
26 if(idislo == 2)
27
28 ndis = 11;
29 jj=0;
30
31 for idis=1:ndis;
32
33 jj= jj +12;
34
35 for i=1:Nx
36 for j=1:Ny
37
38 if( i>=34 && i <=94)
39 %if(j == jj-1 || j == jj)
40 if( j == jj)
41 ed12(i,j) = 5.0e-3;
42 end
43 end
44
45 end %j
46 end %i
47
48 end %idis
49 end %if
50
51 end %endfunction

```

Line numbers:

5–7:	Initialize eigenstrain field of dislocations.
11–12:	Begin and end grid points of dislocation dipoles.
14–23:	Introduce eigenstrain values of dislocation dipole into the simulation cell.
26–49:	Introduce eigenstrain values of a dislocation array, 12 grid points apart in the y-direction, into the simulation cell.

Function**FeCr_chem_potent_v1.m**

This function calculates the derivative of the chemical energy with respect to Cr concentration at the current grid point under consideration in the main program.

Variable and array list:

i:	x-index of current grid point.
j:	y-index of current grid point
tempr:	Temperature.
dfdcr:	Derivative of chemical energy at the current grid point.
cr(Nx, Ny):	Cr concentration at the grid points.

Listing:

```

1 function [dfdcr] = FeCr_chem_
potent_v1(i,j,cr,tempr);
2
3 format long;
4 %
5
6 RT=8.314462*tempr;
7
8 dfdcr =(-cr(i,j)*(20500.0-9.68*
tempr)+(1.0-cr(i,j))* ...
(20500.0-9.68*tempr) ...
9 +(log(cr(i,j))-log(1.0-cr(i,j)))
*RT)/RT;
10 end %endfunction

```

Line numbers:

6:	Value of gas constant x temperature.
8–9:	Derivative of chemical energy with respect to Cr at current grid point. Note that it is normalized with the value of <i>RT</i> .

Function**FeCr_chem_potent_v2.m**

This function calculates the derivative of the chemical energy with respect to Cr concentration at all grid points.

Variable and array list:

tempr:	Temperature.
cr(Nx,Ny):	Cr concentration at the grid points.
dfdcr(Nx, Ny):	Derivative of chemical energy at all grid points of the simulation cell.

Listing:

```

1 function [dfdcr] = FeCr_chem_
potent_v2(cr,tempr);
2
3 format long;
4
5 %
6
7 RT=8.314462*tempr;
8
9 dfdcr =(-cr.* (20500.0-9.68*tempr)
+ (1.0-cr)*(20500.0-9.68*tempr) ...
10 +(log(cr)-log(1.0-cr))*RT)/RT;
11 end %endfunction
12

```

Line numbers:

7:	The value of gas constant x temperature.
9–10:	Derivative of chemical energy with respect to Cr concentration at all grid points. Note that they are normalized with the value of RT .

References

- Cottrell AH, Bilby B (1949) Dislocation theory of yielding and strain aging of iron. Proc Phys Soc, London 62:49
- Cahn JW (1957) Nucleation on dislocations. Acta Mater 5:169
- Hin C, Brechet Y, Mangis P, Soison F (2008) Kinetics of heterogeneous dislocation precipitation of NbC in alpha-iron. Acta Mater 56:5535
- Wang Y, Srolovitz DJ, Rickman JM, LeSar R (2000) Dislocation motion in the presence of diffusing solutes: a computer simulation. Acta Mater 48:2163
- Chen Q, Liu XY, Biner SB (2008) Solute and dislocation junction interactions. Acta Mater 56:2937
- Liu XY, Biner SB (2008) Hydrogen and self-interstitial interactions with edge dislocation in Ni, atomistic and elasticity comparisons. Model Simul Mater Sci Eng 16:045002
- Zhang Y, Millett PC, Tonks MR, Bai XM, Biner SB (2015) Preferential Cu precipitation at extended defects in bcc Fe: an atomistic study. Comput Mater Sci 101:81
- Hu SY, Chen LQ (2001) Solute segregation and coherent nucleation and growth near a dislocation-A phase-field model integrating defect and phase microstructures. Acta Mater 49:463
- Leonard F, Desai RC (1998) Spinodal decomposition and dislocation lines in thin films and bulk materials. Phys Rev B 58:8277
- Ni Y, He LH (2003) Spontaneous ordering of composition pattern in epitaxial monolayer by subsurface dislocation array. Thin Solid Films 440:285
- Hu SY, Chen LQ (2004) Spinodal decomposition in a film with periodically distributed interfacial dislocations. Acta Mater 52:3069
- Li YS, Li SX, Zhang TY (2009) Effect of dislocations on spinodal decomposition in Fe-Cr alloys. J Nucl Mater 295:120
- Biner SB, Weifeng R, Yonfeng Z (2016) The stability of precipitates and the role of lattice defects in Fe-1at%Cu-1at%Ni-1At%Mn alloy: a phase-field model study. J Nucl Mater 468:9
- Miller MK, Hyde JM, Hetherington MG, Cerezo A, Smith GDW, Elliott CM (1995) Spinodal decomposition in Fe-Cr alloys: experimental study at the atomic level and comparison with computer models-I introduction and methodology. Acta Mater 43:3385
- Hyde JM, Miller MK, Hetherington MG, Cerezo A, Smith GDW, Elliott CM (1995) Spinodal decomposition in Fe-Cr alloys: experimental study at the atomic level and comparison with computer models-II Development of domain size and composition amplitude. Acta Mater 43:3403
- Hyde JM, Miller MK, Hetherington MG, Cerezo A, Smith GDW, Elliott CM (1995) Spinodal decomposition in Fe-Cr alloys: experimental study at the atomic level and comparison with computer models-II Development of morphology. Acta Mater 43:3415
- Andserson JO, Sudman B (1987) Thermodynamic properties of Cr-Fe system. Calphad 11:89
- Hirth JP, Lothe J (1968) Theory of dislocations. McGraw-Hill, New York
- Read WT, Shockley W (1950) Dislocation models of crystal boundaries. Phys Rev 78:275
- Wolf D (1989) A Read-Schokley model for high-angle grain boundaries. Scr Metall 23:1713

6.1 Introduction

The finite element method, FEM, and sometimes also called finite element analysis, FEA, was originally developed in the aircraft industry in 1960s [1, 2]. Therefore, it is a very mature algorithm and widely used in engineering and science as a general numerical approach for the solution of PDEs subject to known boundary and initial conditions. The use of piecewise continuous functions over subregions of domain to approximate the unknown function was first introduced by Courant [3]. This approach was later formalized [4, 5] and term finite elements for these subregions was introduced by Clough [6]. Therefore, similar to finite difference technique the FEM is also local in nature. However, FEM has superior and unique characteristics to describe very complex geometries and boundaries of domains. There are plenty of textbook and online materials covering both theoretical and practical aspects of FEM or FEA and some of them are listed in the reference section, [7–11].

We will start this section by first introducing the isoparametric representation of domain and its numerical integration. Then, the strong and weak forms of FEM formulation will be demonstrated for simple transient heat transfer problem, since all the case studies given in this chapter utilize the

weak form of phase-field equations in their FEM implementation. Then, a working FEM code for linear elasticity will be developed by using the formulism based on the principals of virtual work. This code will form the backbone for solving different phase-field models with FEM given in this chapter. Therefore, the reader is urged to review this code and associated functions, even if they are seasoned FEM programmers. There are also dedicated books, such as [12–14], just covering Matlab/Octave implementation of FEM. It is possible to code a FEM with 50 lines or less with Matlab/Octave as demonstrated in [15–17]. However, to bring the clarity to the computational aspects and its underlying fundamentals of the algorithm, we will proceed with more traditional implementation of FEM by following the procedures and the notation given in [18–20].

6.2 Isoparametric Representation and Numerical Integration

The main concept in the development isoparametric formulation is that the geometry of an element is defined using its nodal coordinates and the same shape functions which are used to interpolate the nodal unknowns (i.e., displacements, temperatures, chemical concentrations). In isoparametric formulism, it is convenient to express the shape functions in terms of a nondimensional, localized coordinate system (ζ and η) in which they vary

The original version of this chapter was revised. An erratum to this chapter can be found at DOI [10.1007/978-3-319-41196-5_9](https://doi.org/10.1007/978-3-319-41196-5_9)

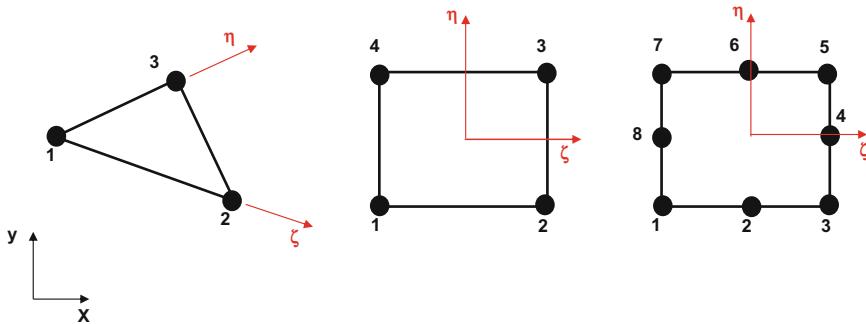


Fig. 6.1 Global and local definition of isoparametric elements. Numbering of element nodes follows the sequence of the node numbers shown in the figure

from -1 to 1 over an element. The adaptation of this local coordinate system is particularly useful for the numerical integrations that are required for the formation of system of equations in FEM. The notation and the isoparametric element types used in the formulation of the FEM codes in the book are summarized in Fig. 6.1. In addition to numbering sequence of the nodes in the programs used in this chapter, the figure also shows the representation of elements in their local coordinates.

Any point within an element with x and y global Cartesian coordinates are represented in the local coordinates with use of element shape functions. In isoparametric formulism for two-dimensional case:

$$\begin{aligned} x(\xi, \eta) &= \sum_i^n N_i^e x_i^e \\ y(\xi, \eta) &= \sum_i^n N_i^e y_i^e \end{aligned} \quad (6.1)$$

where n is the number of nodes in the element and N_i^e is nodal values of shape functions, x_i^e and y_i^e are the nodal coordinate values in Cartesian coordinate system.

The shape functions are not arbitrary functions, they are chosen to satisfy the following requirements.

- Interpolation condition: Takes a unit value at node i and is zero at other nodes of the element.
- Local support condition: Vanishes over any element boundary that does not include the node i .

- Interelement compatibility condition: Satisfies continuity between adjacent elements over any element boundary that includes node i .
- Completeness condition: The interpolation is able to represent exactly any displacement field which is a linear polynomial in x and y .

The Cartesian derivative of any function f is defined over the element using the following expressions:

$$f(\xi, \eta) = \sum_i^n N_i^e f_i^e \quad (6.2)$$

where f_i^e is the value of the function at the element nodal positions. Utilizing the chain rule of differentiation:

$$\begin{aligned} \frac{\partial f}{\partial x} &= \frac{\partial f}{\partial \xi} \cdot \frac{\partial \xi}{\partial x} + \frac{\partial f}{\partial \eta} \cdot \frac{\partial \eta}{\partial x} \\ \frac{\partial f}{\partial y} &= \frac{\partial f}{\partial \xi} \cdot \frac{\partial \xi}{\partial y} + \frac{\partial f}{\partial \eta} \cdot \frac{\partial \eta}{\partial y} \end{aligned} \quad (6.3)$$

in which $\partial f / \partial \xi$ and $\partial f / \partial \eta$ are calculated as:

$$\begin{aligned} \frac{\partial f}{\partial \xi} &= \sum_i^n \frac{\partial N_i^e}{\partial \xi} \cdot f_i^e \\ \frac{\partial f}{\partial \eta} &= \sum_i^n \frac{\partial N_i^e}{\partial \eta} \cdot f_i^e \end{aligned} \quad (6.4)$$

where the terms $\partial N_i^e / \partial \xi$ and $\partial N_i^e / \partial \eta$ are the derivatives of shape function at local coordinates.

The values of $\partial \xi / \partial x$, $\partial \eta / \partial x$, $\partial \xi / \partial y$, and $\partial \eta / \partial y$ are obtained, first forming a Jacobian matrix and then inverting this Jacobian matrix.

$$\begin{aligned} J^e &= \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{bmatrix} \\ &= \begin{bmatrix} \sum_i^n \frac{\partial N_i^e}{\partial \xi} \cdot x_i^e & \sum_i^n \frac{\partial N_i^e}{\partial \xi} \cdot y_i^e \\ \sum_i^n \frac{\partial N_i^e}{\partial \eta} \cdot x_i^e & \sum_i^n \frac{\partial N_i^e}{\partial \eta} \cdot y_i^e \end{bmatrix} \quad (6.5) \end{aligned}$$

Then, the inverse of Jacobian matrix, J^e , is:

$$[J^e]^{-1} = \begin{bmatrix} \frac{\partial \xi}{\partial x} & \frac{\partial \xi}{\partial y} \\ \frac{\partial \eta}{\partial x} & \frac{\partial \eta}{\partial y} \end{bmatrix} = \frac{1}{\det J^e} \begin{bmatrix} \frac{\partial y}{\partial \eta} & -\frac{\partial y}{\partial \xi} \\ -\frac{\partial x}{\partial \eta} & \frac{\partial x}{\partial \xi} \end{bmatrix} \quad (6.6)$$

where $\det J^e$ is the determinant of the Jacobian matrix and it is also used in calculating the area/volume of the elements:

$$dxdy = \det J^e d\xi d\eta \quad (6.7)$$

Any function $f(x, y)$ defined over an isoparametric element can be integrated numerically by transforming it from global Cartesian coordinates to local coordinates as:

$$\begin{aligned} \int_{\Omega^e} \int f(x, y) dx dy &= \int_{-1}^{+1} \int_{-1}^{+1} f(\xi, \eta) \det J^e d\xi d\eta \\ &= \int_{-1}^{+1} \int_{-1}^{+1} g(\xi, \eta) d\xi d\eta \quad (6.8) \end{aligned}$$

By utilizing Gauss–Legendre rule for four- and eight-node isoparametric elements with n -point integration, the above integral can be expressed as:

$$\int_{-1}^{+1} \int_{-1}^{+1} g(\xi, \eta) d\xi d\eta = \sum_{i=1}^n \sum_j^n W_i W_j g(\bar{\xi}_i, \bar{\eta}_i) \quad (6.9)$$

where W_i and W_j are the weight coefficients and $g(\bar{\xi}_i, \bar{\eta}_i)$ corresponds to the value evaluated at the sampling points. For three-node isoparametric elements the integration rule is chosen as Radau rule and in this case Eq. 6.9 takes the following form:

$$\int_{\Omega^e} g(\xi, \eta) d\xi d\eta = \sum_{i=1}^n W_i g(\bar{\xi}_i, \bar{\eta}_i) \quad (6.10)$$

6.3 Introduction to Strong and Weak Forms of FEM Formulation

The governing PDEs, based on the heat balance, for transient heat conduction can be described as

$$\rho c \frac{\partial T}{\partial t} + \operatorname{div} q - Q = 0 \quad (6.11)$$

where ρ is the density of the material, c is the specific heat, T is the temperature, t is the time, q is the heat flux, Q is the volumetric heat generation per unit volume per unit time.

The heat flux q is described with Fourier constitutive relation which states that heat flows opposite to the gradient of the temperature. For materials that have different thermal conductivities in orthogonal directions, the constitutive model is:

$$q = -\kappa (\operatorname{grad} T)^T \quad (6.12)$$

where κ is the conductivity matrix:

$$\kappa = \begin{bmatrix} \kappa_x & 0 & 0 \\ 0 & \kappa_y & 0 \\ 0 & 0 & \kappa_z \end{bmatrix} \quad (6.13)$$

For thermally isotropic materials $\kappa = \kappa_x = \kappa_y = \kappa_z$ and

$$\operatorname{grad} T = \left[\frac{\partial T}{\partial x} \quad \frac{\partial T}{\partial y} \quad \frac{\partial T}{\partial z} \right]^T \quad (6.14)$$

Assuming isotropic material and inserting Eqs. 6.12 and 6.13 into Eq. 6.11, it becomes

$$\rho c \frac{\partial T}{\partial t} - \operatorname{div} [k (\operatorname{grad} T)^T] - Q = 0 \quad (6.15)$$

Equation 6.15 together with the boundary and initial conditions represents the *strong-form* of the heat conduction problem. As can be seen, in order to solve Eq. 6.15, it is necessary to compute the second-order spatial derivatives of temperature.

A weak form of set of PDEs to be solved with FEM usually are obtained with the following four steps.

- Multiply PDEs with test function which reduces the equation to a scalar.
- Integrate the resulting equation from first step over the domain under consideration.
- Integrate by parts using Green's theorem to reduce the order of derivatives.
- Supplement the resulting equation with appropriate boundary conditions.

Step-1

By multiplying Eq. 6.11 with test function η the residual becomes

$$\eta\rho c \frac{\partial T}{\partial t} + \eta \operatorname{div} q - \eta Q = 0 \quad (6.16)$$

Step-2

Integrate Eq. 6.16 over a domain of interest.

$$\int_{\Omega} \eta\rho c \frac{\partial T}{\partial t} d\Omega + \int_{\Omega} \eta \operatorname{div} q d\Omega - \int_{\Omega} \eta Q d\Omega = 0 \quad (6.17)$$

Step-3

Integrate the parts involving with partial derivatives using Green's theorem which can be expressed as a general case:

$$\int_{\Omega} u \cdot v_{,i} d\Omega = - \int_{\Omega} u_{,i} \cdot v d\Omega + \int_{\Gamma} (u \cdot v) \cdot n_i d\Gamma \quad (6.18)$$

in which, i represents the partial derivatives of function for the spatial dimensions and Γ is the surface around the domain Ω and n_i is the components of the outward vector normal to surface Γ . Thus, by integrating the second integral on the left-hand side of Eq. 6.17 by parts, it becomes:

$$\int_{\Omega} \eta \operatorname{div} q d\Omega = - \int_{\Omega} (\operatorname{grad} \eta) q d\Omega + \int_{\Gamma} (\eta \cdot q) \cdot n d\Gamma \quad (6.19)$$

Step-4

The some values of $q \cdot n$ is known on some parts of the boundary. The other parts of the boundary may also have heat flux through the convection. In addition, heat flux may also take place through radiation which is not considered in here. Thus, surface integral is modified to take these effects as below:

$$\int_{\Gamma} (\eta \cdot q) \cdot n d\Gamma = \int_{\Gamma_1} \eta \bar{q}_n d\Gamma_1 + \int_{\Gamma_2} \eta h(T - T_e) d\Gamma_2 \quad (6.20)$$

where h is the convection coefficient and T_e is the convective exchange temperature.

Expendng Eq. 6.17 with these modifications, it takes the following form:

$$\begin{aligned} & \int_{\Omega} \eta\rho c \frac{\partial T}{\partial t} d\Omega + \int_{\Omega} (\operatorname{grad} \eta) \kappa (\operatorname{grad} T)^T d\Omega \\ & - \int_{\Omega} \eta Q d\Omega + \int_{\Gamma_1} \eta \bar{q}_n d\Gamma_1 + \int_{\Gamma_2} \eta h(T - T_e) d\Omega \\ & = 0 \end{aligned} \quad (6.21)$$

Now, Eq. 6.21 is the *weak-form* of Eq. 6.15. As can be seen, in order to solve the problem, Eq. 6.15 requires only the first derivatives in comparison to strong form.

6.3.1 FEM Discretization of Weak Form

By utilizing the isoparametric formulism presented in the previous section, Eq. 6.21 can be cast into a FEM solution by replacing the test functions with the shape functions. The integrals appearing in Eq. 6.21 given below are at an element level. Recalling that any point within an element and any function over an element can be described with the nodal values of shape functions with Eqs. 6.1 and 6.2, respectively,

$$x(\xi, \eta) = \sum_i^n N_i^e x_i^e \quad (6.22)$$

$$T(\zeta, \eta) = \sum_i^n N_i^e T_i^e \quad (6.23)$$

where n is the number of nodes in the element and N_i^e is nodal values of shape functions, x_i^e and T_i^e are the nodal values. Again, the gradient terms appearing in Eq. 6.21 can be expressed with shape functions,

$$\text{grad}T = \begin{bmatrix} \frac{\partial N_1}{\partial x} \frac{\partial N_2}{\partial x} \dots \frac{\partial N_n}{\partial x} \\ \frac{\partial N_1}{\partial y} \frac{\partial N_2}{\partial y} \dots \frac{\partial N_n}{\partial y} \end{bmatrix} T_i^e = B^e T_i^e \quad (6.24)$$

The components of B matrix can be easily evaluated by forming the Jacobian matrix and utilizing the chain rule of differentiation as described with Eqs. 6.4–6.6.

By utilizing Eqs. 6.22–6.24 and Eq. 6.21 can be expressed at element level as

$$\int_{\Omega} [N \rho c N^T] \frac{\partial T^e}{\partial t} d\Omega + \int_{\Omega} [B \kappa B^T] T d\Omega - \int_{\Omega} [N Q] d\Omega - \int_{\Gamma_1} [N \bar{q}_n] d\Gamma_1 + \int_{\Gamma_2} [N h N] T d\Gamma_2 - \int_T [N h] T_e d\Gamma_2 = 0 \quad (6.25)$$

In matrix representation and rearranging:

$$C^e \frac{\partial T}{\partial t} + K^e T + H^e T = F_1 + F_2 + F_3 \quad (6.26)$$

where:

Capacity matrix:

$$C^e = \int_{\Omega} [N \rho c N^T] d\Omega \quad (6.27)$$

Conductivity matrix:

$$K^e = \int_{\Omega} [B \kappa B^T] d\Omega \quad (6.28)$$

Surface heat transfer matrix:

$$H^e = \int_{\Gamma_2} [N h N^T] d\Omega \quad (6.29)$$

6.3.2 Discretization in Time for Transient Heat Transfer

A generalized trapezoidal method for FEM solutions involving time integration is described in [10], including its accuracy and stability properties. Accordingly, the generalized trapezoidal method can be cast into following relationship for temperatures and their rates at two different time steps, t_n and t_{n+1} , as

$$\frac{T_{n+1} - T_n}{\Delta t} = \theta \dot{T}_{n+1} + (1 - \theta) \dot{T}_n \quad (6.30)$$

where $\Delta t = t_{n+1} - t_n$ is the time increment between two consecutive time steps.

Equation 6.30 is applied to the time integration of Eq. 6.26, by writing Eq. 6.26 at two consecutive time steps t_n and t_{n+1} , and then adding two equations.

$$\begin{aligned} & \theta [C^e \dot{T}_{n+1} + K^e T_{n+1} + H^e T_{n+1}] \\ &= \theta [F_1^e + F_2^e + F_3^e]_{n+1} \end{aligned} \quad (6.31)$$

and

$$\begin{aligned} & (1 - \theta) [C^e \dot{T}_n + K^e T_n + H^e T_n] \\ &= (1 - \theta) [F_1^e + F_2^e + F_3^e]_n \end{aligned} \quad (6.32)$$

By rearranging the resulting equation from this addition and using Eq. 6.30 the final FEM equation for an element for the case of transient heat transfer takes the form given below, assuming that there is no change in the prescribed boundary condition between the two time steps:

$$\begin{aligned} & \left[\frac{1}{\Delta t} C^e + \theta (K^e + H^e) \right] T_{n+1} \\ &= \left[\frac{1}{\Delta t} C^e - (1 - \theta) (K^e + H^e) \right] T_n \\ &+ f_1^e + f_2^e + f_3^e \end{aligned} \quad (6.33)$$

Equation 6.33 can be solved provided that T_n is known from the previous time step, t_n .

As described earlier in Chap. 2, the following values for θ leads to:

- $\theta = 0$ forward (explicit) Euler method.
- $\theta = 0.5$ Crank–Nicholson method.
- $\theta = 1$ backward (implicit) Euler method.

The forward Euler method has the limitation on conditional stability, leading to severe restrictions on the time steps size; on the other hand, both Crank–Nicholson and backward Euler methods are unconditionally stable. However, backward Euler method is usually preferred in FEM solutions owing to the oscillations in the solutions obtained with the Crank–Nicholson method even though it is second-order accurate. Also, it should be noted that Eq. 6.33 in present form does not include any temperature-dependent variations in the material properties; in other words, it represents the formulism for transient linear heat transfer problem.

6.4 FEM Formulation of Linear Elasticity Based on the Principles of Virtual Work

With the isoparametric formulism given earlier, the displacements, δ , within any point of an element can be obtained from the displacement values at the nodes by utilizing the shape functions (i.e., Eq. 6.2), thus:

$$\delta = \begin{bmatrix} u \\ v \end{bmatrix} = N\delta^e = \sum_i^n N_i \delta_i \quad (6.34)$$

where u and v are the displacement components in x - and y -directions in a Cartesian coordinate system, N_i are the values of shape functions and δ_i are the values of the displacements at the element nodes. The summation is carried out for total number of nodes, n , for that element.

For linear elasticity, the strains within an element can be obtained from the derivatives of nodal displacements as:

$$\varepsilon = \begin{bmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ \varepsilon_{xy} \end{bmatrix} = \begin{bmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial v}{\partial y} \\ \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \end{bmatrix} \quad (6.35)$$

Inserting the displacement values from Eqs. 6.34 and 6.35 takes the form of:

$$\varepsilon = B\delta^e = \sum_{i=1}^n B_i \delta_i \quad (6.36)$$

where B is termed strain matrix with the following components:

$$B_i = \begin{bmatrix} \frac{\partial N_i}{\partial x} & 0 \\ 0 & \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial y} & \frac{\partial N_i}{\partial x} \end{bmatrix} \quad (6.37)$$

The derivatives appearing in the strain matrix can be easily evaluated by utilizing the chain rule and the inverse of the Jacobian matrix given earlier (see Eqs. 6.5 and 6.6).

For linear elasticity the Hook's law provides the necessary constitutive relationship between the stresses and the strains within an element. For example, this relationship in two dimensions for isotropic materials is expressed as

$$\sigma = \begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{bmatrix} = D \begin{bmatrix} \varepsilon_{xx} - \varepsilon_{xx}^0 \\ \varepsilon_{yy} - \varepsilon_{yy}^0 \\ \varepsilon_{xy} - \varepsilon_{xy}^0 \end{bmatrix} + \begin{bmatrix} \sigma_{xx}^0 \\ \sigma_{yy}^0 \\ \sigma_{xy}^0 \end{bmatrix} \quad (6.38)$$

where $[\varepsilon_{xx}^0, \varepsilon_{yy}^0, \varepsilon_{xy}^0]$ and $[\sigma_{xx}^0, \sigma_{yy}^0, \sigma_{xy}^0]$ are the initial strains and stresses that exist in the solid. The D matrix is called elasticity matrix, and for plane-stress case, its components are:

$$D = \frac{E}{1-v^2} \begin{bmatrix} 1 & v & 0 \\ v & 1 & 0 \\ 0 & 0 & \frac{1-v}{2} \end{bmatrix} \quad (6.39)$$

and for plane-strain case, its components are:

$$D = \frac{E(1-v)}{(1+v)(1-2v)} \begin{bmatrix} 1 & \frac{v}{1-v} & 0 \\ \frac{v}{1-v} & 1 & 0 \\ 0 & 0 & \frac{1-2v}{2(1-v)} \end{bmatrix} \quad (6.40)$$

in which E is the Young's modulus and v is the Poisson's ratio.

External nodal loads F^e and body forces p acting on a single element create an equilibrating stress field σ within the element. If this element is subjected to an arbitrary virtual displacement field δ_*^e , it results in compatible internal displacements δ_* and strains ϵ_* within the element. The principles of virtual work requires that

$$[\delta_*^e]^T F^e + \int_{\Omega_e} [\delta_*]^T p d\Omega_e = \int_{\Omega_e} [\epsilon_*]^T \sigma d\Omega_e \quad (6.41)$$

in which the integrations are carried over element area/volume, Ω_e , and superscript T represents the transpose.

Inserting Eqs. 6.34 and 6.35 into Eq. 6.41 produces:

$$[\delta_*^e]^T \left\{ F^e + \int_{\Omega_e} [N]^T p d\Omega_e \right\} = [\delta_*^e] \int_{\Omega_e} [B]^T \sigma d\Omega_e \quad (6.42)$$

Owing to the fact that virtual displacement field is arbitrary, and by eliminating them:

$$F^e + \int_{\Omega_e} [N]^T p d\Omega_e = \int_{\Omega_e} [B]^T \sigma d\Omega_e \quad (6.43)$$

Putting expressions for strains (Eq. 6.36) and stresses (Eq. 6.38), Eq. 6.43 becomes

$$\begin{aligned} F^e + \int_{\Omega_e} [N]^T p d\Omega_e &= \left\{ \int_{\Omega_e} [B]^T D B d\Omega \right\} \delta^e \\ &\quad - \int_{\Omega_e} [B]^T D \epsilon^0 d\Omega + \int_{\Omega_e} [B]^T \sigma^0 d\Omega \end{aligned} \quad (6.44)$$

By rearranging Eq. 6.44, the resulting system of simultaneous equations for an element are:

$$K^e \delta^e = F^e + F_{\epsilon^0}^e + F_{\sigma^0}^e \quad (6.45)$$

where δ^e is the displacement vector and,

Stiffness matrix:

$$K^e = \int_{\Omega_e} [B]^T D B d\Omega \quad (6.46)$$

Applied force vector:

$$F^e = [P_x^1, P_y^1, \dots, P_x^n, P_y^n] \quad (6.47)$$

Force vector due to body forces:

$$F_p^e = \int_{\Omega_e} [N]^T p d\Omega \quad (6.48)$$

Force vector due to initial internal strains:

$$F_{\epsilon^0}^e = \int_{\Omega_e} [B]^T D \epsilon^0 d\Omega \quad (6.49)$$

Force vector due to initial internal stresses:

$$F_{\sigma^0}^e = - \int_{\Omega_e} [B]^T \sigma^0 d\Omega \quad (6.50)$$

Of course, Eq. 6.45 is only for a single element. Therefore, Eq. 6.45 should be evaluated for each element; then their values are summed altogether into a system of simultaneous equations to represent the overall solution.

In general, the steps taken for a FEM solution are:

- Subdivision of the geometry into set of elements and establishing the Cartesian coordinates of the nodes and inclusion of boundary conditions in the form of an input file. These are usually accomplished by using FEM mesh generators. A simple yet an efficient FEM mesh generator Matlab/Octave program is given in *Appendix-C* which is used for generation of all input files that are used in this chapter.

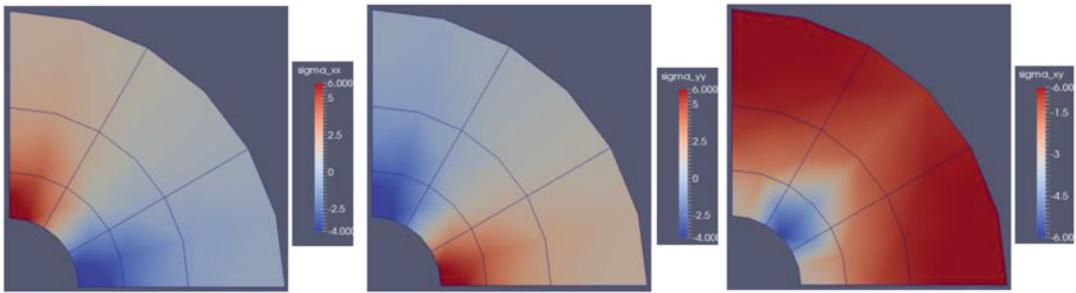


Fig. 6.2 Stress fields obtained from a plane-strain solution of *fem_elast_v1.m* for an internally pressurized thick cylinder. The *left* is for σ_{xx} , the *middle* is for σ_{yy} , and the *right* is for σ_{xy}

- Reading the input file into the FEM program.
- Evaluation of element stiffness matrices and force vectors and their assembly into a global stiffness matrix (left-hand side, lhs) and force vector (right-hand side, rhs).
- Solution of nodal unknowns (displacements, temperatures, chemical concentrations, etc.) from the resulting system of simultaneous equations.
- Calculation of other desired quantities such as stress and strain.
- Output the overall solutions into output files (tabulated/graphical).

6.4.1 A Numerical Example for FEM Solution of Linear Elasticity

A linear elastic solution of internally pressurized thick circular cylinder conforming plane-strain condition in two dimensions is given as the numerical example for the FEM program *fem_elast_v1.m* developed by following the formulism and the steps given earlier in this section. The input file together with the output files is provided in subdirectory *elasticity* in downloadable file.

The resulting stress fields from the solution is given in Fig. 6.2 together with the FEM mesh used in the analysis. Because of the symmetry properties only one-quarter of the cylinder was analyzed by imposing appropriate boundary conditions along the x - and y -axis (see input file). There were 40 nodes and 9 eight node-node elements in the FEM mesh. The number of

integration points was 9, ($\text{ngaus} = 3$). The number of degree of freedom per node was 2, $\text{ndofn} = 2$, representing the displacement vector in the x and y Cartesian coordinate directions. The magnitude of the pressure on the inner surface was 10 units. The results seen in Fig. 6.2 in graphical form and tabulated values given in *results_1.out* file are in excellent agreement with the one given in [19].

6.5 Source Codes

As before, a modular approach is chosen in an assembled FEM programs in this chapter. In the following, for each function routines, a variables and arrays list explaining their contents is given first. The listing of the source codes follows this as they appear in the downloadable file, with the line numbers. Lastly, the annotation of line numbers in the source code will be provided.

In addition, the input file to solve thick cylinder with internal pressure problem given above is also described in here, since similar format is used for the input files of the case studies given throughout this chapter.

Program

fem_elast_v1.m

This FEM program solves the linear elasticity problems in two dimensions by using three-, four-, and eight-node isoparametric elements.

The program makes calls to the following functions:

- **input_fem_elast.m**
- **gauss.m**
- **stiffnes.m**
- **loads.m**
- **boundary_cond.m**
- **stress.m**
- **output_fem.m**

Listing:

```

1  %%%%%%
2  %
3  % FEM CODE FOR LINEAR ELASTICITY %
4  %
5  %%%%%%
6
7  % get wall time:
8
9  time0=clock();
10 format long;
11
12 %--- Open input & output files:
13
14 global in;
15 in=fopen('mesh_1.inp','r');
16
17 global out;
18 out=fopen('result_1.out','w');
19
20 %-----
21 %Input FEM data:
22 %-----
23
24 [npoin,nelem,nvfix,ntype,nnode,
25 ndofn,ndime/ngaus, ...
25 nstre,nmats,nprop,lnods,matno,
26 coord,props,nofix, ...
26 iffix,fixed]=input_fem_elast();
27 ntotv=npoin*ndofn;
28 [posgp, weigp]=gauss(ngaus,nnode);
29
30 %-----
31 %Form global stiffness matrix (lhs):
32 %-----
33 [gstif]=stiffness(npoin,nelem,
33 nnode,nstre,ndime,ndofn, ...
34 ngaus,ntype,lnods,matno,coord,
34 props, ...
35 posgp,weigp);
36
37 %-----
38 % Force vector & Boundary Conditions:
39 %-----
40
41 [gforce]=loads(npoin,nelem,ndofn,
41 nnode,ngaus,ndime,posgp, ...
42 weigp,lnods,coord);
43
44 [gstif,gforce]=boundary_cond
44 (npoin,nvfix,nofix,iffix, ...
45 fixed,ndofn,gstif,gforce);
46
47 % Solve displacements :
48
49 asdis=gstif\gforce;
50
51
52 %-----
53 % calculate stresses:
54 %-----
55
56 [elem_stres]=stress(asdis,nelem,
56 npoin,nnode,ngaus,nstre,props, ...
57 ntype,ndofn,ndime,lnods,matno,
57 coord,posgp,weigp);
58
59
60 % output results:
61
62
63 output_fem(npoin,nelem,nnode,
63 lnods,coord,ndofn,ngaus,nstre,
63 asdis,elem_stres);
64
65 %--- end of execution:
66 disp('Done')
67 compute_time=etime(clock(),time0)

```

Line numbers:

9:	Get initial wall clock time beginning of the execution.
12–19:	Assign unit names for input and output files. These file names should be modified for different input and output files.
24–26:	Input solution variables and FEM mesh.
28:	Get the values for the numerical integration.
33–35:	Form the global stiffness matrix.

(continued)

41–42:	Form the global force vector from the loading information.
44–45:	Apply boundary conditions.
50:	Solve the resulting system of equation for the nodal displacements.
56–57:	Calculate stress values.
63:	Output displacements and stress values to files.
67:	Display total execution time for the solution.

Function

input fem elast.m

This function reads the input file for the FEM analysis of two-dimensional elasticity using three-, four-, and eight-node isoparametric elements.

Variable and array list:

npoin:	Total number of nodes in the solution.
nelem:	Total number of elements.
nvfix:	Total number of constrained nodes.
ntype:	Solution type (ntype = 1, plane-stress and ntype = 2 plane-strain).
nnode:	Number of nodes per element.
ndofn:	Number of degree of freedom (DOF) per node.
ndime:	Number of spatial dimensions.
ngaus:	The order of numerical integration.
nmats:	Total number of different materials in the solution.
nstre:	Number of stress components.
nprop:	Number of material properties.
matno(nelem):	Material types for the elements.
nofix(nvfix):	Node numbers at which one or more DOFs are constrained.
lnods(nelem, nnode):	Element nodal connectivity list.
coord(npoin, ndime):	Cartesian coordinates of each node.
iffix(nvfix, ndofn):	List of constrained DOFs.
fixed(nvfix, ndofn):	Prescribed value of any constrained DOFs.
props(nmats, nprops):	For each different material, the properties of that material.

Listing:

```
1     function [npoin,nelem,nvfix,ntype,
2             nnode,ndofn,ndime,ngaus, ...
2             nstrel,nmats,nprop,lnods,
2             matno.coord.props.nofix, ...
```

```

3      iffix, fixed]=input_fem_
4          elast( )
5
6      global in;
7      global out;
8
9      %--- read the Control data:
10
11     npoin=fscanf(in,'%d',1);
12     nelem=fscanf(in,'%d',1);
13     nvfix=fscanf(in,'%d',1);
14     ntype=fscanf(in,'%d',1);
15     nnodes=fscanf(in,'%d',1);
16     ndofn=fscanf(in,'%d',1);
17     ndime=fscanf(in,'%d',1);
18     ngaus=fscanf(in,'%d',1);
19     nstrel=fscanf(in,'%d',1);
20     nmats=fscanf(in,'%d',1);
21     nprop=fscanf(in,'%d',1);
22
23
24      %--- Element node numbers & material
25      % property number
26
27      for ielem=1:nelem
28          jelem=fscanf(in,'%d',1);
29          dummy=fscanf(in,'%d',[nnodes+1,1]);
30          for inode=1:nnodes
31              lnods(jelem,inode)=dummy(inode);
32          end
33          matno(jelem)=dummy(nnodes+1);
34      end
35
36      %--- Nodal coordinates:
37
38      for ipoin=1:npoin
39          jpoin=fscanf(in,'%d',1);
40          dummy=fscanf(in,'%lf %lf',[2,1]);
41          for idime=1:ndime
42              coord(ipoin,idime)=dummy(idime);
43          end
44      end
45
46      %-- Constraint nodes and their
47      % values:
48
49      for ivfix=1:nvfix
50          nofix(ivfix)=fscanf(in,'%d',1);
51          dummy1=fscanf(in,'%d %d',[2,1]);
52          dummy2=fscanf(in,'%lf %lf',[2,1]);

```

```

51 for idofn=1:ndofn
52 iffix(ivfix,idofn)=dummy1(idofn);
53 fixed(ivfix,idofn)=dummy2(idofn);
54 end
55 end
56
57 %--- Material properties:
58
59 for imats=1:nmats
60 jmats=fscanf(in,'%d',1);
61 dummy=fscanf(in,'%lf %lf',[2,1]);
62 for iprop=1:nprop
63 props(jmats,iprop)=dummy(iprop);
64 end
65 end
66
67 %%printout:
68
69 fprintf(out,'*****\n');
70 fprintf(out,'* FEM input data *\n');
71 fprintf(out,'*****\n');
72 fprintf(out,'\n');
73
74 fprintf(out,'Number of Elements : %5d\n',nelem);
75 fprintf(out,'Number of Node : %5d\n',npoin);
76 fprintf(out,'Number of Fixed nodes : %5d\n',nvfix);
77 fprintf(out,'Number of Nodes per element : %5d\n',nnode);
78 fprintf(out,'Number of Integration points : %5d\n',ngaus);
79 fprintf(out,'Number of Materials : %5d\n',nmats);
80 fprintf(out,'Number of properties : %5d\n',nprop);
81 fprintf(out,'\n');
82 %--
83
84 if(ntype == 1)
85   fprintf(out,'Plane-stress elasticity solution\n');
86 end
87 if(ntype == 2)
88   fprintf(out,'Plane-strain elasticity solution\n');
89 end
90 %--
91 end %endfunction

```

Line numbers:

6–7:	Unit names for input and output files.
9–21:	Read the control data (npoin nprop).
24–33:	Read element connectivity list (lnods) and material number list (matno).
36–43:	Read nodal coordinates (coord).
46–55:	Read the data on constrained nodes (nofix, iffix, and fixed).
57–66:	Read each material properties.
67–90:	Print out solution input parameters to the output file.

Function**sfr2.m**

This function evaluates the values of shape functions and their derivatives in local coordinates.

Variable and array list:

exisp:	x-coordinate of sampling point in local coordinates.
etasp:	y-coordinate of sampling point in local coordinates.
nnode:	Number of nodes per element.
shape (nnode):	Shape functions values for each node in an element.
deriv(ndime, nnode):	Derivatives of shape functions for each local coordinate direction.

Listing:

```

1 function [shape, deriv] = sfr2(exisp,
2                                 etasp,nnode)
3 format long;
4
5 if(nnode == 3)
6   %--- 3 nodes elements:
7
8 s=exisp;
9 t=etasp;
10 p=1.0-s-t;
11
12 shape(1)=p;
13 shape(2)=s;
14 shape(3)=t;
15
16 deriv(1,1)=-1.0;
17 deriv(1,2)=1.0;
18 deriv(1,3)=0.0;

```

```

19                                *0.25;
20 deriv(2,1)=-1.0;
21 deriv(2,2)=0.0;
22 deriv(2,3)=1.0;
23
24 end
25
26 if(nnode == 4 )
27 %-- 4 nodes elements
28
29 s=exisp;
30 t=etasp;
31 st=s*t;
32
33 shape(1)=(1.0-t-s+st)*0.25;
34 shape(2)=(1.0-t+s-st)*0.25;
35 shape(3)=(1.0+t+s+st)*0.25;
36 shape(4)=(1.0+t-s-st)*0.25;
37 %
38 deriv(1,1)=(-1.0+t)*0.25;
39 deriv(1,2)=(1.0-t)*0.25;
40 deriv(1,3)=(1.0+t)*0.25;
41 deriv(1,4)=(-1.0-t)*0.25;
42 %
43 deriv(2,1)=(-1.0+s)*0.25;
44 deriv(2,2)=(-1.0-s)*0.25;
45 deriv(2,3)=(1.0+s)*0.25;
46 deriv(2,4)=(1.0-s)*0.25;
47
48 end
49
50 if(nnode == 8 )
51 %-- 8 nodes elements
52
53 s=exisp;
54 t=etasp;
55 s2=2.0*s;
56 t2=2.0*t;
57 ss=s*s;
58 tt=t*t;
59 st=s*t;
60 sst=s*s*t;
61 stt=s*t*t;
62 st2=2.0*s*t;
63
64 shape(1)=(-1.0+st+ss+tt-sst-stt)
   *0.25;
65 shape(2)=(1.0-t-ss+sst)*0.5;
66 shape(3)=(-1.0-st+ss+tt-sst+stt)

```

Line numbers:

6–24:	The values of shape functions and their derivatives are calculated for three-node isoparametric elements.
26–48:	The values of shape functions and their derivatives are calculated for four-node isoparametric elements.
50–91:	The values of shape functions and their derivatives are calculated for eight-node isoparametric elements.

Function

gauss.m

This function evaluates the position of the sampling points and the associated weights for chosen order of numerical integration.

<i>Variable and array list:</i>	
ngaus:	The order of numerical integration.
nnode:	Number of nodes per element.
posgp (ngaus):	Position of sampling points.
weigp (ngaus):	Weighting factors at the sampling points.

Listing:

```

1  function [posgp, weigp] =gauss
2    (ngaus,nnode)
3    format long;
4
5    if(nnode == 3)
6
7      if(ngaus == 1)
8        posgp(1)=1.0/3.0;
9        posgp(2)=1.0/3.0;
10     weigp(1)=0.5;
11   end
12
13   if(ngaus == 3)
14     posgp(1)=0.5;
15     posgp(2)=0.5;
16     posgp(3)=0.0;
17   %
18   posgp(4)=0.0;
19   posgp(5)=0.5;
20   posgp(6)=0.5;
21
22   weigp(1)=1.0/6.0;
23   weigp(2)=1.0/6.0;
24   weigp(3)=1.0/6.0;
25 end
26
27 if(ngaus == 7)
28   posgp(1)=0.0;
29   posgp(2)=0.5;
30   posgp(3)=1.0;
31   posgp(4)=0.5;
32   posgp(5)=0.0;
33   posgp(6)=0.0;
34   posgp(7)=1.0/3.0;
35
36   posgp(8)=0.0;
37   posgp(9)=0.0;
38   posgp(10)=0.0;
39   posgp(11)=0.5;
40   posgp(12)=1.0;
41   posgp(13)=0.5;
42   posgp(14)=1.0/3.0;
43
44   weigp(1)=1.0/40.0;
45   weigp(2)=1.0/15.0;
46   weigp(3)=1.0/40.0;
47   weigp(4)=1.0/15.0;
48   weigp(5)=1.0/40.0;
49   weigp(6)=1.0/15.0;
50   weigp(7)=9.0/40.0
51 end
52
53 end
54
55
56
57   if (nnode ~=3)
58
59     if(ngaus == 2)
60       posgp(1)=-0.57735026918963;
61       weigp(1)=1.0;
62     end
63
64     if(ngaus > 2)
65       posgp(1)=-0.7745966241483;
66       posgp(2)=0.0;
67       weigp(1)=0.55555555555556;
68     end
69
70   kgaus=ngaus/2;
71   for igash=1:kgaus
72     jgash=ngaus+1-igash;
73     posgp(jgash)=-posgp(igash);
74     weigp(jgash)=weigp(igash);
75   end
76 end
77
78 end %endfunction

```

Line numbers:

5–53:	For three-node isoparametric triangular elements, the position of sampling points and weights using Radau rule.
57–76:	For four- and eight-node isoparametric elements, the position of sampling points and weights by using Gauss–Legendre rule.

Function**jacob2.m**

This function evaluates the Cartesian shape function derivatives, determinant of the Jacobian for the numerical integration of area/volume of elements, and the Cartesian coordinates of the integration points within elements.

Variable and array list:

ielem:	Current element number.
kgasp:	The current integration point number.
ndime:	Number of global coordinate components.
nnode:	Number of nodes per element.
djacb:	The determinate of the Jacobian matrix sampled within the element (ielem) at the integration points and used in calculation of area/volume of the elements.
shape(nnode):	Shape function values.
elcod(ndime, nnode):	Global coordinates of current element (ielem) nodes.
derive(ndime, nnode):	Derivatives of shape functions in local coordinates.
gpcod(ndime, kgasp):	Cartesian coordinates of the integration points within the element (ielem).

Listing:

```

1  function [cartd,djacb,gpcod]
2    =jacob2(ielem,elcod,kgasp,shape,
3      deriv,nnode,ndime)
4
5  format long;
6
7  %--- Gauss point coordinates:
8
9  for idime=1:ndime
10 gpcod(idime,kgasp)=0.0;
11 for inode=1:nnode
12   gpcod(idime,kgasp)=gpcod(idime,
13     kgasp)+elcod(idime,inode) ...
14   *shape(inode);
15 end
16 end
17
18 %--- Jacobian
19 for idime=1:ndime
20   for jdime=1:ndime
21     xjacm(idime,jdime)=0.0;
22     for inode=1:nnode
23       xjacm(idime,jdime)=xjacm(idime,
24         jdime)+deriv(idime,inode)*elcod
25         (jdime,inode);
26     end
27   end
28 end
29 end
30
31 djacb=xjacm(1,1)*xjacm(2,2)-xjacm
32 (1,2)*xjacm(2,1);
33
34 if(djacb <= 0.0)
35 disp('Program Terminated')
36 disp('Zero or negative area for
37 element:');
38 disp(ielem);
39 error('Program terminated zero or
40 negative area');
41
42 %--- Cartesian derivatives:
43 xjaci(1,1)=xjacm(2,2)/djacb;
44 xjaci(2,2)=xjacm(1,1)/djacb;
45 xjaci(1,2)=-xjacm(1,2)/djacb;
46 xjaci(2,1)=-xjacm(2,1)/djacb;
47
48 for idime=1:ndime
49   for inode=1:nnode
50     cartd(idime,inode)=0.0;
51     for jdime=1:ndime
52       cartd(idime,inode)=cartd(idime,
53         inode)+...xjaci(idime,jdime)
54       *deriv(jdime,inode);
55     end
56   end
57 end
58 end
59
60 end %endfunction

```

Line numbers:

6–13:	Calculate the Cartesian coordinates of the integration points within the elements.
16–25:	Form the Jacobian Matrix (Eq. 6.5).
27–36:	Calculate the determinant of the Jacobian matrix. If its value is zero or negative, terminate the overall solution.
40–43:	Calculate the inverse of the Jacobian matrix (Eq. 6.6).
46–54:	Calculate Cartesian derivatives of shape functions using chain rule (Eq. 6.3).

```

20 const=young/(1.0-poiss*poiss);
21 dmatx(1,1)=const;
22 dmatx(2,2)=const;
23 dmatx(1,2)=const*poiss;
24 dmatx(2,1)=const*poiss;
25 dmatx(3,3)=(1.0-2.0*poiss)*const/
2.0;
26 end
27
28 if(ntype == 2)
29
30 %--- Plane Strain:
31
32 const=young*(1.0-poiss)/
((1+poiss)*(1.0-2.0*poiss));
33 dmatx(1,1)=const;
34 dmatx(2,2)=const;
35 dmatx(1,2)=const*poiss/
(1.0-poiss);
36 dmatx(2,1)=const*poiss/
(1.0-poiss);
37 dmatx(3,3)=(1.0-2.0*poiss)*const/
(2.0*(1.0-poiss));
38
39 end
40
41 end %endfunction

```

Function**modps.m**

This function calculates the elasticity matrix for plane-stress or plane-strain.

Variable and array list:

mtype:	Material type, for the current element.
ntype:	Solution type, ntype = 1 for plane-stress, ntype = 2 for plane-strain.
nstre:	Number of stress components.
props(nmats, nprop):	Properties of material type.
dmatx(nstre, nstre):	Elasticity matrix.

Listing:

```

1 function [dmatx] = modps(mtype,
ntype,nstre,props)
2
3 format long;
4
5 %--- Material Parameters:
6
7 young=props(mtype,1);
8 poiss=props(mtype,2);
9
10 for istre=1:3
11 for jstre=1:3
12 dmatx(istre,jstre)=0.0;
13 end
14 end
15
16 if(ntype == 1)
17
18 %--- Plane Stress:
19

```

Line numbers:

7–8:	Determine the values of the Young's modulus and Poisson's ratio.
16–26:	Evaluate the elasticity matrix for plane-stress (Eq. 6.39).
28–39:	Evaluate the elasticity matrix for plane-strain (Eq. 6.40).

Function**bmats.m**

This function forms the strain matrix by using the Cartesian derivatives of the shape functions.

Variable and array list:

nnode:	Number of nodes per element.
shape(nnode):	Shape function values.
cartd(ndime, nnode):	Cartesian derivatives of shape functions at the current integration point.
bmatx(nstre, nevab):	Strain matrix at the current integration point (nevab = nnode × ndofn).

Listing:

```

1 function [bmatx]=bmats(cartd,
shape,nnode)
2
3 format long;
4
5 ngash=0;
6
7 for inode=1:nnode
8
9 mgash=ngash+1;
10 ngash=mgash+1;
11
12 bmatx(1,mgash)=cartd(1,inode);
13 bmatx(1,ngash)=0.0;
14 bmatx(2,mgash)=0.0;
15 bmatx(2,ngash)=cartd(2,inode);
16 bmatx(3,mgash)=cartd(2,inode);
17 bmatx(3,ngash)=cartd(1,inode);
18
19 end
20
21 end %endfunction

```

Line numbers:

9–10:	Counter for the elements of the strain matrix.
12–19:	Evaluate the strain matrix (Eq. 6.37).

Function**dbe.m**

This function multiplies the elasticity matrix and the strain matrix.

Variable and array list:

nevab:	Total number of element variables (nevab = nnode × ndofn).
nstre:	Number of stress components.
bmatx(nstre, nevab):	Strain matrix.
dmatx(nstre, nstre):	Elasticity matrix.
dbmat(nstre, nevab):	Multiplication results of bmatx and dmatx.

Listing:

```

1 function [dbmat] = dbe(nevab,nstre,
bmatx,dmatx)
2

```

```

3 format long;
4
5 for istre=1:nstre
6 for ievab=1:nevab
7 dbmat(istre,ievab)=0.0;
8 for jstre=1:nstre
9 dbmat(istre,ievab)=dbmat(istre,
ievab)+dmatx(istre,jstre) ...
10 *bmatx(jstre,ievab);
11 end
12 end
13 end
14
15 end %endfunction

```

Line numbers:

5–13:	Matrix multiplication of elasticity matrix (dmatx) and strain matrix (bmatx).
-------	---

Function**stiffness.m**

This function forms the stiffness matrices at the element level first, then, assembles them into the global stiffness matrix.

The function makes calls to the following functions:

- **sfr2.m**
- **jacob2.m**
- **modps.m**
- **bmats.m**
- **dbe.m**

Variables and arrays listing:

npoin:	Total number of nodes in the solution.
nelem:	Total number of elements in the solution.
nnode:	Number of nodes per element.
nstre:	Number of stress components.
ndime:	Number of global Cartesian components.
ndofn:	Number of degrees of freedom per node.
ngaus:	The order of Gaussian integration.
ntype:	Solution type, ntype = 1 for plane-stress, ntype = 2 for plane-strain.
matno (nelem):	Material types for elements.
posgp(ngaus):	Position of sampling points for numerical integration.

(continued)

weigp(ngaus):	Weights of sampling points for numerical integration.
lnods(nelem, nnode):	Element nodal connectivity list.
coord(npoin, ndime):	Cartesian coordinates of nodes.
props(manto, ndime):	For each different material, the properties of that material.
gstif(ntotv, ntotv):	Global stiffness matrix (ntotv = npoin \times ndofn).

Listing:

```

1   function [gstif]=stiffness(npoin,
2     nelem,nnode,nstre,ndime,ndofn, ...
3     ngaus,ntype,lnods,matno,coord,
4     props,...)
5   posgp,weigp)
6   format long;
7   %--- Initialize the global
8   stiffness:
9   nevab=nnode*ndofn;
10  ngaus2=ngaus;
11  if(nnode == 3)
12    ngaus2=1;
13  end
14  ntotv=npoin*ndofn;
15  gstif=zeros(ntotv,ntotv);
16
17  %--- Element stiffness & loads:
18
19  for ielem=1:nelem
20
21  %-- Initialize element stiffness:
22
23  for ievab=1:nevab
24  for jevab=1:nevab
25  estif(ievab,jevab)=0.0;
26  end
27  end
28
29  %--- Form elasticity matrix:
30
31  mtype=matno(ielem);
32  dmatx =modps(mtype,ntype,nstre,
33  props);
34  %--- Coordinates of element nodes:
35
36  for inode=1:nnode
37  lnode=lnods(ielem,inode);
38  for idime=1:ndime
39  elcod(idime,inode)=coord(lnode,
40  idime);
41  end
42
43  %--- Integrate the element stiffness :
44
45  kgasp=0;
46  for igaus=1:ngaus
47  exisp=posgp(igaus);
48  for jgaus=1:ngaus2
49  etasp =posgp(jgaus);
50  if(nnode ==3)
51  etasp=posgp(ngaus+igaus);
52  end
53
54  kgasp=kgasp+1;
55
56  [shape,deriv]=sfr2(exisp,etasp,
57  nnode);
58  [cartd,djacb,gpcod]=jacob2(ielem,
59  elcod,kgasp,shape,deriv,nnode,
60  ndime);
61  [bmatx]=bmats(cartd,shape,inode);
62  [dbmat] =dbe(nevab,nstre,bmatx,
63  dmatx);
64
65  dvolu=djacb*weigp(igaus)*weigp
66  (jgaus);
67
68  if(nnode == 3)
69  dvolu=djacb*weigp(igaus);
70
71  end
72
73  %---Form element stiffness:
74  for ievab=1:nevab
75  for jevab=1:nevab
76  for istre=1:nstre
77  estif(ievab,jevab)=estif(ievab,
78  jevab)+bmatx(istre,ievab)* ...
79  dbmat(istre,jevab)*dvolu;
80
81  end
82  end
83
84  end %jgaus

```

```

80 end %igaus
81
82 %--- Form Global stiffness matrix:
83
84 for inode=1:nnode
85 lnode=lnods(ielem,inode);
86 for idofn=1:ndofn
87 itotv=(lnode-1)*ndofn+idofn;
88 ievab=(inode-1)*ndofn+idofn;
89 for jnode=1:nnode
90 knode=lnods(ielem,jnode);
91 for jdofn=1:ndofn
92 jtotv=(knode-1)*ndofn+jdofn;
93 jevab=(jnode-1)*ndofn+jdofn;
94
95 gstif(itotv,jtotv)=gstif(itotv,
96 jtotv)+estif(ievab,jevab);
96 end
97 end
98 end
99 end
100
101
102 end %ielem
103
104 end %endfunction

```

Line numbers:

8:	Calculate total number of element variables.
9–12:	Change the order of numerical integration parameter for three-node isoparametric elements.
14–15:	Calculate the total number of variables in the solution and initialize the global stiffness matrix.
19–102:	Loop over elements.
21–27:	Initialize element stiffness matrix.
31–32:	Depending upon the material type of current element form the elasticity matrix Eqs. 6.39 or 6.40.
34–41:	Define coordinates of the nodes of the current element (ielem) in a local array.
43–80:	Numerical integration loop.
45:	Counter for integration points.
46–52:	Determine the positions of integration points.
54:	Increment integration point counter.
56:	Calculate the shape functions and their derivatives for the current integration point.
57:	Calculate the Cartesian derivatives of the shape functions, the determinant of the Jacobian, and the Cartesian coordinates of the integration points (Eq. 6.6).
58:	Calculate the strain matrix (Eq. 6.37).

59:	Multiply the elasticity matrix with strain matrix.
61:	Calculate the area of the element (Eq. 6.7).
63–65:	Correct the area of the element, if ielem is a three-node isoparametric element.
67–77:	Form the element stiffness matrix (Eq. 6.46).
82–100:	Place the components of the element stiffness matrix into global stiffness matrix based on the information provided by the elemental connectivity list, lnods.

Function**loads.m**

This function integrates the prescribed distributed external loads at the element edges and also the prescribed point loads at the nodes into the global force vector.

The function makes calls to the following functions:

• **sfr2.m***Variable and array list:*

npoin:	Total number of nodes in the solution.
nelem:	Total number of element in the solution.
ndofn:	Number of DOFs per node.
ngaus:	The order of numerical integration.
ndime:	Number of global Cartesian coordinates.
eload(nelem, nevab):	Element force vector (nevab = nnode × ndofn).
lnods(nelem, nnode):	Element nodal connectivity list.
coord(npoin, ndime):	Cartesian coordinates of nodes.
posgp(ngaus):	Position of sampling points for numerical integration.
weigp(ngaus):	Weight of sampling points for numerical integration.
gforce(ntotv):	Global force vector (ntotv = npoin × ndofn).

Listing:

```

1 function [gforce]=loads(npoin,
2 nelem,ndofn,nnode,ngaus,ndime, ...
3 posgp,weigp,lnods,coord)
4 format long;

```

```

5                               55
6 global in;                  56 %--- Distributed Forces:
7 global out;                 57
8
9 %--- Initialize global force vector (rhs)      58 if(nedge ~= 0)
& element loads:               59 fprintf(out,'Number of loaded edges:
10                                %d\n',nedge);
11 nevab = nnode*ndofn;        60 fprintf(out,'List of loaded edges and
12 ntotv = npoin*ndofn;       61 applied loads:\n');
13
14 for itotv=1:ntotv          62 for iedge=1:nedge
15 gforce=zeros(itotv,1);     63
16 end                         64 nodeg=3;
17
18 for ielem=1:nelem          65 if(nnode ~= 8)
19 for ievab=1:nevab          66 nodeg=2;
20 eload(ielem,ievab)=0.0;    67 end
21 end                         68
22 end                         69 %--- Read loads:
23
24 %--- Loading types:         70
25
26 dummy=fscanf(in,'%d %d',[2,1]); 71 neass=fscanf(in,'%d',[1,1]);
27 iplod=dummy(1);            72 dummy=fscanf(in,'%d %d %d',[nodeg,1]);
28 nedge=dummy(2);           73 for iodeg=1:noddeg
29
30 %--- Point forces:         74 noprs(iodeg)=dummy(iodeg);
31
32 if(iplod~=0)              75 end
33 nplod=fscanf(in,'%d',[1,1]); 76
34 for jplod=1:nplod          77 for iodeg=1:noddeg
35 lodpt=fscanf(in,'%d',[1,1]); 78 dummy=fscanf(in,'%lf %lf',[2,1]);
36 dummy=fscanf(in,'%lf %lf',[2,1]); 79 for idofn=1:ndofn
37
38 for idofn=1:ndofn          80 press(iodeg,idofn)=dummy(idofn);
39 point(idofn)=dummy(idofn); 81 end
40 end                         82 end
41
42 for ielem=1:nelem          83
43 for inode=1:nnode          84 %--- Print:
44 nloca=lnods(ielem,inode); 85 fprintf(out,'\n');
45 if(lodpt == nloca);        86 fprintf(out,'%5d',neass);
46 for idofn=1:ndofn          87 for iodeg=1:noddeg
47 nposi=(inode-1)*ndofn+idofn; 88 fprintf(out,'%5d',noprs(iodeg));
48 eload(ielem,nposi)=point(idofn) 89 end
49 end                         90 fprintf(out,'\n');
50 end                         91 for iodeg=1:noddeg
51 end                         92 for idofn=1:ndofn
52 end                         93 fprintf(out,'%14.6e',press(iodeg,
53 end                           idofn));
54 end%if_iplod                94 end
                                95 end
                                96 fprintf(out,'\n');
                                97
                                98 %-- End of reading
                                99 %
100 %--- Integrate along the edges: 100 etasp=-1.0;

```

```

103 for iodeg=1:nodeg
104 lnode=noprs(iodeg);
105 for idime=1:ndime
106 elcod(idime,iodeg)=coord(lnode,idime);
107 end
108 end
109
110 for igaus=1:ngaus
111 exisp=posgp(igaus);
112 [shape, deriv] = sfr2(exisp, etasp, nnode);
113
114 for idofn=1:ndofn
115 pgash(idofn)=0.0;
116 dgash(idofn)=0.0;
117 for iodeg=1:noddeg
118 pgash(idofn)=pgash(idofn)+press(iodeg,
    idofn)*shape(iodeg);
119 dgash(idofn)=dgash(idofn)+elcod(idofn,
    iodeg)*deriv(1,iodeg);
120 end
121 end
122
123 dvolu=weigp(igaus);
124 pxcom=dgash(1)*pgash(2)-dgash(2)*
    pgash(1);
125 pycom=dgash(1)*pgash(1)+dgash(2)*
    pgash(2);
126
127 for inode=1:nnode
128 nloca=lnods(neass,inode);
129 if(nloca==noprs(1))
130 jnode=inode+noddeg-1;
131 kount=0;
132 for knode=inode:jnode
133 kount=kount+1;
134 ngash=(knode-1)*ndofn+1;
135 mgash=(knode-1)*ndofn+2;
136 if(knode > nnode)
137 ngash=1;
138 mgash=2;
139 end
140 eload(neass,ngash)=eload(neass,ngash)
    +pxcom*dvolu*shape(kount);
141 eload(neass,mgash)=eload(neass,mgash)
    +pycom*dvolu*shape(kount);
142 end
143 end
144 end
145 end%igauss
146
147 end%iedge
148 end %nedge
149
150 %--- Print Nodal Forces:
151
152 fprintf(out, '\n');
153 fprintf(out, 'Nodal forces for elements:
    \n');
154 for ielem=1:nelem
155
156 fprintf(out, '\n');
157 fprintf(out, 'Element No: %d\n', ielem);
158 for ievab=1:nevab
159 fprintf(out, '%14.6e', eload(ielem,
    ievab));
160 if((nnode == 8) && (ievab == nevab / 2))
161 fprintf(out, '\n');
162 end
163 end
164 fprintf(out, '\n');
165 end
166
167 %--- Generate global force vector:
168
169 for ielem=1:nelem
170 for inode=1:nnode
171 lnode=lnods(ielem,inode);
172 for idofn=1:ndofn
173 itotv=(lnode-1)*ndofn+idofn;
174 ievab=(inode-1)*ndofn+idofn;
175 gforce(itotv)=gforce(itotv)+eload
    (ielem, ievab);
176 end
177 end
178 end
179
180 end %endfunction

```

Line numbers:

6–7:	Unit names for input and output files.
14–16:	Initialize global force vector.
18–22:	Initialize element force vector.
24–28:	Determine the load types (ipload = 0 no point loading, nedge = number of edges having distributed loads, pressures).
30–54:	Place the load values of the nodes having point loads into the element force vector.
56–148:	Integrate the distributed loads along the element edges and place them into the element force vector.
62–147:	Loop over number of loaded edges.
64–67:	Set the number of nodes on the edges depending on the element type (noddeg = 3

(continued)

	for eight-node isoparametric elements and nodeg = 2 for four- and three-node isoparametric elements).
69–75:	Read element numbers having distributed loads and edge node numbers.
77–82:	Read the values of the distributed loads.
84–97:	Write the distributed loads information to the output file.
100–145:	Integrate the distributed loads along the element edges and incorporate them into the element force vector. Note that, integration is done in one dimension along the edge of the element by setting etasp = -1.0 and only the first three (for eight-node isoparametric elements) or two (for four- or three-node isoparametric elements) shape functions are utilized in this integration.
167–178:	Assemble the values in the element force vector into the global force vector by utilizing the information provided in the element nodal connectivity data, lnods.

```

5
6 ntotv=npoin*ndofn;
7
8 for ivfix=1:nvfix
9 lnode=nofix(ivfix);
10
11 for idofn=1:ndofn
12 if(iffix(ivfix,idofn)==1)
13 itotv=(lnode-1)*ndofn+idofn;
14
15 for jtotv=1:ntotv
16 gstif(itotv,jtotv)=0.0;
17 end
18
19 gstif(itotv,itotv)=1.0;
20 gforce(itotv)=fixed(ivfix,idofn);
21
22 end
23 end
24 end
25
26 end %endfunction

```

Function

boundary_cond.m

This function rearranges the stiffness matrix and force vector for prescribed boundary conditions.

Variable and array list:

npoin:	Total number of nodes in the solution.
nvfix:	Total number of nodes with prescribed boundary conditions.
ndofn:	Number of DOF per node.
nofix(nvfix):	Node numbers of which one or more DOF are constrained.
fixed(nvfix):	The values of prescribed DOFs.
gforce(ntotv):	Global force vector, ntotv = npoin × ndofn
iffix(nvfix, ndofn):	List of constrained DOFs.
gstif(ntotv, ntotv):	Global stiffness matrix.

Line numbers:

6:	Total number of variables in the solution.
8–24:	For nodes that their one or more DOFs are constrained, modify the global stiffness and global force vector corresponding to these DOFs.
16:	Zero the row vector of the global stiffness corresponding to the DOF.
19:	Insert value of one to the diagonal of the global stiffness matrix corresponding to the DOF.
20:	Insert the prescribed value for this DOF to the global force vector.

Function

stress.m

This function calculates the strain and stress values at the integration points of the elements resulting from the displacements.

The function makes call to the following functions:

- **modps.m**
- **sfr2.m**
- **jacob2.m**
- **bmats.m**

Listing:

```

1 function [gstif,gforce]=boundary_cond
(npoin,nvfix,nofix,iffix, ...
2 fixed,ndofn,gstif,gforce)
3
4 format long;

```

Variable and array list:

nelem:	Total number of elements in the solution.
npoin:	Total number of the nodes in the solution.
nnode:	Number of nodes per element.
ngaus:	The order of numerical integration.
nstre:	Number of stress components.
ntype:	Solution type (ntype = 1 for plane-stress, ntype = 2 for plane-strain).
ndofn:	Number of DOF per node.
ndime:	Number of global Cartesian coordinate components.
asdis(ntov):	Global displacement vector (ntov = npoin × ndofn).
manto(nelem):	Material types for the elements.
posgp(ngaus):	Position of sampling points for numerical integration.
weigp(ngaus):	Weights of sampling points for numerical integration.
lnods(nelem, nnode):	Element nodal connectivity list.
props(nmats, nprop):	For each different material the properties of that material.
coord(npoin, ndime):	Cartesian coordinates of nodes.
elem_stres(nelem,mgaus, nstre):	Stress components at element gauss points.

Listing:

```

1 function [elem_stres]=stress
2     (asdis,nelem,npoin,nnode,ngaus,
3     nstre,props, ...
4     ntype,ndofn,ndime,lnods,matno,
5     coord,posgp,weigp)
6
7     format long;
8
9     %--- Number of integration
10    ngaus2=ngaus;
11    if(nnode == 3)
12        ngaus2=1;
13    end
14    mgaus =ngaus*ngaus2;
15
16    %--- Material Parameters and
17    %--- Elasticity Matrix:
18    mtype=matno(ielem);
19    dmatx=modps(mtype,ntype,nstre,
20    props);
21    poiss=props(mtype,2);
22    %--- Nodal Displacements:
23
24    for inode=1:nnode
25        lnode=lnods(ielem,inode);
26        for idofn=1:ndofn
27
28            nposn=(lnode-1)*ndofn+idofn;
29            eldis(idofn,inode)=asdis(nposn);
30            elcod(idofn,inode)=coord(lnode,
31            idofn);
32        end
33    end
34    %--- Integrate Stresses:
35
36    kgasp=0;
37    for igaus=1:ngaus
38        for jgaus=1:ngaus2
39
40            kgasp=kgasp+1;
41            exisp=posgp(igaus);
42            etasp=posgp(jgaus);
43
44            if(ngaus2 == 1)
45                etasp =posgp(ngaus+igaus)
46            end
47
48            [shape,deriv]=sfr2(exisp,etasp,
49            nnode);
50            [cartd,djacb,gpcod]=jacob2(ielem,
51            elcod,kgasp,shape,deriv,nnode,
52            ndime);
53            [bmatx]=bmats(cartd,shape,inode);
54
55            %--- Calculate the strains:
56            for istre=1:nstre
57                stran(istre)=0.0;
58                for inode=1:nnode
59                    for idofn=1:ndofn
60                        ievab=(inode-1)*ndofn+idofn;
61                        stran(istre)=stran(istre)+bmatx
62                        (istre,ievab)*eldis(idofn,inode);
63                    end
64                end
65            end
66        end
67    end
68
```

```

62 end
63
64 %--- Calculate stresses :
65
66 for istre=1:nstre
67 stres(istre)=0.0;
68 for jstre=1:nstre
69 stres(istre)=stres(istre)+dmatx
    (istre,jstre)*stran(jstre);
70 end
71 end
72
73 if(ntype == 1)
74 stres(4)=0.0
75 end
76 if(ntype == 2)
77 stres(4)=poiss*(stres(1)+stres(2));
78 end
79
80 for istre=1:nstre+1
81 elem_stres(ielem,kgaspl,istre)
    =stres(istre);
82 end
83
84 end%igaus
85 end%jgaus
86
87 end%ielem
88
89 end%endfunction

```

Line numbers:

6–12:	Change the order of numerical integration for three-node isoparametric elements.
14–87:	Loop over elements.
18–20:	Calculate the elasticity matrix of the current element (ielem).
22–33:	Define the displacement values and the Cartesian coordinates of the nodes that are associated with the current element (ielem).
34–85:	Evaluate the strains and stresses at the integration points of the current element (ielem).
40:	Integration point counter.
41–42:	Determine the position of the integration points.
48:	Calculate the shape function values and their derivatives for the current integration point.
49:	Calculate the Cartesian derivatives of shape functions, determinant of the Jacobian matrix, and Cartesian coordinates of the current integration point.
50:	Calculate the strain matrix.

52–62:	Calculate the strains at the current integration point using strain matrix and element nodal displacement values (Eq. 6.36).
64–78:	Calculate the stress values at the current integration points using elasticity matrix and strains (Eq. 6.38).
76–78:	Calculate the σ_{33} values depending on plane-stress or plane-strain solution.
80–82:	Accumulate gauss point stresses in array elem_stres.

Function**output.m**

This function prints out the solution results to file in tabulated form and in vtk file format for viewing the results by using Paraview.

The function makes calls to the following functions:

- **write_vtk_fem.m**

Variable and array list:

npoin:	Total number of nodes in the solution.
nelem:	Total number of elements in the solution.
nnode:	Number of nodes per element.
ndofn:	Number of degree of freedom per node.
ngaus:	The order of numerical integration.
nstre:	Number of stress components.
asdis(ntotv):	Global displacement vector, ntotv = npoin \times ndofn.
lnods(nelem, nnode):	Element connectivity list.
coord(npoin, ndime):	Cartesian coordinates of the nodes.
elem_stres(nelem, mgaus,nstre):	Element stress values at integration points.

Listing:

```

1 function[ ] =output_fem(npoin,
2 nelem,nnode,lnods,coord,ndofn,%
3 ngaus,nstre,asdis,elem_stres)
4 format long;
5
6 global out;

```

```

7   elem_stres(ielem,kgasp,3),elem_stres
8   fprintf(out,' \n');
9   fprintf(out,'*****\n');
10  fprintf(out,'* OUTPUTS *\n');
11  fprintf(out,'*****\n');
12  \n';
13 %--- Print Nodal Displacements:
14
15  fprintf(out,' \n');
16  fprintf(out,'Displacements:\n');
17  fprintf(out,'Node Number X-Disp Y-Disp
18  \n');
19  for ipoin=1:npoint
20  fprintf(out,'%5d',ipoin);
21  for idofn=1:ndofn
22  itotv=(ipoin-1)*ndofn+idofn;
23  fprintf(out,'%14.6e',asdis(itotv));
24  end
25  fprintf(out,' \n');
26  end
27
28 %--- Print Stress values:
29
30  fprintf(out,' \n');
31  fprintf(out,'Stress at elements\n');
32  fprintf(out,'gauss-point sigma-11 sigma-
33  22 sigma-12 sigma-33\n');
34
35 %--- Number of integration
36  ngaus2=ngaus;
37  if(nnode == 3)
38  ngaus2=1;
39  end
40
41  for ielem=1:nelem
42
43  fprintf(out,' \n');
44  fprintf(out,'Element No: %5d\n',ielem);
45  kgasp=0;
46  for igaus=1:ngaus
47  for jgaus=1:ngaus2
48  kgasp = kgasp +1;
49  fprintf(out,'%d %14.6e %14.6e %14.6e
50  %14.6e\n',kgasp,...);
51  elem_stres(ielem,kgasp,1),elem_stres
52  (ielem,kgasp,2),...
53  elem_stres(ielem,kgasp,3),elem_stres
54  (ielem,kgasp,4));
55  end
56  end
57
58  end %ielem
59
60 %--- prepare results for graphical output:
61
62 %-- deformed mesh:
63
64 facto = 3.0;
65
66 for ipoin=1:npoint
67 for idofn=1:ndofn
68 itotv=(ipoin-1)*ndofn+idofn;
69 disp_cord(ipoin,idofn)=coord(ipoin,
70 idofn) + facto * asdis(itotv);
71 end
72 end
73
74 %--- Extraplot stresses from integration
75 points
76 %--- and average over entire mesh
77
78 %--number of connection of nodes:
79
80 for ipoin=1:npoint
81 node_con(ipoin) = 0;
82 for ielem=1:nelem
83 for inode=1:nnode
84 lnode=lnods(ielem,inode);
85 if(lnode == ipoin);
86 node_con(ipoin)=node_con(ipoin) +1;
87 end
88 end %ipoin
89
90 %-- initialize nodal_stress
91
92 for ipoin=1:npoint
93 for istre=1:nstres
94 node_stres(ipoin,istre)=0.0;
95 end
96 end
97
98 for ielem=1:nelem
99
100 for istre=1:nstres
101 ave_stres(istre)=0.0;

```

```

102 end
103
104 kgasp = 0.0;
105 for igaus=1:ngaus
106 for jgaus=1:ngaus2
107 kgasp = kgasp +1;
108
109 for istre=1:nstre
110 ave_stres(istre)=ave_stres(istre)
+elem_stres(ielem,kgasp,istre);
111 end
112 end
113 end
114
115 for inode=1:nnode
116 lnode=lnods(ielem,inode);
117 for istre=1:nstre
118 node_stres(lnode,istre) =node_stres
(lnode,istre) + ave_stres(istre)/kgasp;
119 end
120 end
121 end%ielem
122
123 for ipoin=1:npoin
124 for istre=1:nstre
125 node_stres(ipoin,istre)=node_stres
(ipoin,istre)/node_con(ipoin);
126 end
127 end
128
129 %% switch order of element connectivity
if nnode==8
130 if(nnode == 8 )
131 for ielem=1:nelem
132 for inode=1:nnode
133 dummy(inode)=lnods(ielem,inode);
134 end
135 lnods(ielem,1)=dummy(1);
136 lnods(ielem,2)=dummy(3);
137 lnods(ielem,3)=dummy(5);
138 lnods(ielem,4)=dummy(7);
139 lnods(ielem,5)=dummy(2);
140 lnods(ielem,6)=dummy(4);
141 lnods(ielem,7)=dummy(6);
142 lnods(ielem,8)=dummy(8);
143 end
144 end%if
145
146 %% output to vtk file.
147 istep=1;

```

148	
149	write_vtk_fem(npoin,nelem,nnode,lnods, coord,istep,node_stres)
150	
151	end %endfunction
Line numbers:	
6:	Unit name of the output file.
13–26:	Write the displacement values in tabulated format to the output file.
28–58:	Write the element stress values in tabulated format to the output file.
60–146:	Prepare the results for vtk file format.
62–71:	Add displacement component values, amplified with the value of the facto, to the nodal coordinates, if viewing of deformed mesh is desired.
76–88:	Find out for each node, their number of connectivity.
90–121:	Accumulate the stress values at the nodes, from element integration points based on the element connectivity list.
123–127:	Average the nodal stress values based on their number of connections.
129–141:	If elements are eight-node isoparametric elements, rearrange the connectivity list as required by vtk file format.
149:	Write results in vtk format to be viewed by using Paraview. If the deformed mesh configuration is desired, replace the coord array with the dis_cord array which is calculated in lines 62–71.

Function

write_vtk_fem.m

This function writes the FEM mesh information and the nodal values in vtk file format to be viewed by using Paraview.

Variable and array list:

npoin:	Total number of nodes in the solution.
nelem:	Total number of elements in the solution.
nnode:	Number of nodes per element.
cont1(npoin)	The nodal values of the variables to be contour plotted.
lnods(nelem, nnode):	Nodal connectivity list of elements.
coord(npoin, ndime):	Cartesian coordinates of the nodes.

Listing:

```

1  function [ ] = write_vtk_fem(npoin,
2      nelem,nnode,lnods,coord,istep,
3      cont1)
4
5  format long;
6
7  fname=sprintf('time_%d.vtk',
8      istep);
9  out=fopen(fname,'w');
10
11 %--- start writing:
12
13 %% header
14 fprintf(out,'# vtk DataFile
Version 2.0\n');
15 fprintf(out,'time_10.vtk\n');
16 fprintf(out,'ASCII\n');
17 fprintf(out,'DATASET UNSTRUCTURED_
GRID\n');
18
19 %write nodal coordinates:
20
21 fprintf(out,'POINTS %5d float\n',npoin);
22
23 dummy=0.0;
24
25 for ipoin=1:npoin
26     fprintf(out,'%14.6f %14.6f %14.6f\n',
27         coord(ipoin,1),coord(ipoin,2),dummy);
28 end
29
30 %--- write element connectivity:
31 iconst1=nelem*(nnode+1);
32
33 fprintf(out,'CELLS %5d %5d\n', nelem,
34     iconst1);
35
36 for ielem=1:nelem
37     fprintf(out,'%5d',nnode);
38     for inode=1:nnode
39         fprintf(out,'%5d',(lnods(ielem,
40             inode)-1));
41     end
42     fprintf(out,'\n');
43 %--- write cell types:
44
45 if(nnode == 8)
46     ntype = 23;
47 end
48
49 if(nnode == 4)
50     ntype = 9;
51 end
52
53 if(nnode == 3)
54     ntype = 5;
55 end
56
57 fprintf(out,'CELL_TYPES %5d\n',nelem);
58
59 for i=1:nelem
60     fprintf(out,'%2d\n', ntype);
61 end
62
63
64 %--- write Nodal scalar & vector values:
65
66 fprintf(out,'POINT_DATA %5d\n',npoin);
67
68 %--- write stress values as scalar:
69
70 fprintf(out,'SCALARS sigma_xx
float 1\n');
71
72 fprintf(out,'LOOKUP_TABLE default\n');
73
74 for ipoin=1:npoin
75     fprintf(out,'%14.6e\n',cont1
76         (ipoin,1));
77 end
78
79 fprintf(out,'SCALARS sigma_yy float 1
80 \n');
81
82 for ipoin=1:npoin
83     fprintf(out,'%14.6e\n',cont1
84         (ipoin,2));
85 end
86
87 fprintf(out,'SCALARS sigma_xy float 1
88 \n');

```

```

88 fprintf(out,'LOOKUP_TABLE default\n');
89
90 for ipoin=1:npoint
91 fprintf(out,'%14.6e\n',cont1
92 (ipoin,3));
93 end
94
95 fclose(out);
96
97 end %endfunction

```

Line numbers:

7–8:	Open an output file.	23 13 5.774 3.333
13–18:	Header of the vtk file, these lines should not be modified.	24 14 7.217 4.167
19–27:	Write the Cartesian coordinates of the nodes.	25 15 9.238 5.333
29–41:	Write the element connectivity list in vtk format.	26 16 11.258 6.500
43–61:	For each element write its cell type.	27 17 14.289 8.250
64–92:	Write the values of the nodal variables. In this case, they are the stress components.	28 18 17.321 10.000 29 19 3.536 3.536 30 20 5.893 5.893 31 21 9.192 9.192 32 22 14.142 14.142 33 23 2.500 4.330 34 24 3.333 5.774 35 25 4.167 7.217 36 26 5.333 9.238 37 27 6.500 11.258 38 28 8.250 14.289 39 29 10.00 17.321 40 30 1.294 4.830 41 31 2.157 8.049 42 32 3.365 12.557 43 33 5.176 19.319 44 34 0.000 5.000 45 35 0.000 6.667 46 36 0.000 8.333 47 37 0.000 10.667 48 38 0.000 13.000 49 39 0.000 16.500 50 40 0.000 20.000 51 1 0 1 0.0 0.0 52 2 0 1 0.0 0.0 53 3 0 1 0.0 0.0 54 4 0 1 0.0 0.0 55 5 0 1 0.0 0.0 56 6 0 1 0.0 0.0 57 7 0 1 0.0 0.0 58 34 1 0 0.0 0.0 59 35 1 0 0.0 0.0 60 36 1 0 0.0 0.0 61 37 1 0 0.0 0.0 62 38 1 0 0.0 0.0 63 39 1 0 0.0 0.0 64 40 1 0 0.0 0.0 65 1 10000.0 0.3 66 0 3 67 1 12 8 1 68 10.0 0.0 69 10.0 0.0 70 10.0 0.0 71 4 23 19 12 72 10.0 0.0 73 10.0 0.0

Input File**mesh_1.inp*****Listing:***

1 40 9 14 2 8 2 2 3 3 1 2	52 2 0 1 0.0 0.0
2 1 1 2 3 9 14 13 12 8 1	53 3 0 1 0.0 0.0
3 2 3 4 5 10 16 15 14 9 1	54 4 0 1 0.0 0.0
4 3 5 6 7 11 18 17 16 10 1	55 5 0 1 0.0 0.0
5 4 12 13 14 20 25 24 23 19 1	56 6 0 1 0.0 0.0
6 5 14 15 16 21 27 26 25 20 1	57 7 0 1 0.0 0.0
7 6 16 17 18 22 29 28 27 21 1	58 34 1 0 0.0 0.0
8 7 23 24 25 31 36 35 34 30 1	59 35 1 0 0.0 0.0
9 8 25 26 27 32 38 37 36 31 1	60 36 1 0 0.0 0.0
10 9 27 28 29 33 40 39 38 32 1	61 37 1 0 0.0 0.0
11 1 5.000 0.000	62 38 1 0 0.0 0.0
12 2 6.667 0.000	63 39 1 0 0.0 0.0
13 3 8.333 0.000	64 40 1 0 0.0 0.0
14 4 10.667 0.000	65 1 10000.0 0.3
15 5 13.000 0.000	66 0 3
16 6 16.500 0.000	67 1 12 8 1
17 7 20.000 0.000	68 10.0 0.0
18 8 4.8300 1.294	69 10.0 0.0
19 9 8.0490 2.157	70 10.0 0.0
20 10 12.557 3.365	71 4 23 19 12
21 11 19.319 5.176	72 10.0 0.0
22 12 4.330 2.500	73 10.0 0.0

```

74 10.0 0.0
75 7      34     30 23
76 10.0 0.0
77 10.0 0.0
78 10.0 0.0

```

Line numbers:

1:	<i>Solution control parameters.</i> The order of parameters are: npoin, nelem, nvfix, ntype, nnodes, ndofn, ndime, ngaus, nstrel, nmats, nprop.
2–10:	<i>Element connectivity list,</i> lnods(nelem,nnodes) and material numbers, matno(nelem): element number, node numbers, material number.
11–50:	<i>Nodal coordinates,</i> (coord(npoin,ndime)): node number, x-coordinate, y-coordinate.
51–64:	<i>Constrained nodes,</i> (nofix(ivfix), iffix(nvfix, ndofn), fixed(nvfix,ndofn)): node number, DOF in x-direction (0 is for free, 1 is for constrained), DOF in y-direction (0 is for free and 1 is for constrained), constrain value in x-direction, constrained value in y-direction.
65:	<i>Material properties,</i> props(nmats,nprop): nmats, Young's modulus, Poisson's ratio.
66:	<i>Load control values:</i> ipload, nedge. ipload = 0 no point loading. If ipload is different from zero, give number of point loads and x and y components of the point loads. nedge is the number of loaded edges.
67:	Information about the first loaded edge: element number, node numbers of the loaded edges.
68–70:	Magnitude of the loads on the edge nodes: x-value and y-value.
71–78:	Information on other loaded edges with the same format as lines 67–70.

6. Clough R (1960) The finite element method in plane stress analysis. ASCE J Struc Div Proc. 2nd Conf 'Electronic computation', p 345
7. Zienkiewicz OC, Taylor RL, Zhu JZ (2013) The finite element method: Its basis and fundamentals, 7th edn. Butterworth-Heinemann, Oxford
8. Reddy J (2005) An introduction to the finite element method, 3rd edn. In: Series in mechanical engineering. McGraw-Hill, New York
9. Rao SS (2005) The finite element method in engineering, 5th edn. Butterworth-Heinemann, Oxford
10. Huges TJR (2000) Finite element method: linear static and dynamic finite element analysis. Dover, New York
11. Bathe KJ (1982) Finite element procedures in engineering analysis. In: Civil engineering and engineering mechanics series, Prentice-Hall, New Jersey
12. Ferreira AJM (2009) Matlab codes for finite element analysis: solids and structures. Springer, New York
13. Kwon YM, Bang J (2000) The finite element using Matlab, 2nd edn. CRC Press Taylor and Francis Group, Boca Raton
14. Kattan P (2007) Matlab guide to finite elements an interactive approach, 2nd edn. Springer, New York
15. Bartels S, Carstensen C, Hecht A (2006) P2QIso2D = 2D isoparametric FEM in Matlab. J Comput Appl Math 192:219
16. Alberly J, Cartensen C, Funken SA (1999) Remarks around 50 lines of Matlab: Short finite element implementation. Num Algor 20:117
17. Alberly J, Cartensen C, Funken SA, Klose R (2002) Matlab implementation of the finite element method in elasticity. Computing 69:239
18. Hinton E, Owen DRJ (1980) An introduction to finite element computations. Pineridge Press
19. Hinton E, Owen DRJ (1980) Finite element programming. Academic, London
20. Owen DRJ, Hinton E (1980) Finite elements in plasticity. Pineridge Press

References

1. Hreinkoff A (1941) Solutions of problems in elasticity by the framework method. J Appl Mech Trans ASME 8:169
2. McHenry D (1943) A lattice analogy for the solution of plane-stress problems. J Inst Civ Eng 21:59
3. Courant R (1943) Variational methods for the solution of problems of equilibrium and vibration. Bull Amer Math Soc 49:1
4. Argyris JH, Kelsey S (1960) Energy theorems and structural analysis, Butterworth Sci Publ
5. Turner M, Clough R, Martin H, Toppl LJ (1956) Stiffness and deflection analysis complex structures. J Aeronaut Sci 23:805

6.6 Case Study-XI**Simulation of spinodal decomposition of a binary alloy with finite element method****Objectives:**

The objective of this case study is to demonstrate a numerical implementation of the finite element method, FEM, for the solution of conserved Cahn–Hilliard equation in phase-field modeling.

6.6.1 Background

The background information regarding this case study has already been described in *Case Study-I* in which the Cahn–Hilliard equation was solved by utilizing finite difference algorithm. For convenience, the phase-field model is briefly summarized here again.

6.6.2 Phase-Field Model

The evolution equation for the Cahn–Hilliard equation (Chap. 1, Eq. 1.1) can be expressed as:

$$\frac{\partial c}{\partial t} = \nabla \cdot M \nabla \left(\frac{\delta F}{\delta c} \right) \quad (6.51)$$

Recall Eqs. 4.12 and 4.13, the total free energy, F , in its simplest form is taken as:

$$F = \int_V \left[f(c) + \frac{1}{2} \kappa (\nabla c)^2 \right] dv \quad (6.52)$$

and the chemical energy $f(c)$ is described as simple double-well potential:

$$f(c) = Ac^2(1 - c)^2 \quad (6.53)$$

in which κ is the gradient energy coefficient and A is a positive constant and controls the magnitude of the energy barrier between two equilibrium phases (see Fig. 4.3).

6.6.3 FEM Implementation

By rearranging Eq. 6.51, it reads

$$\begin{aligned} \frac{\partial c}{\partial t} - \nabla \cdot M \left(\nabla \left(\frac{\partial f}{\partial c} - \kappa \nabla^2 c \right) \right) &= 0 \quad \text{in } V \\ M \left(\nabla \left(\frac{\partial f}{\partial c} - \kappa \nabla^2 c \right) \right) &= 0 \quad \text{on } \partial V \\ M \kappa \nabla c \cdot n &= 0 \quad \text{on } \partial V \end{aligned} \quad (6.54)$$

where V is the volume, ∂V is the boundary, and n is the outward normal vector to the boundary.

As can be seen, Eq. 6.54 is a parabolic equation, involving first-order time derivative,

second- and fourth-order spatial derivatives. Directly casting it into weak form results in the presence of second-order spatial derivatives. This requires use of special finite elements with higher order shape functions such as cubic Hermite elements with C^1 continuity, adding to the computational effort significantly [1].

Alternatively, Eq. 6.54 can be rearranged as two coupled second-order equations as given below [2–4]:

$$\begin{aligned} \frac{\partial c}{\partial t} - \nabla \cdot M \nabla \mu &= 0 \quad \text{in } V \\ \mu - \frac{\partial f}{\partial c} + \kappa \nabla^2 c &= 0 \quad \text{in } V \end{aligned} \quad (6.55)$$

The unknowns now become c and μ . Thus, Eq. 6.55 permits the use of standard isoparametric elements with very high computational efficiency and sufficient accuracy [1]. Utilizing the tests functions η for c and ζ for μ , the weak form of Eq. 6.55 takes the form:

$$\begin{aligned} \int_V \frac{\partial c}{\partial t} \eta dV + \int_V M \nabla \mu \cdot \nabla \eta dV &= 0 \\ \int_V \mu \zeta dV - \int_V \frac{\partial f}{\partial c} \zeta dV - \int_V \kappa \nabla c \cdot \nabla \zeta dV &= 0 \end{aligned} \quad (6.56)$$

Applying implicit Euler time integration scheme to the coupled equations in Eq. 6.56 results in

$$\begin{aligned} \int_V \frac{c^{n+1} - c^n}{dt} \eta dV + \int_V M \nabla \mu^n \cdot \nabla \eta dV &= 0 \\ \int_V \mu^{n+1} \zeta dV - \int_V \frac{\partial f^{n+1}}{\partial c} \zeta dV - \int_V \kappa \nabla c^{n+1} \cdot \nabla \zeta dV &= 0 \end{aligned} \quad (6.57)$$

where dt is the time increment, $dt = t^{n+1} - t^n$.

6.6.4 Numerical Implementation

By taking the nodal variables as c and μ and utilizing the shape functions of isoparametric elements, the values of c and μ can be determined at any Cartesian point within an element as:

$$c = \sum_i^n N_i c_i \quad \text{and} \quad \mu = \sum_i^n N_i \mu_i \quad (6.58)$$

where n is number of nodes of that element and N_i , c_i , and μ_i are the nodal values of shape

$$R_i^e = \int_V \left[(c^{n+1} - c^n)N_i + dtM\mu^{n+1} \left(\frac{\partial N_i}{\partial x} \frac{\partial N_j}{\partial y} \right)^T \cdot \left(\frac{\partial N_j}{\partial x} \frac{\partial N_j}{\partial y} \right) N_j \right. \\ \left. \mu^{n+1}N_i - \frac{\partial f^{n+1}}{\partial c} N_i - \kappa c^{n+1} \left(\frac{\partial N_i}{\partial x} \frac{\partial N_i}{\partial y} \right)^T \cdot \left(\frac{\partial N_j}{\partial x} \frac{\partial N_j}{\partial y} \right) N_j \right] dV \quad (6.59)$$

in which $(\partial N / \partial x \partial N / \partial y)$ are the derivatives of shape function in Cartesian coordinates and T stands for the transpose.

Corresponding element stiffness to Eq. 6.59 is:

$$K_{ij}^e = \frac{\partial R_i^e}{\partial c \text{ or } \partial \mu} = \begin{bmatrix} K_{ij}^{cc} & K_{ij}^{c\mu} \\ K_{ij}^{\mu c} & K_{ij}^{\mu\mu} \end{bmatrix} \quad (6.60)$$

and its components are calculated as:

$$K_{ij}^{cc} = \int_V [N_i^T \cdot N_j] dV \quad (6.61)$$

$$K_{ij}^{c\mu} = \int_V \left[dtM \left(\frac{\partial N_i}{\partial x} \frac{\partial N_i}{\partial y} \right)^T \cdot \left(\frac{\partial N_j}{\partial x} \frac{\partial N_j}{\partial y} \right) \right] dV \quad (6.62)$$

$$K_{ij}^{\mu c} = \int_V \left[-\frac{\partial^2 f}{\partial c^2} N_i^T N_j - \kappa \left(\frac{\partial N_i}{\partial x} \frac{\partial N_i}{\partial y} \right) \cdot \left(\frac{\partial N_j}{\partial x} \frac{\partial N_j}{\partial y} \right) \right] dV \quad (6.63)$$

$$K_{ij}^{\mu\mu} = \int_V [N_i^T N_j] dV \quad (6.64)$$

After forming element stiffness and residuals, they are assembled into the global stiffness and right-hand side vectors, and the resulting set of nonlinear equation is

$$[K^G]\{d\delta\} = \{R^G\} \quad (6.65)$$

functions, c and μ , respectively. By utilizing Eqs. 6.58 in 6.57, the nodal residual vector can be expressed as:

Note that the assembly process is carried out in such a way that the unknown vector $d\delta$ in Eq. 6.65 takes the form below. This is just a numerical convince for bookkeeping purposes.

$$d\delta = (dc_1, dc_2, \dots, dc_N, \dots, d\mu_1, d\mu_2, \dots, d\mu_N) \quad (6.66)$$

where N is the total number of nodes in the FEM mesh, c is the concentration values at the nodes, and μ is the auxiliary variable values at the nodes.

Equation 6.65 is set of nonlinear equation; therefore, at each time increment step, it is solved iteratively with the modified Newton-Raphson scheme, in order to gain computational efficiency. In this modified version of Newton-Raphson algorithm, the left-hand global stiffness matrix K^G is formed only once at the beginning of each time step increment. In the following iteration steps the rhs residual vector, R^G , reevaluated and Eq. 6.65 solved again. This iterative solution process continues until desired accuracy is reached. Thus, the steps in the algorithm are:

Step-1

For each time increment:

Form the global stiffness matrix, $[K^G]$, in Eq. 6.65 only once.

Step-2

Newton-Raphson iterations:

Form the right-hand side residual vector, $\{R^G\}$, in Eq. 6.65

Solve Eq. 6.65

Update $\delta^{n+1} = \delta^n + d\delta$, where n is the time step number.

Check convergence, if it satisfied go to step-1, else repeat step-2.

6.6.5 Results and Discussion

All the parameters used in these simulations were identical to that used for *Case Studies-I and IV*. The simulations were carried out by using two different FEM meshes. In the first simulation, the FEM mesh is composed of triangular isoparametric elements and there were 1681 nodes and 3200 elements. The element integration was carried out by using Radau rule and the total integration points per element was 6. In the second simulation the mesh is composed of four-node isoparametric elements and, for this case, there were 1681 nodes and 1600 elements. The integration of elements was carried out with Gauss-Legendre rule and the total integration points per element was 9. The time evolution of the microstructure obtained from these simulations are shown in Figs. 6.3 and 6.4.

Similar to that seen earlier, *Case Studies-I and VI*, initially (at time step 750) the microstructure is relatively fine and contains a large number of small precipitates. As time progresses, coarsening of the second phase through migration of the phase boundaries, dissolution, merging, and breakup can be easily inferred from the figures. As can be seen, the growth mainly occurs with Ostwald ripening process in which the small precipitates dissolve and are absorbed by the larger ones, and as a result the number of the precipitates becomes smaller with time.

The animation files summarizing these simulations together with the FEM mesh input files are given in subdirectory, *case_study_11*, of the downloadable file.

6.6.6 Source Codes

Program

fem_ch_v1_2.m

This is the main program to solve Cahn-Hilliard phase-field equation with FEM algorithm. Depending on the selection of the isolve parameter in the code: For isolve = 1, the code executes in un-optimized in longhand format mode. For isolve = 2, execution is for Matlab/Octave optimized mode. Therefore, for the desired execution mode, this parameter in line 21 should be modified in the program.

The program makes calls to the following functions:

- **input_fem_pf.m**
- **periodic_boundary.m**
- **gauss.m**
- **cart_deriv.m**
- **init_micro_ch_fem.m**
- **chem_stiff_v1.m**
- **chem_stiff_v2.m**
- **apply_periodic_bc.m**
- **recover_slave_dof.m**
- **write_vtk_fem.m**

Listing:

```

1   %%%%%%%%%%%%%%%%
2   %                                     %
3   % FEM PHASE-FIELD CODE FOR SOLVING %
4   % CAHN-HILLIARD EQUATION           %
5   %                                     %
6   %%%%%%%%%%%%%%%%
7
8   % get wall time:
9
10  time0=clock();
11  format long;
12
13  %--
```

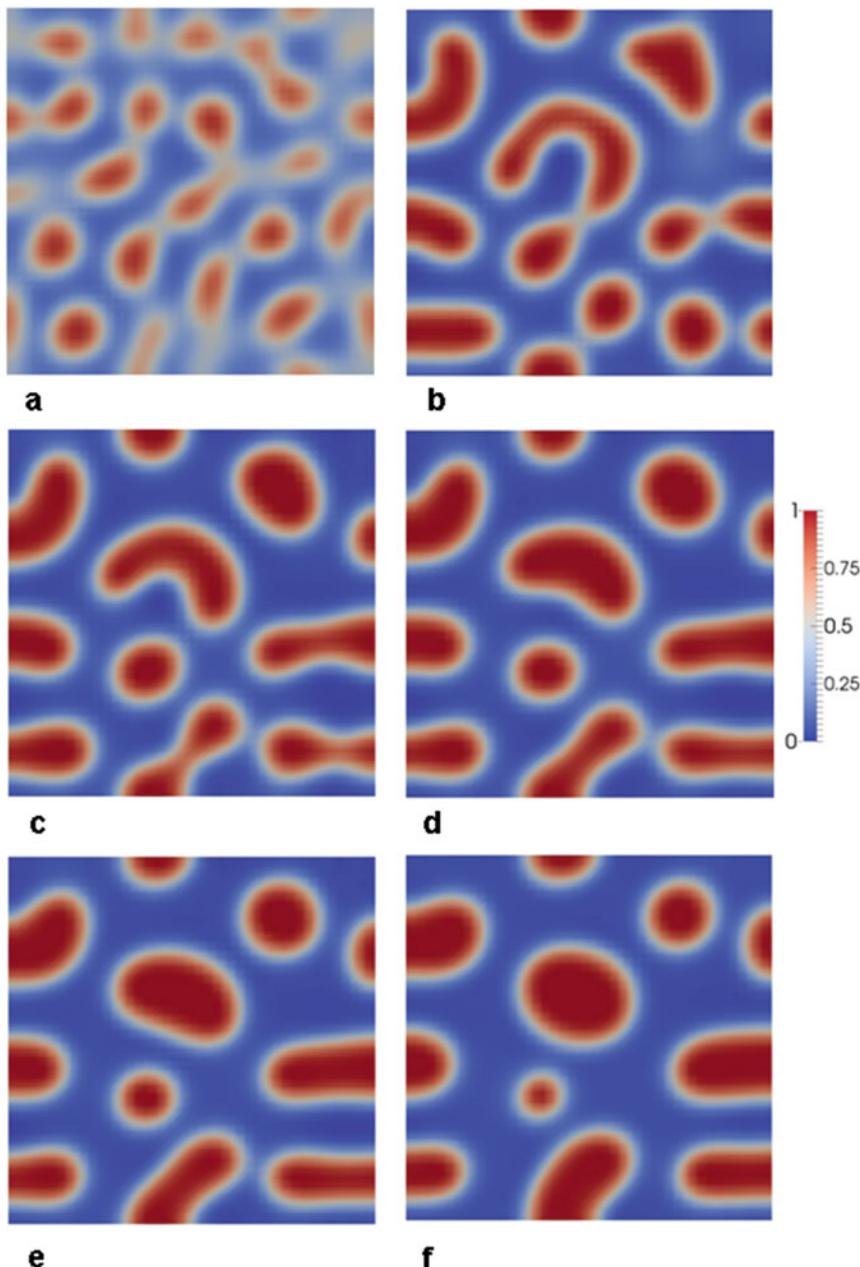


Fig. 6.3 Time evolution of microstructure obtained from the simulation with tri-node isoparametric elements. Time steps in the figure are: (a) 750, (b) 1875, (c) 2500, (d) 3125, (e) 3750, and (f) 5000, respectively

```

14 global in;
15 in=fopen('mesh3n_40.inp','r');
16
17 global out;
18 out=fopen('result_1.out','w');
19 %--
20
21 isolve=2;
22
23 %--- Time integration parameters:
24
25 nstep= 5000;
26 nprnt= 25;
27 dtime= 2.0e-2;

```

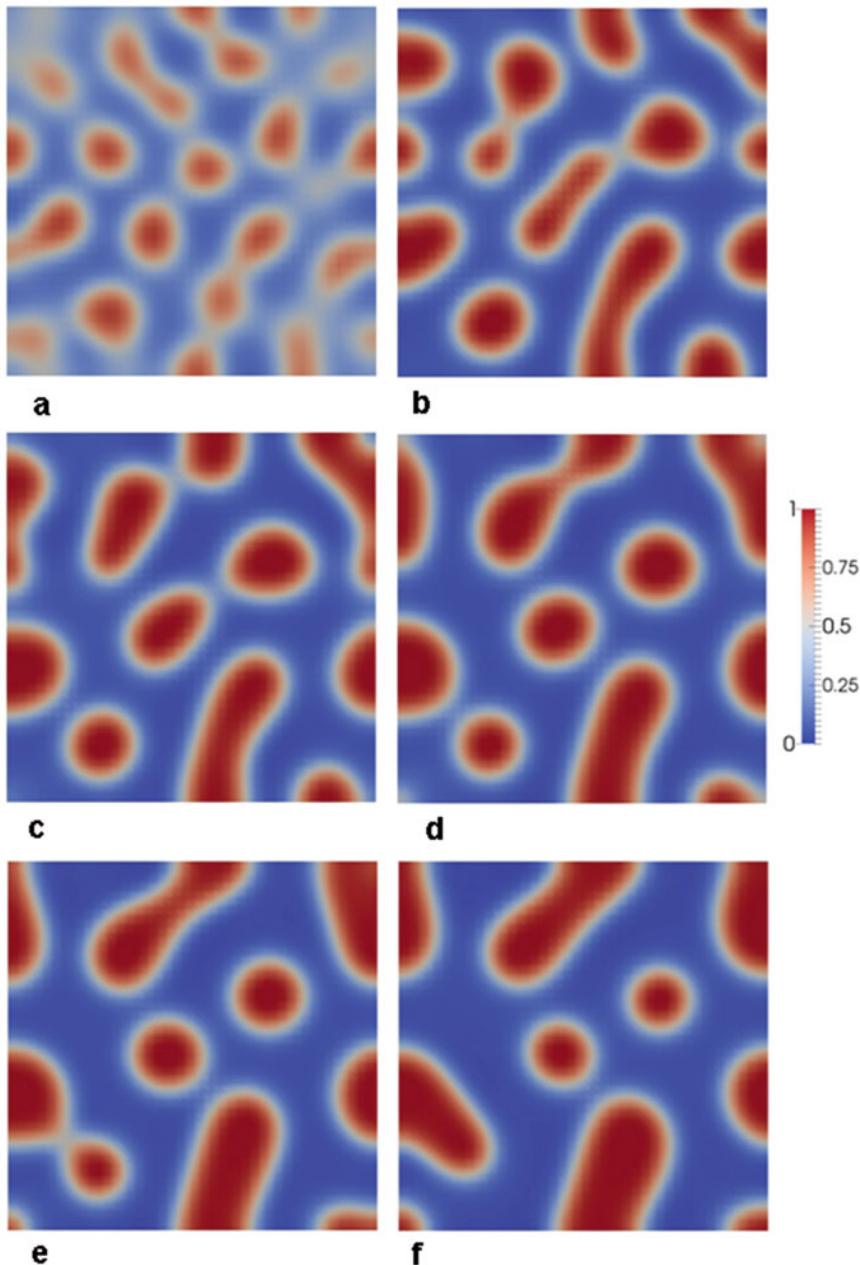


Fig. 6.4 Time evolution of microstructure obtained from the simulation with four-node isoparametric elements. Time steps in the figure are: (a) 750, (b) 1875, (c) 2500, (d) 3125, (e) 3750, and (f) 5000, respectively

```
28 toler= 5.0e-5;          34 mobil= 1.0;
29 miter= 10;              35 grcoef= 0.5;
30                         36
31 %--- Material specific parameters: 37 %-----
32                         38 %input data:
33 conc0= 0.40;            39 %-----
```

```

40                                         83      ndofn,ngaus,ntype,lnods,
41 [npoin,nelem,ntype,nnode,ndofn,          84      coord, ...
42 ndime,ngaus,...                         85      mobil,grcoef,con,con_old,
43 nstre,nmats,nprop,lnods,                 86      dtime, ...
44 matno,coord]                            87      posgp,weigp,istep,iter,
45 =input_fem_pf();                         88      gstif);
46                                         89      end
47                                         90      %
48                                         91      if(isolve == 2)
49                                         92      [gstif,gforce]=chem_stiff_v2
50                                         93      (npoin,nelem,nnode,nstre,ndime, ...
51                                         94      ndofn,ngaus,ntype,lnods,
52 neaus,ntype,lnods,coord,posgp,           95      coord, ...
53 weigp);                                96      mobil,grcoef,con,con_old,
54                                         97      dtime, ...
55                                         98      posgp,weigp,dgdx,dvolum,
56                                         99      istep, ...
57                                         100     iter,gstif);
58                                         101    end
59                                         102   %-----
60                                         103   % Rearrange gstif & gforce for PBC
61 [con] = init_micro_ch_fem(npoin,         104
62 ndofn,conc0);                          105   %-----
63                                         106   % solve equations and update   &
64 %          EVOLVE          & 107   %-----&
65 %-----& 108
66                                         109   asdis = gstif\gforce;
67 for istep=1:nstep                      110
68                                         111   %-----
69 con_old=con;                           112   %Recover slave node values
70                                         113   %-----
71 %-----& 114
72 %--Newton iteration:                  115   [asdis] =recover_slave_dof(asdis,
73 %-----& 116   ncountm,ncounts, ...
74 for iter = 1: miter                   117   master,slave,npoin,ndofn);
75                                         118   %-- update concentration field
76 if(iter == 1)                          119
77 gstif=sparse(ntotv,ntotv);            120   con = con + asdis';
78 end                                     121
79                                         122   %-- for small deviations:
80 if(isolve == 1)                        123
81                                         124   if(isolve == 1)
82 [gstif,gforce]=chem_stiff_v1        (npoin,nelem,nnode,nstre,ndime, ...

```

```

125
126 for ipoin=1:npoint
127 if(con(ipoin) <= 0.0001)
128 con(ipoin)=0.0001;
129 end
130 if(con(ipoin) >= 0.9999)
131 con(ipoin)=0.9999;
132 end
133 end
134
135 else
136 inrange=(con > 0.9999);
137 con(inrange) = 0.9999;
138
139 inrange=(con < 0.0001);
140 con(inrange) = 0.0001;
141 end
142
143
144 %--- check norm for convergence
145
146 normF = norm(gforce, 2);
147
148 if(normF <= toler)
149 break
150 end
151
152 end %end of Newton
153
154 %--- print out
155
156 if(mod(istep,nprnt) == 0)
157
158 fprintf('Done step: %5d\n',istep);
159
160 %fname=sprintf('time_%d.out',istep);
161 %out1=fopen(fname,'w');
162
163 %for ipoin=1:npoint
164 %fprintf(out1,'%14.6e%14.6e%14.6e\n',
165 %coord(ipoin,1), coord(ipoin,2),
166 %con(ipoin));
167 %fclose(out1);
168
169 write_vtk_fem(npoint,nelem,nnode,
170 lnods,coord,istep,con);
171
172 end%if
173

```

```

174 end%istep
175
176 compute_time=etime(clock(),time0)
177
178 fprintf(out,'compute time: %7d\n',
179 compute_time);

```

Line numbers:

10:	Get initial wall clock time beginning of the execution.
13–19:	Assign unit names for input and output files.
21:	Solution flag, isolve = 1 for un-optimized execution mode, isolve = 2 for Matlab/Octave optimized mode.
23–30:	Time integration parameters.
25:	Number of time increment steps.
26:	Print frequency to output the results to file.
27:	Time increment for numerical integration.
28:	Tolerance value for iterative solution.
29:	Number of maximum iterations.
31–36:	Material-specific parameters.
33:	Alloy concentration.
34:	Mobility parameter.
35:	Gradient energy coefficient.
41–42:	Input FEM mesh and control parameters.
44:	Determine the master and slave nodes for periodic boundary condition.
46:	Parameters for numerical integration of elements.
48–52:	If the execution is in optimized mode, precalculate the Cartesian derivatives of shape functions and area/volume for all elements in the solution.
54:	Total number of variables in the solution.
55:	Total number of variables per element.
58–61:	Modulate the microstructure and initialize the vector containing nodal variables.
64–174:	Evolve microstructure.
69:	Assign nodal variables from previous time step.
72–152:	Modified Newton–Raphson iterations.
76–78:	If iter = 1, initialize the global stiffness matrix.
80–86:	If execution is un-optimized mode (isolve = 1), form the global stiffness matrix for only in first iteration and form rhs, load, vector in every following iterations.
89–96:	If execution is optimized mode (isolve = 2), form the global stiffness matrix for only in first iteration and form rhs, load, vector in every following iterations.
99–104:	Modify the global stiffness matrix and rhs, load, vector for periodic boundaries.
106–110:	Solve the resulting linear equations.

(continued)

112–117:	Recover the nodal values for the slave nodes in the periodic boundaries.
118–121:	Update the nodal variable vector.
122–142:	For small deviations from max and min values, reset the limits.
124–134:	For un-optimized mode.
135–140:	For optimized mode.
144–150:	Check convergence. If converged solution is reached, exit from Newton–Raphson iteration.
156–172:	If print frequency is reached, output the results to file.
160–167:	Open an output file and print the coordinates and the nodal values to file. These lines are commented out, but they can be changed, including the output format.
169:	Write the results in vtk format for contour plots to be viewed by Paraview.
176–178:	Compute the total execution time and print.

Function

init_micro_ch_fem.m

This function modulates the initial microstructure and initializes nodal variable list, according to Eq. 6.66.

Variable and array list:

npoint:	Total number of nodes in the solution.
ndofn:	Number of DOF per node.
conc0:	Alloy concentration.
nodco (ntotv):	Nodal variable list, $ntotv = npoin \times ndofn$.

Listing:

```

1 function [nodco] = init_micro_ch_fem
2   (npoin,ndofn,conc0)
3   format long;
4
5   npoin2 = 2*npoin;
6   nodcon=zeros(npoin2);
7
8   noise = 0.02;
9
10  for ipoin=1:npoin
11
12    nodco(ipoin)=conc0+noise*
13      (0.5-rand);
```

```

13  nodco(ipoin+npoin)=0.0;
14
15  end
16
17  end %endfunction
```

Line numbers:

5:	Total number of variables in the solution.
6:	Initialize the nodal variable vector.
12:	Modulate the composition field.

Function

apply_periodic_bc.m

This function modifies the global stiffness matrix and rhs, load, vector to impose the periodic boundary conditions to FEM mesh. The nodal information is determined in function *periodic_boundary.m*. Note this algorithm only works if mesh is uniform and the Cartesian coordinates of the boundary nodes also have symmetry properties. The algorithms to impose periodic boundary conditions for arbitrary FEM meshes are given in [5–7].

Variable and array list:

ncountm:	Number of master nodes.
ncounts:	Number of slave nodes.
ndofn:	Number DOF per node.
npoin:	Total number of nodes in the solution.
iter:	Iteration number in Newton–Raphson algorithm.
master (ncountm):	Array containing the node numbers of master nodes.
slave (ncounts):	Array containing the node numbers of slave nodes.
gforce (ntotv):	Global rhs, load, vector.
gstif(ntotv, ntotv):	Global stiffness matrix ($ntotv = npoin \times ndofn$).

Listing:

```

1 function [gstif,gforce] =
2   apply_periodic_bc(ncountm,ncounts,
3   master,slave,...)
4   ndofn,npoin,gstif,gforce,iter)
5   format long;
```

```

6 for ipbc=1:nountm
7
8 im=master(ipbc);
9 is=slave(ipbc);
10
11 if(iter==1)
12 %%-- Add rows:
13 gstif(im,:)=gstif(im,:)+
14 gstif(is,:);
15 %%-- Add columns:
16 gstif(:,im)=gstif(:,im)+
17 gstif(:,is);
18 %%-- zero slave dofs
19
20 gstif(is,:)=0.0;
21 gstif(:,is)=0.0;
22
23 gstif(is,is)=1.0;
24
25 end
26
27 %%-- add rhs
28 gforce(im)=gforce(im)+gforce(is);
29 gforce(is)=0.0;
30
31 end
32
33 end %endfunction
34

```

Line numbers:

6–31:	Loop over number of master nodes.
11–25:	If iter = 1, add the row and columns of global stiffness matrix values of the slave nodes to the rows and columns of corresponding master nodes.
12–16:	The addition of rows and columns.
18–21:	Zero the row and columns of slave nodes.
23:	Insert value of one to the diagonals of slave nodes.
28:	Add the rhs vector values of the slave nodes to the corresponding rhs values of the master nodes.
29:	Zero the rhs vectors of slave nodes.

Function**cart_deriv.m**

This function precalculates the Cartesian derivatives of shape functions at all integration

of points of all elements in the simulation. In addition, function also calculates the area/volume contributions of the integration points.

The function makes calls to the following functions:

- **sfr2.m**
- **jacob3.m**

Variable and array list:

npoin:	Total number of nodes in the solution.
nelem:	Total number of elements in the solution.
nnode:	Number of nodes per element.
nstre:	Number of stress components.
ndime:	Number of Cartesian coordinate dimension.
ndofn:	Number DOF per node.
ngaus:	The order of numerical integration.
ntype:	Solution type (ntype = 1, plane-stress and ntype = 2 plane-strain).
posgp (ngaus):	Position of sampling points.
weigp(ngaus):	Weighting factors at the sampling points.
lnods(nelem, nnode):	Element nodal connectivity list.
coord(npoin,ndime):	Cartesian coordinates of nodes.
dvolum(nelem, mgaus):	Contributions of integration points to element area/volume.
dgdx(nelem,mgaus, ndime,nnode):	Cartesian derivatives of shape functions at integration points.

Listing:

```

1 function [dgdx,dvolum] = cart_deriv
2   (npoin,nelem,nnode,nstre,ndime,ndofn, ...
3    ngaus,ntype,lnods,coord,posgp,
4    weigp)
5
6 format long;
7
8 ngaus2 = ngaus;
9 if(nnode == 3)
10 ngaus2 = 1;
11 end
12
13 ngaus = ngaus*nngaus2;
14
15 dvolum = zeros(nelem, ngaus);
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
778
779
779
780
781
782
783
784
785
786
787
787
788
788
789
789
790
791
792
793
794
795
796
797
797
798
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
817
818
818
819
819
820
821
822
823
824
825
826
827
827
828
828
829
829
830
831
832
833
834
835
836
837
837
838
838
839
839
840
841
842
843
844
845
846
846
847
847
848
848
849
849
850
851
852
853
854
855
856
856
857
857
858
858
859
859
860
861
862
863
864
865
866
866
867
867
868
868
869
869
870
871
872
873
874
875
875
876
876
877
877
878
878
879
879
880
881
882
883
884
885
885
886
886
887
887
888
888
889
889
890
891
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
901
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
911
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
921
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
931
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
941
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
951
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
961
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
971
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
981
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1011
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1021
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1031
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1041
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1051
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1061
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1071
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1081
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1541
1542
1542
1543
1543
1544
1544
1545
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1551
1552
1552
1553
1553
1554
1554
1555
1555
1556
1556
1557
1557
1558
1558
1559
1559
1560
1561
1562
1562
1563
1563
1564
1564
1565
1565
1566
1566
1567
1567
1568
1568
1569
1569
1570
1571
1572
1572
1573
1573
1574
1574
1575
1575
1576
1576
1577
1577
1578
1578
1579
1579
1580
1581
1582
1582
1583
1583
1584
1584
1585
1585
1586
1586
1587
1587
1588
1588
1589
1589
1590
1591
1592
1592
1593
1593
1594
1594
1595
1595
1596
1596
1597
1597
1598
1598
1599
1599
1600
1601
1602
1602
1603
1603
1604
1604
1605
1605
1606
1606
1607
1607
1608
1608
1609
1609
1610
1611
1612
1612
1613
1613
1614
1614
1615
1615
1616
1616
1617
1617
1618
1618
1619
1619
1620
1621
1622
1622
1623
1623
1624
1624
1625
1625
1626
1626
1627
1627
1628
1628
1629
1629
1630
1631
1632
1632
1633
1633
1634
1634
1635
1635
1636
1636
1637
1637
1638
1638
1639
1639
1640
1641
1642
1642
1643
1643
1644
1644
1645
1645
1646
1646
1647
1647
1648
1648
1649
1649
1650
1651
1652
1652
1653
1653
1654
1654
1655
1655
1656
1656
1657
1657
1658
1658
1659
1659
1660
1661
1662
1662
1663
1663
1664
1664
1665
1665
1666
1666
1667
1667
1668
1668
1669
1669
1670
1671
1672
1672
1673
1673
1674
1674
1675
1675
1676
1676
```

```

15 dgdx=zeros(nelem,mgaus,ndime,
    nnodes);
16
17 for ielem=1:nelem
18 for inode=1:nnodes
19 lnode=lnods(ielem,inode);
20 for idime=1:ndime
21 elcod(idime,inode)=coord(lnode,
    idime);
22 end
23 end
24
25 %==== gauss points:
26
27 kgasp=0;
28 for igaus=1:nGAUS
29 exisp=posgp(igaus);
30 for jgaus=1:nGAUS2
31 etasp=posgp(jgaus);
32 if (nnode==3)
33 etasp=posgp(nGAUS+igaus);
34 end
35
36
37 kgasp=kgasp+1;
38 nGAUS=nGAUS+1;
39
40 [shape,deriv]=sfr2(exisp,etasp,
    nnodes);
41 [cartd,djacb,gpcod]=jacob3(ielem,
    elcod,kgasp,shape,deriv,nnode,
    ndime);
42
43 dvolu=djacb*weigp(igaus)*weigp
    (jgaus);
44
45 if (nnode==3)
46 dvolu=djacb*weigp(igaus);
47 end
48
49 dvolum(ielem,kgasp)=dvolu;
50
51 for idime=1:ndime
52 for inode=1:nnodes
53 dgdx(ielem,kgasp,idime,inode)=cartd
    (idime,inode);
54 end
55 end
56
57 end%igaus

```

```

58 end%jgaus
59 end%ielem
60
61 end%endfunction
62

```

Line numbers:

6–9:	The order of numerical integration.
11:	Total number of integration points in the elements.
13:	Initialize the area contributions of integration points to the element area/volume.
15:	Initialize Cartesian derivatives of shape functions at the integration points of element.
17–59:	Loop over number of elements in the solution.
18–23:	Calculate the coordinates of the element nodes.
25–58:	Numerical integration of elements.
29, 31, 33:	Determine the position of integration points.
37–38:	Increase the counters of integration points.
40:	Calculate the shape functions, their derivative values for the current integration point.
41:	Calculate the Cartesian derivatives of the shape functions, the determinant of the Jacobian and the Cartesian coordinates of the integration point (Eq. 6.5).
43–47:	Calculate the area/volume contribution of the integration point (Eq. 6.7).
49:	Assemble the element area/volume values into global array dvolum.
51–55:	Assemble the Cartesian derivative values in global array dgdx.

Function**chem_stiff_v1.m**

This function forms the global stiffness and global rhs, load, vector for solution of Cahn–Hilliard Equation for the modified Newton–Raphson solution algorithm. It is in longhand format and is not optimized for Matlab/Octave.

The function makes calls to the following functions:

- **sfr2.m**
- **jacob2.m**
- **free_energ_fem_v1**

Variable and array list:

npoin:	Total number of nodes in the solution.
nelem:	Total number of elements in the solution.
nnode:	Number of nodes per element.
nstre:	Number of stress components.
ndime:	Number of Cartesian coordinate dimension.
ndofn:	Number DOF per node.
ngaus:	The order of numerical integration.
ntype:	Solution type (ntype = 1, for plane-stress and ntype = 2 for plane-strain).
mobil:	Mobility coefficient.
grcoef:	Gradient energy coefficient.
dtime:	Time increment used in time integration.
istep:	Time increment step number.
iter:	Newton–Raphson iteration number.
con(ntotv):	Nodal values, which includes c and μ values, $ntotv = npoin \times ndofn$
con_old (ntotv):	Nodal values from previous time step.
lnods(nelem, nnode):	Element nodal connectivity list.
coord(npoin, ndime):	Cartesian coordinates of nodes.
posgp (ngaus):	Position of sampling points.
weigp(ngaus):	Weighting factors at the sampling points.
gstif(ntotv, ntotv):	Global stiffness matrix.
gforce(ntotv):	Global rhs, load, vector.

Listing:

```

1 function [gstif,gforce]=chem_stiff
2     _v1(npoin,nelem,nnode,nstre,
3         ndime, ...
4             ndofn,ngaus,ntype,lnods,
5             coord, ...
6             mobil,grcoef,con,con_old,
7             dtime, ...
8             posgp,weigp,istep,iter,
9             gstif)
10    format long;
11
12    %=====
13    % global and local variables:      &
14    %=====
15    ntotv=npoin*ndofn;
16    nevab=nnode*ndofn;
17
18    %=====
19    ngaus2=ngaus;
20    if(nnode == 3)
21        ngaus2=1;
22    end
23    gforce=zeros(ntotv,1);
24    for ielem=1:nelem
25        %=====
26        % initialize elements stiffness %& rhs: %
27        %=====
28        if(iter == 1)
29            for inode=1:nnode
30                for jnode=1:nnode
31                    kcc(inode,jnode)=0.0;
32                    kcm(inode,jnode)=0.0;
33                    kmm(inode,jnode)=0.0;
34                    kmc1(inode,jnode)=0.0;
35                    kmc(inode,jnode)=0.0;
36
37                end
38            end
39        end
40    end
41    end %if
42
43    %--- rhs
44    for inode=1:nnode
45        eload1(inode) = 0.0;
46        eload2(inode) = 0.0;
47    end
48    for ievab=1:nevab
49        eload(ievab) =0.0;
50    end
51
52
53
54    %=====
55    %--- elemental values          &
56    %=====
57
58    for inode=1:nnode
59
60        lnode = lnods(ielem,inode);
61        cv(inode) = con(lnode);
62        cm(inode) = con(npoin+lnode);
63        cv_old(inode) = con_old(lnode);
64

```

```

65 end
66
67 %--- coords of the element nodes
68 for inode=1:nnode
69 lnode=lnds(ielem,inode);
70 for idime=1:ndime
71 elcod(idime,inode)=coord(lnode,
idime);
72 end
73 end
74
75 %=====
76 % integrate element stiffness %
77 % & rhs: %
78 %=====
79
80 kgasp=0;
81
82 for igaus=1:ngaus
83 exisp=posgp(igaus);
84 for jgaus=1:ngaus2
85 etasp =posgp(jgaus);
86 if(nnode ==3)
87 etasp=posgp(ngaus+igaus);
88 end
89
90 kgasp=kgasp+1;
91 [shape,deriv]=sfr2(exisp,etasp,
nnode);
92 [cartd,djacb,gpcod]=jacob2(ielem,
elcod,kgasp,shape, ...
93 deriv,nnode,ndime);
94
95 dvolu=djacb*weigp(igaus)*weigp
(jgaus);
96
97 if(nnode == 3)
98 dvolu=djacb*weigp(igaus);
99 end
100
101 %== values at the gauss points:
102
103 cvgp = 0.0;
104 cmgp = 0.0;
105 cv_ogp = 0.0;
106
107 for inode=1:nnode
108
109 cvgp = cvgp + cv(inode)*shape
(inode);
110 cmgp = cmgp + cm(inode)*shape
(inode);
111 cv_ogp = cv_ogp + cv_old(inode)*
shape(inode);
112
113 end
114
115 %== chemical potential:
116 [dfdc,df2dc]=free_energ_fem_v1
(cvgp);
117
118
119 %
120
121 if(iter == 1)
122
123 %--- kcc matrix:
124
125 for inode=1:nnode
126 for jnode=1:nnode
127 kcc(inode,jnode) = kcc(inode,
jnode) + ...
128 shape(inode)*shape(jnode)*
dvolu;
129
130 end
131 end
132 %--- kcm matrix:
133
134 for inode=1:nnode
135 for jnode=1:nnode
136 for idime=1:ndime
137
138 kcm(inode,jnode) = kcm (inode,
jnode) + ...
139 dttime*mobil*cartd(idime,
inode)* ...
140 cartd(idime,jnode)*dvolu;
141
142 end
143 end
144
145 %--- kmm matrix:
146
147 for inode=1:nnode
148 for jnode=1:nnode
149 kmm(inode,jnode) = kmm(inode,
jnode) + ...
150 shape(inode)*shape(jnode)*
dvolu;

```

```
151 end 195 end
152 end 196 end
153 197 end
154 198
155 %--- kmc matrix: 199 for inode=1:nnode
156 for inode=1:nnode 200 eload2(inode) =eload2(inode) - ...
157 for jnode=1:nnode 201 shape(inode)*(cmgp-dfdc)
158 for idime=1:ndime 202 *dvolu;
159 203 end
160 kmc(inode,jnode) = kmc(inode, 204 for inode=1:nnode
161 jnode) - ... 205 for jnode=1:nnode
162 cartd(idime,jnode)*dvolu; 206 for idime=1:ndime
163 end 207
164 end 208 eload2(inode) =eload2(inode) + ...
165 end 209 grcoef*cvgp*shape(inode)*cartd
166 210 (idime,inode)* ...
167 for inode=1:nnode 211 cartd(idime,jnode)*dvolu;
168 for jnode=1:nnode 212 end
169 213 end
170 kmc(inode,jnode) = kmc(inode, 214
171 jnode) - ... 215 end %igaus
172 df2dc*shape(inode)* 216 end %jgaus
173 shape(jnode)*dvolu;
174 end 217
175 end %if iter 218 %-----
176 219 %--assemble element stiffness
177 220 %----- and rhs:
178 %----- 221
179 % element rhs 222 if(iter == 1)
180 %----- 223
181 224 for inode=1:nnode
182 for inode=1:nnode 225 ievab=nnode+inode;
183 eload1(inode) = eload1(inode) - ... 226 for jnode=1:nnode
184 shape(inode)*(cvgp-cv_ogp) 227 jevab=nnode+jnode;
185 *dvolu; 228 estif(inode,jnode)=kcc(inode,
186 end 229 jnode);
187 for inode=1:nnode 230 estif(inode,jevab)=kcm(inode,
188 for jnode=1:nnode 231 jnode);
189 for idime=1:ndime 232 estif(ievab,jnode)=kmc(inode,
190 233 jnode);
191 eload1(inode) = eload1(inode) 234 end
192 - dtim*mobil ... 235 end
193 *cmgp*shape(inode)*cartd 236 end %if
194 (idime,inode)* ... 237
```

```

238 %rhs
239 for inode=1:nnode
240 ievab=nnode+inode;
241 eload(inode)=eload1(inode);
242 eload(ievab)=eload2(inode);
243 end
244
245 %=====
246 %form global stiffness and rhs
247 %=====
248
249 if(iter == 1)
250
251 for idofn=1:ndofn
252 for inode=1:nnode
253 ievab = (idofn - 1)*nnode+inode;
254 for jdofn=1:ndofn
255 for jnode=1:nnode
256 jevab = (jdofn-1)*nnode+jnode;
257
258 gstif = gstif + sparse(((idofn-1)
259 *npoin+lnods(ielem,inode)), ...
260 ((jdofn-1)*npoin+lnods(ielem,
261 jnode)), ...
262 estif(ievab,jevab),ntotv,ntotv);
263 end
264 end
265
266 end %if iter
267
268 %% rhs
269
270 for idofn=1:ndofn
271 for inode=1:nnode
272 ievab=(idofn-1)*nnode+inode;
273
274 gforce = gforce +sparse(((idofn-1)
275 *npoin+lnods(ielem,inode)),1...
276 ,eload(ievab),ntotv,1);
277 end
278 end
279
280 end % ielem
281
282 end %endfunction
283

```

<i>Line numbers:</i>	
11:	Total number of variables in the solution.
12:	Total number of variables per element.
14–17:	Order of numerical integration depending on the element type.
18:	Initialize global rhs, load, vector.
19–280:	Loop over elements.
28–41:	If iter = 1, initialize element stiffness submatrices (Eq. 6.60).
44–50:	Initialize the element rhs vectors.
58–65:	Get the nodal values of the element.
67–73:	Get the Cartesian coordinates of the element nodes.
80–216:	Numerical integration of the element stiffness and rhs vector.
83, 85, 87:	Positions of the integration points.
91:	Calculate the shape functions and their derivative values for the current integration point.
92–93:	Calculate the Cartesian derivatives of shape functions, the determinant of the Jacobian and the Cartesian positions of the integration point (Eq. 6.5).
101–113:	Transfer the nodal values to the integration points using Eq. 6.58.
115–117:	Calculate the derivatives of the chemical energy for this integration point.
121–176:	If iter = 1, form the submatrices of element stiffness matrix.
125–131:	Form the submatrix, K_{ij}^{cc} (Eq. 6.61).
132–143:	Form the submatrix, $K_{ij}^{c\mu}$ (Eq. 6.62).
145–152:	Form the submatrix, $K_{ij}^{\mu\mu}$ (Eq. 6.64).
155–174:	Form the submatrix, $K_{ij}^{\mu c}$ (Eq. 6.63).
179–213:	Calculate the element rhs, load, vector.
182–185:	First term of first row of element rhs vector (Eq. 6.59).
187–197:	Second term of first row element rhs vector (Eq. 6.59).
199–202:	First and second terms of second row element rhs vector (Eq. 6.59).
204–213:	Third term of second row element rhs vector (Eq. 6.59).
222–236:	If iter = 1, assemble the element stiffness matrix from submatrices, (Eq. 6.60).
239–243:	Assemble components of element rhs, load, vector.
246–278:	Assemble element stiffness and rhs vector into the global stiffness and global rhs, load, vector.
249–266:	If iter = 1, carry out the global stiffness matrix assembly.
268–278:	Assemble the global rhs, load, vector.

Function**chem_stiff_v2.m**

This function forms the global stiffness and global rhs, load, vector for solution of Cahn–Hilliard Equation for the modified Newton–Raphson solution algorithm. It is optimized for Matlab/Octave.

The function makes calls to the following functions:

- **sfr.m**
- **free_energ_fem_v2.m**

Variable and array list:

npoin:	Total number of nodes in the solution.
nelem:	Total number of elements in the solution.
nnode:	Number of nodes per element.
nstre:	Number of stress components.
ndime:	Number of Cartesian coordinate dimension.
ndofn:	Number DOF per node.
ngaus:	The order of numerical integration.
ntype:	Solution type (ntype = 1, for plane-stress and ntype = 2 for plane-strain).
mobil:	Mobility coefficient.
grcoef:	Gradient energy coefficient.
dtime:	Time increment used in time integration.
istep:	Time increment step number.
iter:	Newton–Raphson iteration number.
con(ntotv):	Nodal values, which includes c and μ values, ntotv = npoin \times ndofn.
con_old(ntotv):	Nodal values from previous time step.
posgp (ngaus):	Position of sampling points.
weigp(ngaus):	Weighting factors at the sampling points.
gforce(ntotv):	Global rhs, load, vector.
lnods(nelem, nnode):	Element nodal connectivity list.
coord(npoin, ndime):	Cartesian coordinates of nodes.
gstif(ntotv,ntotv):	Global stiffness matrix.

dvolum(nelem, mgaus):	Element area/volume.
dgdx(nelem,mgaus, ndime,nnode):	Element, Cartesian derivatives of shape functions.

Listing:

```

1  function [gstif,gforce]=chem_
2    stiff_v2(npoin,nelem,nnode,
3    nstre, ...
4    ndime,ndofn,ngaus,ntype, ...
5    lnods,coord,mobil,grcoef, ...
6    con,con_old,dtime,posgp, ...
7    weigp,dgdx,dvolum,istep, ...
8    iter,gstif)
9
10 %=====
11 %   global and local variables:   %
12 %=====
13
14 ntotv=npoin*ndofn;
15 nevab=nnode*ndofn;
16 %--
17 ngaus2=ngaus;
18 if(nnode == 3)
19 ngaus2=1;
20 end
21
22 %=====
23 %   initialize elements stiffness   %
24 %           & rhs:                 %
25 %=====
26
27 if(iter == 1)
28
29 %-- stiffness matrices:
30 estif1 = zeros(nelem,nnode,nevab);
31 estif2 = zeros(nelem,nnode,nevab);
32 estif = zeros(nelem,nevab,nevab);
33
34 kcc = zeros(nelem,nnode,nnode);
35 kcm = zeros(nelem,nnode,nnode);
36 kmm = zeros(nelem,nnode,nnode);
37 kmc = zeros(nelem,nnode,nnode);
38 kmc1 = zeros(nelem,nnode,nnode);
39
40 end %iter
41

```

```

42 %--- rhs
43 eload1 = zeros(nelem,nnode);
44 eload2 = zeros(nelem,nnode);
45 eload = zeros(nelem,nevab);
46 gforce= zeros(ntotv,1);
47
48 %---composition:
49 cv = zeros(nelem,nnode);
50 cm = zeros(nelem,nnode);
51 cv_old = zeros(nelem,nnode);
52
53 %=====
54 %      elemental values      &
55 %=====
56
57 for inode =1:nnode
58 lnode = lnods( :,inode);
59 cv( :,inode) = con(lnode);
60 cm( :,inode) = con(npoin+lnode);
61 cv_old( :,inode) = con_old(lnode);
62 end
63
64 %=====
65 %      integrate element stiffness    %
66 %      & rhs:                      %
67 %=====
68
69 kgasp=0;
70 for igaus=1:ngaus
71 exisp=posgp(igaus);
72 for jgaus=1:ngaus2
73 etasp =posgp(jgaus);
74 if(nnode ==3)
75 etasp=posgp(ngaus+igaus);
76 end
77
78 kgasp=kgasp+1;
79 [shape,deriv]=sfr2(exisp,etasp,
nnode);
80
81 %== values at the gauss points:
82
83 cvgp = zeros(nelem,1);
84 cmgp = zeros(nelem,1);
85 cv_ogp = zeros(nelem,1);
86
87 for inode=1:nnode
88
89 cvgp = cvgp + cv( :,inode)*shape
(inode);
90
91 cmgp = cmgp + cm( :,inode)*shape
(inode);
92
93 end
94
95 %== chemical potential:
96
97 [dfdc,df2dc]=free_energ_fem_v2
(nelem,cvgp);
98
99 %
100
101 if(iter == 1)
102
103 for inode = 1:nnode
104 for jnode = 1:nnode
105
106 %--- kcc matrix:
107
108 kcc( :,inode,jnode)=kcc( :,inode,
jnode) + ...
109 shape(inode)*shape(jnode) .
*dvolum( :,kgasp);
110
111 %--- kcm matrix:
112
113 kcm( :,inode,jnode) =kcm( :,,
inode,jnode) + dtim*mobil* ...
114 (dgdx( :,kgasp,1,inode).*dgdx( :,
kgasp,1,jnode)+...
115 dgdx( :,kgasp,2,inode).*dgdx( :,
kgasp,2,jnode)).*dvolum( :,kgasp);
116
117 %--- kmmmatrix:
118
119 kmm( :,inode,jnode)= kmm( :,inode,
jnode) + ...
120 shape(inode)*shape(jnode) .
*dvolum( :,kgasp);
121
122 %--- kmc matrix:
123
124 kmc( :,inode,jnode)=kmc( :,,
inode,jnode) -grcoef* ...
125 (dgdx( :,kgasp,1,inode) .
*dgdx( :,kgasp,1,jnode) +...
126 dgdx( :,kgasp,2,inode) .
*dgdx( :,kgasp,2,jnode)) .
*dvolum( :,kgasp);

```

```

127
128 kmc( :,inode,jnode)=kmc
129   ( :,inode,jnode) - df2dc( : ) * ...
130 shape(inode)*shape(jnode).
131   *dvolum( :,kgasp);
132 end
133 end%iter
134 %-----
135 % element rhs
136 %-----
137
138 for inode = 1:nnode
139   eload1( :,inode)=eload1( :,inode)
140   - shape(inode)* ...
141     (cvgp( :) - cv_ogp( : )) .* dvolum
142     ( :, kgasp);
143
144 eload2( :, inode) = eload2( :, inode)
145   - shape(inode)* ...
146     (cmgp( :) - dfdc( : )) .* dvolum
147     ( :, kgasp);
148
149 for jnode = 1:nnode
150   eload1( :,inode) = eload1( :,inode)
151   - dtim*mobil*cmgp( : ).* ...
152     (dgdx( :,kgasp,1,inode).*dgdx
153     ( :,kgasp,1,jnode) + ...
154       dgdx( :,kgasp,2,inode).*dgdx
155     ( :,k_gasp,2,jnode))* ...
156       shape(jnode).*dvolum
157     ( :,k_gasp);
158
159 eload2( :,inode) = eload2
160   ( :,inode) + grcoef*cvgp ( : ).* ...
161     (dgdx( :,kgasp,1,inode) .
162      *dgdx( :,k_gasp,1,jnode) + ...
163        dgdx( :,kgasp,2,inode).*dgdx( : ,
164          k_gasp,2,jnode))* ...
165        shape(jnode).*dvolum( :,kgasp);
166
167 end
168 end%igaus
169 end%jgaus
170
171 % assemble element stiffness
172 and rhs:
173 %--
174 if(iter == 1)
175   for inode=1:nnode
176     ievab=nnode+inode;
177   for jnode=1:nnode
178     jevab=nnode+jnode;
179   estif( :,inode,jnode)=kcc( :,inode,
180     jnode);
181   estif( :,inode,jevab)=kcm( :,inode,
182     jnode);
183   estif( :,ievab,jnode)=kmc( :,inode,
184     jnode);
185   estif( :,ievab,jevab)=kmm( :,inode,
186     jnode);
187   end
188   end
189   end%iter
190   %--
191   for inode=1:nnode
192     ievab=nnode+inode;
193   eload( :,inode)=eload1( :,inode);
194   eload( :,ievab)=eload2( :,inode);
195   end
196   %=====
197   % form global stiffness and rhs
198   if( iter == 1)
199     for idofn=1:ndofn
200       for inode=1:nnode
201         ievab=(idofn-1)*nnode+inode;
202       for jnode=1:nnode
203         jevab=(jdofn-1)*nnode+jnode;
204       gstif = gstif + sparse(((idofn-1)*
205         npoin+lnods( :,inode)),...
206           ((jdofn-1)*npoin+lnods
207             ( :,jnode)),...
208             estif( :,ievab,jevab),
209             ntotv,ntotv);

```

```

208 end
209 end
210 end
211 end
212
213 end%iter
214
215 %--- rhs
216
217 for idofn=1:ndofn
218 for inode=1:nnode
219 ievab=(idofn-1)*nnode+inode;
220
221 gforce = gforce +sparse(((idofn-1)
* npoin+lnods( :,inode)),1...
222 ,eload( :,ievab ),ntotv,1 );
223
224 end
225 end
226
227 end%endfunction
228

```

Line numbers:

14:	Total number of variables in the system.
15:	Total number of variables per element.
17–20:	Order of numerical integration.
27–40:	If iter = 1, initialize element stiffness and submatrices (Eq. 6.60).
42–47:	Initialize element rhs, load, vector and the global rhs, load vector.
48–51:	Initialize nodal values of all elements.
57–62:	Get nodal values (c and μ) of every element in the simulation.
65–163:	Numerical integration of every element in the simulation.
71,	Coordinates of the integration points in the local coordinate system.
73, 75:	
79:	Calculate the shape functions and their derivative values for the current integration point for all elements.
81–93:	Transfer the nodal values of elements to the integration points (Eq. 6.58).
97:	Calculate the derivatives of free energy for every element in the simulation.
101–133:	If iter = 1, calculate the submatrices of element stiffness matrix (Eq. 6.60).
108–109:	Form the submatrix, K_{ij}^{cc} (Eq. 6.61).
113–114:	Form the submatrix, $K_{ij}^{c\mu}$ (Eq. 6.62).
118–120:	Form the submatrix, $K_{ij}^{\mu\mu}$ (Eq. 6.64).
124–129:	Form the submatrix, $K_{ij}^{\mu c}$ (Eq. 6.63).

141–142:	First term of first row of element rhs vector (Eq. 6.59).
144–145:	First and second terms of second row element rhs vector (Eq. 6.59).
149–152:	Second term of first row element rhs vector (Eq. 6.59).
154–157:	Third term of second row element rhs vector (Eq. 6.59).
168–183:	If iter = 1, assemble the element stiffness matrix from submatrices (Eq. 6.60).
186–190:	Assemble components of element rhs, load, vector.
193–226:	Assemble element stiffness and rhs vector into the global stiffness and global rhs, load, vector.
196–213:	If iter = 1, carry out the global stiffness matrix assembly.
217–225:	Assemble the global rhs, load, vector.

Function**free_energ_fem_v1.m**

This function calculates the derivatives of free energy, based on the composition value at the integration points. It is in longhand format not optimized for Matlab/Octave.

Variable and array list:

c:	The value of the concentration at the current integration point.
dfdc:	First derivative of free energy.
df2dc:	Second derivative of free energy.

Listing:

```

1 function [dfdc,df2dc] = free_energ_
fem_v1(c)
2
3 format long;
4
5 constA = 1.0;
6
7 dfdc = constA*( 2.0*c - 6.0*c^2 + 4.0*
c^3 );
8
9 df2dc = constA*( 2.0 - 12.0*c +
12.0*c^2 );
10
11 end %endfunction

```

<i>Line numbers:</i>	
5:	Value of constant A in free energy function (Eq. 6.53).
7:	First derivative of free energy with respect to concentration at current integration point.
9:	Second derivative of free energy with respect to concentration at current integration point.

10:	First derivative of free energy for all elements in the simulation.
12:	Second derivative of free energy for all elements in the simulation.

Function

free_energ_fem_v2.m

This function calculates the derivatives of free energy, based on the composition value at the integration points. It is optimized for Matlab/Octave.

Variable and array list:

nelem:	Total number of elements in the solution.
c(nelem):	The value of the concentration at the integration point of all elements.
dfdc (nelem):	First derivative of free energy.
df2dc (nelem):	Second derivative of free energy.

Listing:

```

1  function [dfdc,df2dc] = free_energ_
2    fem_v2(nelem,c)
3    format long;
4
5    constA=1.0;
6
7    dfdc = zeros(nelem,1);
8    df2dc = zeros(nelem,1);
9
10   dfdc = constA*(2.0*c-6.0*c.^2+4.0*
11     c.^3);
12
13   df2dc = constA*(2.0 -12.0*c+12.0*
14     c.^2);
15
16   end %endfunction

```

Line numbers:

5:	Value of A in free energy function (Eq. 6.53).
7–8:	Initialize derivatives for elements in the simulation.

Function

input_fem_pf.m

This function is the shortened version function *input_fem_elast.m* and reads the FEM mesh. Therefore, just its listing is given in here.

Listing

```

1  function [npoin,nelem,ntype,nnode,
2    ndofn,ndime,ngaus, ...
3    nstrel,nmats,nprop,lnods,matno,
4    coord]=input_fem_pf()
5
6  format long;
7
8  global in;
9  global out;
10
11
12  %read the input data:
13
14  npoin=fscanf(in,'%d',1);
15  nelem=fscanf(in,'%d',1);
16  nvfix=fscanf(in,'%d',1);
17  ntype=fscanf(in,'%d',1);
18  nnode=fscanf(in,'%d',1);
19  ndofn=fscanf(in,'%d',1);
20  ndime=fscanf(in,'%d',1);
21  ngaus=fscanf(in,'%d',1);
22  nstrel=fscanf(in,'%d',1);
23  nmats=fscanf(in,'%d',1);
24  nprop=fscanf(in,'%d',1);
25
26
27  %read the element node numbers &
28  %material property number
29
30  for ielem=1:nelem
31    jelem=fscanf(in,'%d',1);
32    dummy=fscanf(in,'%d',[nnode+1, 1]);
33    for inode=1:nnode
34      lnods(jelem,inode)=dummy(inode);
35    end
36  end

```

```

33 matno(jelem)=dummy(nnode+1);
34 end
35
36 %read nodal coordinates:
37
38 for ipoin=1:npoint
39 jpoint=fscanf(in,'%d',1);
40 dummy=fscanf(in,'%lf %lf',[2,1]);
41 for idime=1:ndime
42 coord(ipoin,idime)=dummy(idime);
43 end
44 end
45
46 for ipoin=1:npoint
47 if(coord(ipoin,1) < 0.0)
48 coord(ipoin,1) =0.0;
49 end
50 if(coord(ipoin,2) < 0.0)
51 coord(ipoin,2) = 0.0;
52 end
53 end
54
55 end %endfunction

```

Function

jacob3.m

This function is the optimized version of function *jacob2.m* for Matlab/Octave. It evaluates the Cartesian shape function derivatives, determinant of the Jacobian for the numerical integration of area of elements, and the Cartesian coordinates of the integration points.

Variable and array list:

ielem:	Current element number.
kgasp:	The current integration point number.
ndime:	Number of global coordinate components.
nnode:	Number of nodes per element.
djac:	The determinate of the Jacobian matrix sampled within the element (ielem) at the integration points and used in calculation of area of the elements.
shape(nnode):	Shape function values.
elcod(ndime, nnode):	Global coordinates of current element (ielem) nodes.
deriv(ndime, nnode):	Derivatives of shape functions in local coordinates.

gpcod(ndime, kgasp):	Cartesian coordinates of the integration points within the element (ielem).
-----------------------------	---

Listing:

```

1 function [cartd,djacb,gpcod]
2 =jacob3(ielem,elcod,kgasp,shape,
3 deriv,nnode,ndime)
4
5 format long;
6
7 %gauss point coordinates:
8
9 gpcod(1,kgasp)=cg(1);
10 gpcod(2,kgasp)=cg(2);
11
12 %jacobian
13
14 xjacm=deriv*elcod';
15
16 %Determinate of Jacobian
17
18 djacb=xjacm(1,1)*xjacm(2,2)-xjacm
19 (1,2)*xjacm(2,1);
20
21 if(djacb <= 0.0)
22 fprintf('Element No: %5d\n',ielem);
23 error('Program terminated zero or
negative area');
24 end
25
26 %cartesion derivatives:
27
28 xjaci(1,1)=xjacm(2,2)/djacb;
29 xjaci(2,2)=xjacm(1,1)/djacb;
30 xjaci(1,2)=-xjacm(1,2)/djacb;
31 xjaci(2,1)=-xjacm(2,1)/djacb;
32
33 cartd = xjaci*deriv;
34
35 end %endfunction

```

Line numbers:

7:	Calculate the integration point Cartesian coordinates.
14:	Form the Jacobian matrix (Eq. 6.5).

(continued)

18:	The determinant of the Jacobian matrix.
27–30:	Inverse of the Jacobian matrix.
32:	Cartesian derivatives of the shape functions using chain rule (Eq. 6.3).

Function

periodic_boundary.m

This function finds the nodes that are at the boundaries of the simulation cell. For each pair of edge (left-right and bottom-top) assigns master and slave node designation.

Variable and array list:

npoin:	Total number of nodes in the FEM mesh.
ncountm:	Number of master nodes.
ncounts:	Number of slave nodes.
master(ncountm):	Node numbers of master nodes.
slave(ncounts):	Node numbers of slave nodes.
coord(npoin,ndime):	Cartesian coordinates of nodes.

Listing:

```

1  function [ncountm,ncounts,master,
2   slave] =periodic_boundry(npoin,
3   coord)
4
5   format long;
6
7   ncountm=0;
8
9   %determine xmax,xmin,ymax,ymin
10
11  xmax = max(coord( :,1));
12  xmin = min(coord( :,1));
13
14  ymax = max(coord( :,2));
15  ymin = min(coord( :,2));
16
17  for ipoin=1:npoint
18
19  %--- left master nodes:
20
21  diff =xmin - coord(ipoin,1);
22  if(abs(diff) <= 1.0e-6)
23  ncountm = ncountm+1;
24  master(ncountm)=ipoin;
25
26
27  %--- right slave nodes:
28
29  diff =xmax - coord(ipoin,1);
30  if(abs(diff) <= 1.0e-6)
31  ncounts = ncounts+1;
32  slave(ncounts)=ipoin;
33
34
35  end
36
37  for ipoin=1:npoint
38
39  %--- Bottom master nodes:
40
41  diff =ymin - coord(ipoin,2);
42  if(abs(diff) <= 1.0e-6)
43  ncountm = ncountm+1;
44  master(ncountm)=ipoin;
45
46
47  %--- Top slave nodes:
48
49  diff =ymax - coord(ipoin,2);
50  if(abs(diff) <= 1.0e-6)
51  ncounts = ncounts+1;
52  slave(ncounts)=ipoin;
53
54
55  end
56
57  if(ncountm ~= ncounts)
58
59  error('ncountm should be equal
60  ncounts');
61
62  end %endfunction

```

Line numbers:

5–6:	Initialize counters for master and slave nodes.
11–15:	Determine the maximum and minimum x - and y -coordinates of the FEM mesh.
17–35:	Find the node numbers on the left and right boundaries of the FEM mesh. Assign the nodes residing on the left boundary as master nodes and on the right ones as slave nodes.
37–55:	Find the node numbers on the top and bottom boundaries of the FEM mesh. Assign the nodes residing on the bottom boundary as master nodes and on the top ones as slave nodes.

(continued)

53–61: Check the number of master and slave nodes. For a regular mesh, they should be equal to each other. If not, terminate the solution.

Function

recover_slave_dof.m

This function assigns the solution values to the slave nodes.

Variable and array list:

npoin:	Total number of nodes in the solution.
ndofn:	Number of DOF per node.
ncountm:	Total number of master nodes.
ncounts:	Total number of slave node.
master	Listing of master nodes.
(ncountm):	
slave(ncounts):	Listing of slave nodes.
asdis(ntotv):	Nodal values, ntotv = npoin × ndofn.

Listing:

```

1  function[asdis] =recover_slave_dof
2      (asdis,ncountm,ncounts, ...
3          master,slave,npoin,ndofn)
4
5      format long;
6
7      for ipbc=1:ncountm
8
9          im=master(ipbc);
10         is=slave(ipbc);
11         asdis(is)=asdis(im);
12     end
13 end %endfunction
14

```

Line numbers:

6–11: Loop over the number of master nodes and assign the solution obtained for the master nodes to corresponding slave nodes.

References

1. Zhang L, Tonks MR, Gaston D, Peterson JW, Andrs D, Millett PC, Biner SB (2013) A quantitative comparison between C⁰ and C¹ elements for solving the Cahn–Hilliard equation. J Comput Phys 236:74

2. Elliot CM, French DA, Milner FA (1989) A second order splitting method for the Cahn–Hilliard equation. Numer Math 54:575
3. Kulh E, Schmid D (2007) Computational modeling of mineral unmixing and growth. Comput Mech 39:439
4. Barrett JW, Blowey JF (1999) Finite element approximation for the Cahn–Hilliard equation. J Comput Phys 226:487
5. Larsson F, Rueness K, Saroukhani S, Vafadari R (2011) Computational homogenization based on weak format of micro-periodicity for rve problems. Comput Methods Appl Mech Eng 200:11
6. Tyrus JM, Gosz M, DeSantiago E (2007) A local finite element implementation for imposing periodic boundary conditions on composite micromechanical models. Int J Solid Struct 44:2972
7. Nguyen VD, Bechet E, Geuzaine C, Noels L (2011) Imposing periodic boundary condition on arbitrary meshes by polynomial interpolation. Comput Mat Sci 55:390

6.7 Case Study-XII

Phase-field modeling of grain growth with finite element method

Objectives:

The objective of this case study is to develop a numerical implementation of the finite element method, FEM, for the solution of multicomponent non-conserved Allen–Cahn equations in phase-field modeling.

6.7.1 Background

The background information regarding this case study has already been described in *Case Study-II* in which the multicomponent non-conserved Allen–Cahn equation was solved by utilizing finite difference algorithm with explicit Euler time marching scheme. The same phase-field model was also solved with semi-implicit Fourier spectral method in *Case Study-VII*. For convenience, the phase-field model is briefly summarized in here again.

6.7.2 Phase-Field Model

In the model, each grain is described by one order parameter η_i which takes the value of one for a designated grain and the value of zero in other grains. Recall Eq. 4.31, the evolution of the order parameters described by the non-conserved Allen–Cahn equation in the form of:

$$\frac{\partial \eta_i}{\partial t} = -L_i \frac{\delta F}{\delta \eta_i}, \quad i = 1, 2, \dots, N \quad (6.67)$$

where L_i is the mobility coefficient and F is the free energy functional which is given by Eq. 4.32 as:

$$F = \int_V \left[f(\eta_1, \eta_2, \dots, \eta_N) + \sum_i^N \frac{\kappa_i}{2} |\nabla \eta_i|^2 \right] dv \quad (6.68)$$

in which κ_i are the gradient energy coefficients and f is the local free energy density.

The specific form of local free energy, which is independent of orientation described by Eq. 4.33 as:

$$f(\eta_1, \eta_2, \dots, \eta_N) = \sum_i^N \left(-\frac{A}{2} \eta_i^2 + \frac{B}{4} \eta_i^4 \right) + \sum_i^N \sum_{i \neq j}^N \eta_i^2 \eta_j^2 \quad (6.69)$$

6.7.3 FEM Formulation

By expending Eq. 6.67, it takes the form:

$$\frac{\partial \eta_i}{\partial t} = -L_i \frac{\partial f}{\partial \eta_i} + L_i \kappa_i \nabla \cdot \nabla \eta_i \quad (6.70)$$

By taking the test function ϕ^* for the order parameters, the weak form of Eq. 6.70 is:

$$\begin{aligned} \int_V \left[\phi^* \left(\frac{\partial \eta_i}{\partial t} \right) \right] dV &= -L_i \int_V \left[\phi^* \left(\frac{\partial f}{\partial \eta_i} \right) \right] dV \\ &- L_i \kappa_i \int_V [\nabla \phi^* \cdot \nabla \eta_i] dV \end{aligned} \quad (6.71)$$

The time discretization of Eq. 6.71 with explicit Euler algorithm limits the time stepping size to very small time increments. On the other hand, the fully implicit Euler algorithm leads to computationally costly reevaluation of system matrix and Newton–Raphson iterative procedure at each time steps. Alternatively, the time integration of Eq. 6.71 can be carried out with semi-implicit scheme in which only the linear and the Laplacian parts are sought in the new time step and the local nonlinear part is

taken from the previous time step. Thus, Eq. 6.71 with this semi-implicit scheme can be expressed as:

$$\begin{aligned} \int_V [\phi^* \eta_i^{t+1}] dV + \Delta t L_i \kappa_i \int_V [\nabla \phi^* \cdot \nabla \eta_i^{t+1}] dV \\ = \Delta t \int_V [\phi^* \eta_i^t] dV - \Delta t L_i \int_V \left[\phi^* \frac{\partial f^t}{\partial \eta_i} \right] dV \end{aligned} \quad (6.72)$$

in which $\Delta t = t^{t+1} - t^t$ is the time increment between two time steps. This formulation has considerable computational advantages [1]:

1. The system matrix is formed only once and it is used in all time steps.
2. The nonlinear solution scheme with Newton–Raphson algorithm is avoided completely.

6.7.4 Numerical Implementation

By taking the nodal variables as η_i and utilizing the shape functions of isoparametric elements, the values of non-conserved order parameters at

any Cartesian point within the element can be determined as:

$$\eta_i = \sum_j^n N_j \eta_i^j \quad (6.73)$$

where n is number of nodes of that element and N_j and η_i^j are the nodal values of shape and η_i , respectively. By utilizing Eqs. 6.72 and 6.73, the resulting FEM equations at the element level for semi-implicit time marching scheme can be expressed as:

$$\left[K_{ij}^e \right] \{ \eta_i^{t+1} \} = \{ R_i^e \}^t \quad (6.74)$$

in which K_{ij}^e is the element stiffness matrix, η_i^{t+1} is the nodal unknowns, and R_i^e is the rhs, load, vector. From Eqs. 6.72 and 6.73, the stiffness matrix and rhs, load, vector takes the form:

$$K_{ij}^e = \int_V \left[N_i N_j + \Delta t L K \left(\frac{\partial N_i}{\partial x} \frac{\partial N_j}{\partial y} \right)^T \cdot \left(\frac{\partial N_i}{\partial x} \frac{\partial N_j}{\partial y} \right) \right] dV \quad (6.75)$$

and

$$R_i^e = \int_V \left[N_i \eta_i^t - \Delta t L N_i \frac{\partial f^t}{\partial \eta_i} \right] dV \quad (6.76)$$

where $(\partial N / \partial x, \partial N / \partial y)$ are the derivatives of shape functions in Cartesian coordinates and T stands for the transpose.

There are two possibilities to discretize Eq. 6.74 in the simulation domain. First, to include all the order parameters into the unknown vector, which yields very large system of equations and makes the problem almost numerically un-treatable, even for small number of grains in the solution. The alternative approach is to apply a staggered solution scheme in which each grain is considered one at a time during time increment steps, similar to the solution algorithms used in *Case Studies-II and VII*. Thus, with the chose of second approach, the over all algorithm is:

Step-1

Form the global stiffness matrix from Eq. 6.75.

Step-2

Time integration:

For $i\text{grain} = 1:n\text{grain}$ ($n\text{grain}$ is the number of grains in the simulation)

Form global rhs, load, vector from Eq. 6.76.

Solve resulting system of equations.

Update the nodal unknowns for $i\text{grain}$.

end

Repeat Step-2 till desired total time for evolution is reached.

6.7.5 Results and Discussion

Again, as in two previous *Case Studies-II and VII*, the shrinkage of circular grain in bicrystal form as ideal grain growth and the grain evolution in a polycrystal composed of 25 grains were simulated.

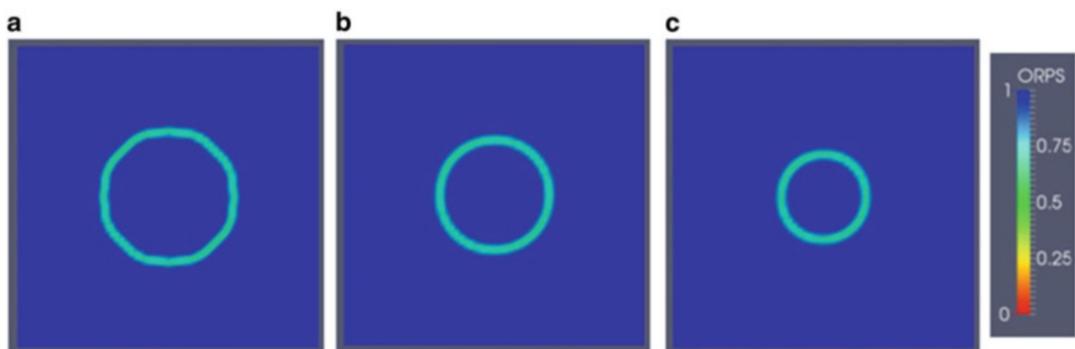


Fig. 6.5 Time evolution of a spherical grain. The nondimensional times are: (a) 0.005, (b) 12.5, and (c) 25.0

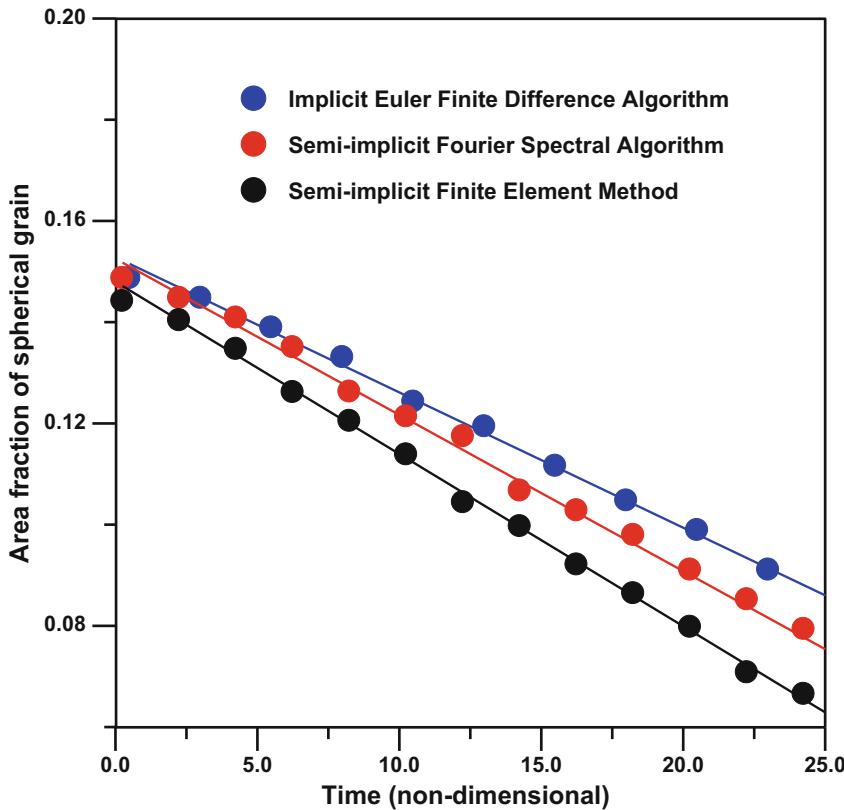


Fig. 6.6 Comparison of the results obtained from explicit Euler finite difference algorithm and semi-implicit Fourier spectral method

The simulations were carried with FEM mesh composed of 4425 nodes and 8192 tri-node isoparametric elements. The mesh input file *mesh_64.inp* and the input_file for polycrystalline simulations *grain_25.inp* are given in subdirectory *case_study_12* of downloadable file, together with the animation files resulting from these simulations.

Figure 6.5 shows the time evolution of the spherical grain during the course of the simulation. With increasing time, the uniform shrinkage of the spherical grain is apparent from the figure. The variation of the area fraction of the spherical grain with time is compared with the results obtained in *Case Study-II* with explicit Euler finite difference algorithm and in *Case Study VII* with semi-implicit Fourier spectral method in Fig. 6.6. Again, we observe a linear shrinkage rate of spherical grain, and the result agrees

with the results seen in two previous solution algorithms reasonable well by considering very coarse spatial discretization was used in these two previous simulations.

In the next set of simulation, the grain growth is considered, again by using the same input file *grain_25.inp*. The time evolution of the polycrystalline microstructure composed of 25 grains is shown in Fig. 6.7. Similar to that observed in previous two case studies the evolution proceeds as the growth of the larger grains and the disappearance of the smaller ones.

After identical time steps, the final grain microstructures seen in the three different simulation algorithms are summarized in Fig. 6.8. As can be seen, the simulations yield almost the same final grain morphology, even though there were large differences in their spatial and temporal discretization.

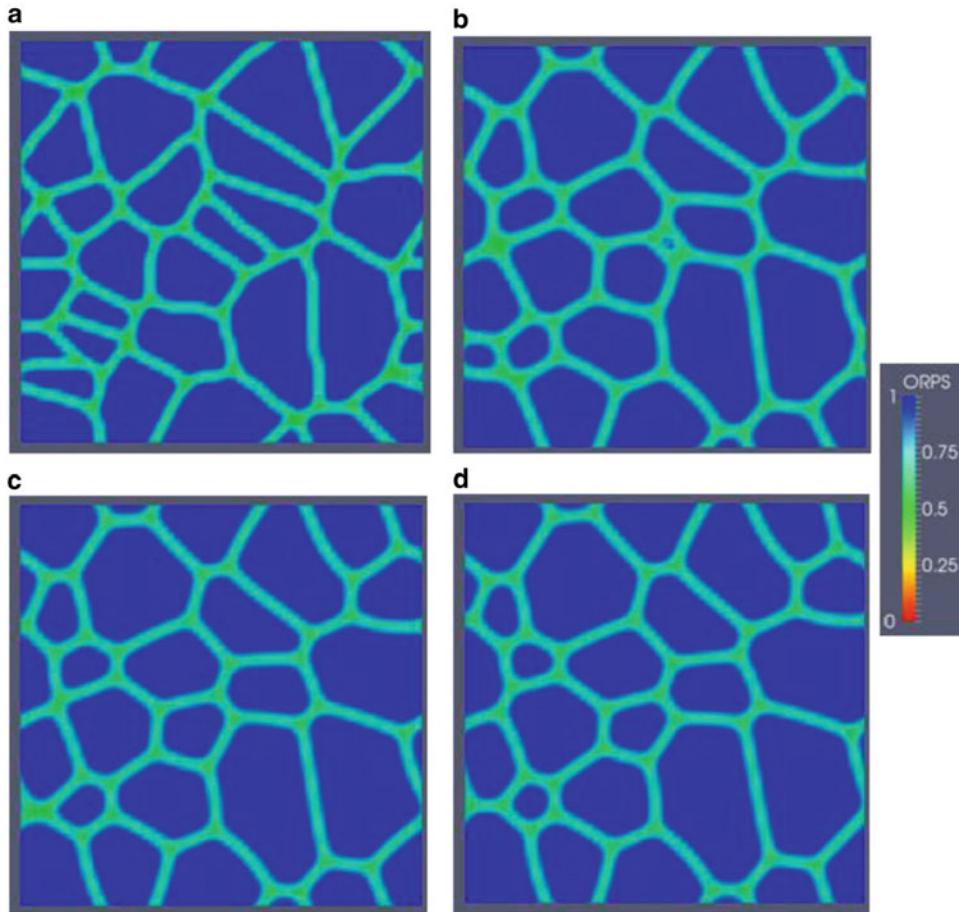


Fig. 6.7 Time evolution of polycrystalline microstructure due to grain growth. The nondimensional times are: (a) 0.005, (b) 12.5, (c) 18.75, and (d) 25.0

6.7.6 Source Codes

Program

fem_ca_v1_2.m

This is the main program to solve multicomponent Allen–Cahn phase-field equation with FEM algorithm. Depending on the selection of the isolve parameter in the code: For isolve = 1, the code executes in un-optimized in longhand format mode. For isolve = 2, execution is for Matlab/Octave optimized mode. Therefore, for the desired execution mode, this parameter in line 21 should be modified in the program.

The program makes calls to the following functions:

- `input_fem_pf.m`
- `periodic_boundary.m`
- `gauss.m`
- `cart_deriv.m`
- `init_micro_grain_fem.m`
- `gg_stiff_1.m`
- `gg_stiff_2.m`
- `apply_periodic_bc2a.m`
- `gg_rhs_1.m`
- `gg_rhs_2.m`
- `apply_periodic_bc2b.m`
- `recover_slave_dof.m`
- `write_vtk_fem.m`

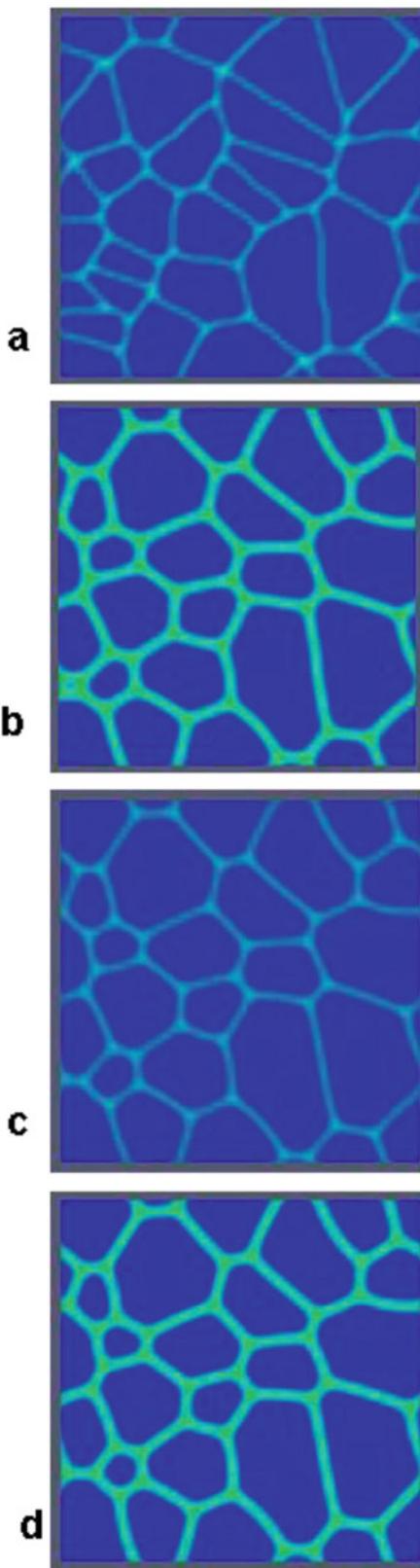


Fig. 6.8 Comparison of the final grain morphologies obtained after identical time steps from the three different simulation algorithms. **(a)** the initial grain microstructure used in all three simulations, **(b)** is for explicit Euler finite

difference algorithm, **(c)** is for semi-implicit Fourier spectral algorithm, and **(d)** is for semi-implicit finite element method

Listing:

```

1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %
3  % FEM PHASE-FIELD CODE FOR SOLVING %
4  %          ALLEN-CAHN EQUATION      %
5  %          (GRAIN GROWTH)        %
6  %
7  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
8
9  % get wall time:
10
11 time0=clock();
12 format long;
13 %-
14 global in; in=fopen
15 ('mesh_64.inp','r');
16 global out; out=fopen
17 ('result_1.out','w');
18 global out2; out2=fopen
19 ('area_fract.out','w');
20 %-
21 isolve=2;
22 %-- Time integration parameters:
23 nstep = 5000;
24 nprnt = 50;
25 dttime = 0.005;
26 ttime = 0.0;
27 %-- Material specific parameters:
28 mobil= 5.0;
29 grcof= 0.1;
30 %-----
31 %input data:
32 %-----
33 [npoin,nelem,nctype,nnode,ndofn,
34 ndime,ngaus, ...
35 nstre,nmats,nprop,lnods,matno,
36 coord]=input_fem_pf();
37 [ncountm,ncounts,master,slave]
38 =periodic_boundary(npoin,coord);
39 %-----
40 % Evolve %
41 %-----%
42 %-----%
43
44 [posgp,weigp]=gauss(ngaus,nnode);
45 if(isolve == 2)
46 %
47 [dgdx,dvolum] = cart_deriv(npoin,
48 nelem,nnode,nstre,ndime,ndofn, ...
49 ngaus,nctype,lnods,coord,
50 posgp,weigp);
51 end
52 %
53 %-- initialize microstructure:
54 %
55 iflag = 1;
56 [etas,ngrain] = init_micro_
57 grain_fem(npoin,coord,iflag);
58 ntotv=npoin*ngrain;
59 %-----
60 %-- form stiffness matrix:
61 %-----
62 %
63 if(isolve == 1)
64 %
65 [gstif] = gg_stiff_1(ngrain,npoin,
66 nelem,nnode,nstre,ndime,ndofn, ...
67 ngaus,nctype,lnods,coord,mobil,
68 grcof, ...
69 dttime,posgp,weigp);
70 %
71 if(isolve == 2)
72 %
73 [gstif] = gg_stiff_2(ngrain,npoin,
74 nelem,nnode,nstre,ndime,ndofn, ...
75 ngaus,nctype,lnods,coord,mobil,
76 grcof, ... dttime,posgp,weigp,
77 dgdx,dvolum);
78 %---Rearrange gstif for pbc
79 %
80 [gstif] = apply_periodic_bc2a
81 (ncountm,ncounts,master,slave, ...
82 ndofn,npoin,gstif);
83 %-----%
84 %-----%

```

```

86   for istep=1:nstep
87
88   ttime = ttime + dtme;
89
90   for igrain=1:ngrain
91
92   if(isolve == 1)
93
94   [gforce]=gg_rhs_1(ngrain,npoin,
95   nelem,nnode,nstre,ndime,ndofn, ...
96   ngaus,ntype,lnods,coord,mobil,
97   grcof, ...
98   etas,dtime,posgp,weigp,
99   igrain);
100
101  end
102
103  %--
104  if(isolve == 2)
105
106  [gforce]=gg_rhs_2(ngrain,npoin,
107  nelem,nnode,nstre,ndime,ndofn, ...
108  ngaus,ntype,lnods,coord,
109  mobil,grcof, ...
110  etas,dtime,posgp,weigp,
111  igrain,dgdx,dvolum);
112
113  end
114
115  %
116  %-- Rearrange gforce for pbc:
117  %
118  [gforce] = apply_periodic_bc2b
119  (ncountm,ncounts,master,slave, ...
120  ndofn,npoin,gforce);
121
122  %-----
123  % solve equations:
124  %
125  asdis = gstif\gforce;
126
127  %-- Recover slave node values:
128
129  for ipoin=1:npoin
130  itotv=(igrain-1)*npoin+ipoin;
131  etas(itotv) = asdis(ipoin);
132
133  end%igrain
134
135  %--- print out:
136
137  if(mod(istep,nprnt) == 0)
138
139  fprintf('done step: %d\n',istep);
140
141  dummy=zeros(npoin,1);
142
143  for igrain=1:ngrain
144  for ipoin=1:npoin
145  itotv=(igrain-1)*npoin+ipoin;
146  dummy(ipoin)=dummy(ipoin)+etas
147  (itotv)^2;
148
149  %fname1=sprintf('time_%d.out',
150  istep);
151
152  %out1=fopen(fname1,'w');
153
154  %for ipoin=1:npoin
155
156  %fclose(out1);
157
158  %-----
159  %-- calculate area fraction of
160  % grains:
161
162  fprintf(out2,'%14.6e',ttime);
163
164  for igrain=1:ngrain
165  ncount=0;
166
167  for ipoin=1:npoin
168  itotv=(igrain-1)*npoin+ipoin;
169  if(etas(itotv) >= 0.5)
170  ncount=ncount+1;
171
172  end
173
174  ncount = ncount/npoin;

```

```

171 fprintf(out2,'%14.6e',ncount);
172 end
173 fprintf(out2,'\n');
174
175 end %if
176
177 end %istep
178
179 compute_time = etime(clock()),
time0)
180 fprintf(out,'Compute_time:
%10d\n',compute_time);

```

Line numbers:

9:	Get wall clock time beginning of the execution.
15–17:	Unit names of input and output files.
21:	Control parameter for solution mode. isolve = 1, the code executes for un-optimized in longhand format. For isolve = 2, execution is for Matlab/Octave optimized mode.
23–29:	Time integration parameters.
25:	Number of time integration steps.
26:	Print frequency, to output results to file.
27:	Time increment of numerical integration.
28:	Total time.
30–34:	Material-specific parameters.
32:	Mobility coefficient.
33:	Gradient energy coefficient.
36–50:	Input data
39–40:	Read FEM mesh and control parameters.
42:	Define master and slave nodes in the FEM mesh.
44:	Parameters for numerical integration of elements.
48–49:	If solution is in optimized mode (isolve = 2), precalculate the Cartesian derivatives of shape functions of all elements in the solution.
53–56:	Initialize grain microstructure.
55:	Iflag = 1 for bicrystal simulation, iflag = 2 for polycrystal simulation.
58:	Total number of variables in the solution.
61–76:	Depending on the solution mode, form global stiffness matrix.
66–68:	Form global stiffness matrix, for un-optimized solution mode (isolve = 1).
73–75:	Form global stiffness matrix, for optimized solution mode (isolve = 2).
78–81:	Rearrange the global stiffness matrix for periodic boundary conditions.
83–177:	Evolve the microstructure.
88:	Update the total time.
90–133:	Loop over number of grains in the solution.

92–97:	Form global rhs, load, vector for un-optimized solution mode (isolve = 1).
101–106:	Form global rhs, load, vector for optimized solution mode (isolve = 2).
116–120:	Solve resulting FEM equations.
122–125:	Recover the nodal values of slave nodes.
126–131:	After solution, update global nodal variable vector, etas(ntotv), ntotv = npoin × ngrain.
137–175:	If print frequency is reached, output the results to file and calculate the area fraction of grains.
142–147:	Loop over grains and collect the nodal values of each grain into dummy(npoin) array in squared form from global nodal variable array etas(ntotv).
149–154:	Open an output file and print the coordinates for nodes and the nodal values to file. These lines are commented out, but they can be changed, including the output format.
157:	Write the results in vtk file format for contour plots to be viewed by using Paraview.
159–174:	Calculate the area fractions of each grain and output the results to <i>area_fraction.out</i> file.
179–180:	Calculate the total execution time and print it.

Function**apply_periodic_bc2a.m**

This function modifies the global stiffness matrix for the periodic boundary conditions.

Variable and array list:

ncountm:	Number of master nodes.
ncounts:	Number of slave nodes.
ndofn:	Number of DOF per node.
npoin:	Total number of nodes in the solution.
master (ncountm):	Node list of master nodes.
slave (ncounts):	Node list of slave nodes.
gstif(ntotv, ntotv):	Global stiffness matrix, ntotv = npoin × ndofn.

Listing:

```

1 function [gstif] = apply_
periodic_bc2a(ncountm,ncounts,
master,slave, ...
2 ndofn,npoin,gstif)
3
4 format long;
5
```

```

6   for ipbc=1:nountm
7
8   im=master(ipbc);
9   is=slave(ipbc);
10
11 %%-- Add rows:
12
13 gstif(im,1:npoint)=gstif
14   (im,1:npoint) + gstif(is,1:npoint);
15 %%-- Add columns:
16
17 gstif(1:npoint,im)=gstif(1:npoint,
18   im) + gstif(1:npoint,is);
19 %%-- zero slave dofs
20
21 gstif(is,1:npoint) = 0.0;
22 gstif(1:npoint,is) = 0.0;
23
24 gstif(is,is) =1.0;
25
26 end
27
28 end %endfunction
29

```

Line numbers:

6–26:	Loop over master nodes.
8:	Node number of master node.
9:	Node number of slave node.
11–18:	Add global stiffness matrix rows and columns of slave nodes to corresponding rows and columns of master nodes.
19–23:	Zero the rows and columns of slave nodes.
24:	Insert value of one to the diagonals of slave nodes in global stiffness matrix.

slave (ncounts):	Node list of slave nodes.
gforce(ntotv, ntotv):	Global rhs, load, vector, ntotv = npoin × ndofn.

Listing:

```

1 function [gforce] = apply_
2   periodic_bc2b(nountm,ncounts,
3     master,slave,...)
4   ndofn,npoin,gforce)
5
6   format long;
7
8   for ipbc=1:nountm
9
10  im=master(ipbc);
11  is=slave(ipbc);
12
13  %%-- add rhs
14
15  gforce(im) = gforce(im)+gforce(is);
16  gforce(is) = 0.0;
17
18 end %endfunction
19

```

Line numbers:

6–16:	Loop over master nodes.
8:	Node number of master node.
9:	Node number of slave node.
13:	Add rhs, load, vector values of slave nodes to the corresponding rhs, load, vector values of master nodes.
14:	Zero the rhs, load, vector values of slave nodes.

Function**apply_periodic_bc2b.m**

This function modifies the global rhs, load, vector for the periodic boundary conditions.

Variable and array list:

nountm:	Number of master nodes.
ncounts:	Number of slave nodes.
ndofn:	Number of DOF per node.
npoin:	Total number of nodes in the solution.
master (nountm):	Node list of master nodes.

Function**gg_free_energ_v1.m**

This function calculates the derivative of free energy function for current grain under consideration in the main program, at the current integration point.

Variable and array list:

igrain:	Current grain number under consideration in the main program.
ngrain:	Total number of grains.

(continued)

dfdeta:	Derivative of free energy with respect to current grain.
eta_gp (ngrain):	Order parameter values of grains at current integration point.

Listing:

```

1 function [dfdeta] =gg_free_energ_v1
2   (igrain,ngrain,eta_gp)
3   format long;
4
5   A=1.0;
6   B=1.0;
7
8   sum = 0.0;
9
10  for jgrain =1:ngrain
11    if(igrain ~= jgrain);
12    sum =sum + eta_gp(jgrain)^2;
13  end
14 end
15
16 dfdeta =A*(2.0*B*eta_gp(igrain)
17 *sum+eta_gp(igrain)^3-eta_gp
18 (igrain));
19
20 %endfunction

```

Line numbers:

5–6:	Coefficients in free energy function, Eq. 6.69.
10–14:	Calculate the value of sum in Eq. 6.69.
16:	Calculate the derivative of free energy function with respect to current grain under consideration in the main program for the current element integration points.

Function

gg_free_energ_v2.m

This function calculates the derivative of free energy function for current grain under consideration in the main program, at the current integration point of all elements in the simulation.

Variable and array list:

nelem:	Total number of elements in the simulation.
igrain:	Current grain number under consideration in the main program.

ngrain:	Total number of grains.
dfdeta(nelem):	Derivative of free energy with respect to current grain.
eta_gp(nelem, ngrain):	Order parameter values of grains at current integration point.

Listing:

```

1 function [dfdeta] =gg_free_energ_v2
2   (nelem,igrain,ngrain,eta_gp)
3   format long;
4
5   A=1.0;
6   B=1.0;
7
8   dfdeta =zeros(nelem,1);
9
10  sum = zeros(nelem,1);
11
12  for jgrain = 1:ngrain
13    if(igrain ~= jgrain)
14    sum =sum + eta_gp( :,jgrain).^2;
15  end
16 end
17
18 dfdeta =A*(2.0*B*eta_gp( :,igrain).
19 *sum +eta_gp( :,igrain).^3-eta_gp
20 ( :,igrain));

```

Line numbers:

5–6:	Coefficients of free energy, Eq. 6.69.
8:	Initialize the array containing the derivatives values for all elements in the solution.
10–16:	Calculate the value of sum in Eq. 6.69 for all elements in the simulation.
18:	Calculate the derivative of free energy function with respect to current grain under consideration in the main program, for all elements in the solution.

Function

gg_rhs_1.m

This function calculates the global rhs, load, vector. It is in longhand format and is not optimized.

The function makes calls to the following functions:

- **sfr2.m**
- **jacob2.m**

Variable and array list:

ngrain:	Number of grains in the simulation.
npoin:	Total number of nodes.
nelem:	Total number of elements.
nnode:	Total number of nodes per element.
nstre:	Number of stress components.
ndime:	Number of Cartesian coordinate dimension.
ndofn:	Number of DOF per node.
ngaus:	Number of integration order.
ntype:	Solution type, ntype = 1 for plane-stress, ntype = 2 for plane-strain.
mobil:	Mobility coefficient.
grcoef:	Gradient energy coefficient.
dtime:	Time step for numerical integration.
igrain:	Current grain number under consideration in the main program.
etas(ntotv):	Array for nodal values of order parameters, ntotv = npoin × ngrain.
posgp(ngaus):	Position of integration points.
weigp(ngaus):	Weights at the integration points.
gforce(npoin):	Global rhs, load, vector.
coord(npoin, ndime):	Cartesian coordinates of the nodes.
lnods(nelem, nnode):	Element node connectivity list.

Listing:

```

1  function[gforce]=gg_rhs_1(ngrain,
2    npoin,nelem,nnode,nstre,ndime,
3    ndofn, ...
4    ngaus,ntype,lnods,coord,mobil,
5    grcoef, ...
6    etas,dtime,posgp,weigp,igrain)
7  format long;
8  %%-- global and local variables:
9  ngaus2=ngaus;
10 if(nnode == 3)
11 ngaus2=1;
12 end
13 ntotv=npoin;
14 nevab=nnode;
15 gforce =sparse(ntotv,1);
16
17 %
18 for ielem=1:nelem
19 %--initialize
20
21 eload(ievab)=0.0;
22 end
23
24 %== elemental nodal values:
25
26
27
28 for igr=1:ngrain
29 for inode=1:nnode
30 lnode = (igr-1)*npoin+lnods(ielem,
31 inode);
32 eta(igr,inode) =etas(lnode);
33 end
34 end
35
36 %--- coords of the element nodes
37
38 for inode=1:nnode
39 lnode=lnods(ielem,inode);
40 for idime=1:ndime
41 elcod(idime,inode)=coord(lnode,
42 idime);
43 end
44 end
45
46 %=====
47 %      integrate element load      %
48 %=====
49
50 kgasp=0;
51 for igaus=1:ngaus
52 exisp=posgp(igaus);
53 for jgaus=1:ngaus2
54 etasp =posgp(jgaus);
55 if(nnode ==3)
56 etasp=posgp(ngaus+igaus);
57 end
58
59 kgasp =kgasp + 1;
60
61 [shape,deriv]=sfr2(exisp,etasp,
62 nnode);
63 [cartd,djacb,gpcod]=jacob2(ielem,
64 elcod,kgasp,shape, ...
65 deriv,nnode,ndime);
```

```

65
66 dvolu=djacb*weigp(igaus)*weigp
(jgaus);
67
68 if(nnode == 3)
69 dvolu = djacb* weigp(igaus);
70 end
71
72 %==== values at gauss points:
73
74 for igr=1:ngrain
75 eta_gp(igr) = 0.0;
76 end
77
78 for igr=1:ngrain
79 for inode =1:nnode
80
81 eta_gp(igr) =eta_gp(igr)+ eta
(igr,inode)*shape(inode);
82
83 end
84 end
85
86 %==== free energy derivatives:
87
88 [dfdeta] =gg_free_energ_v1
(igrain,ngrain,eta_gp);
89
90
91 %==== form element RHS
92
93 for inode =1:nnode
94
95 eload(inode)=eload(inode)+shape
(inode)*eta_gp(igrain).*dvolu;
96
97 eload(inode)=eload(inode) - shape
(inode)*dtim*mobil*
dfdeta*dvolu;
98
99 end
100
101 end %jgaus
102 end %igaus
103
104 %==== GLOBAL RHS:
105
106 for inode=1:nnode
107
108 gforce = gforce +sparse(lnods
(ielem,inode),1,eload(inode),
ntotv,1);
109
110 end
111
112 end %ielem
113
114 end %endfunction

```

Line numbers:

8–11:	Order of numerical integration.
13:	Total number of variables.
14:	Total number of variables per element.
16:	Initialize global rhs, load, vector.
19–112:	Loop over elements.
23–25:	Initialize element rhs, load, vector.
30–35:	Get elemental nodal values.
39–44:	Get Cartesian coordinates of the element nodes.
50–102:	Numerical integration of element.
52,	Coordinates of integration points in local coordinate system, depending on the element type.
54, 56:	
59:	Increment integration point counter.
61:	Calculate the shape functions and their derivative values.
63–64:	Calculate the Cartesian derivatives of shape functions.
66–70:	Calculate the element area/volume.
72–85:	Calculate the values of order parameter values of the grains at the current integration point. Eq. 6.73.
88:	Calculate the derivative of free energy for the current grain under consideration in the main program for current integration point.
91–99:	Form element rhs, load, vector.
95:	Calculate the first term of Eq. 6.76.
97:	Calculate the second term of Eq. 6.76.
104–110:	Assemble element rhs, load, vector into global rhs, load vector.

**Function
gg_rhs_2.m**

This function calculates the global rhs, load, vector. The code is optimized for Matlab/Octave.

The function makes calls to the following functions:

- **sfr2.m**

Variable and array list:	
ngrain:	Number of grains in the simulation.
npoin:	Total number of nodes.
nelem:	Total number of elements.
nnode:	Total number of nodes per element.
nstre:	Number of stress components.
ndime:	Number of Cartesian coordinate dimension.
ndofn:	Number of DOF per node.
ngaus:	Number of integration order.
ntype:	Solution type, ntype = 1 for plane-stress, ntype = 2 for plane-strain.
mobil:	Mobility coefficient.
grcoef:	Gradient energy coefficient.
dtime:	Time step for numerical integration.
igrain:	Current grain number under consideration in the main program.
etas(ntotv):	Array for nodal values of order parameters, ntotv = npoin \times ngrain.
posgp(ngaus):	Position of integration points.
weigp(ngaus):	Weights at the integration points.
gforce(npoin):	Global rhs, load, vector.
coord(npoin, ndime):	Cartesian coordinates of the nodes.
Inods(nelem, nnode):	Element node connectivity list.
dvolum(nelem, mgaus, ndime, nnode):	Elemental area/volume values at the integration points.
dgdx(nelem, mgaus, ndime, nnode):	Cartesian derivatives of shape functions for all elements in the solution.

Listing:

```

1 function[gforce]=gg_rhs_2(ngrain,    46 kgasp =kgasp + 1;
2 npoin,nelem,nnode,nstre,ndime,      47
3 ndofn,...                           48 [shape,deriv]=sfr2(exisp,etasp,
4    ngaus,ntype,lnods,coord,         nnode);
5     mobil,grcoef,...               49
6     etas,dtime,posgp,weigp,        50 %==== values at gauss points:
7     igrain,dgdx,dvolum)           51
8 format long;                      52 eta_gp=zeros(nelem,ngrain);
9                                     53
10 %%-- global and local variables: 54 for igr=1:ngrain
11                                     55 for inode=1:nnode
12 ngaus2=ngaus;                     56

```

```

57 eta_gp( :,igr) =eta_gp( :,igr) +
58 eta( :,igr,inode)*shape(inode);
59 end
60 end
61
62 %== free energy derivatives:
63
64 [dfdeta] =gg_free_energ_v2(nelem,
65 igrain,ngrain,eta_gp);
66
67 %== form element RHS
68
69 for inode =1:nnode
70 ievab= inode;
71
72 eload( :,inode)=eload( :,inode)
73 +shape(inode)*eta_gp( :,igrain) *
74 *dvolum(:,kgasp);
75
76 eload(:,inode)=eload(:,inode)
77 - shape(inode)*dtime*mobil
78 *dfdeta.*dvolum(:,kgasp);
79
80 end
81 end %jgaus
82 end %igaus
83
84 %== GLOBAL RHS:
85
86 for inode=1:nnode
87
88 gforce = gforce +sparse(lnods
89 ( :,inode),1,eload( :,inode),
90 ntovt,1);
91
92 end %endfunction

```

Line numbers:

8–11:	Order of numerical integration.
13:	Total number of variables.
14:	Total number of variables per element.
16:	Initialize global rhs, load, vector.
20:	Initialize temporary array eta for all elements in the solution.

22:	Initialize the rhs, load, vector of all elements.
26–31:	Calculate the nodal values of all elements.
34–79:	Numerical integration of elements.
39,	Coordinates of the integration points in the local coordinate system.
41, 43:	
46:	Increment the integration point counter.
48:	Calculate the shape functions and their derivative values.
50–60:	Calculate the order parameter values at the current integration point for all elements, Eq. 6.73.
62–65:	Derivative of free energy for the current grain under consideration in the main program for all elements in the solution.
67–77:	Form the rhs, load, vector all elements.
72:	Calculate the first term of Eq. 6.76.
74:	Calculate the second term of Eq. 6.76.
83–88:	Assemble the rhs, load, vector all elements into the global rhs, load vector.

Function**gg_stiff_1.m**

This function calculates the global stiffness matrix. It is in longhand format and is not optimized.

The function makes calls to the following functions:

- **sfr2.m**
- **jacob2.m**

Variable and array list:

ngrain:	Number of grains in the simulation.
npoind:	Total number of nodes.
nelem:	Total number of elements.
nnode:	Total number of nodes per element.
nstre:	Number of stress components.
ndime:	Number of Cartesian coordinate dimension.
ndofn:	Number of DOF per node.
ngaus:	Number of integration order.
ntype:	Solution type, ntype = 1 for plane-stress, ntype = 2 for plane-strain.
mobil:	Mobility coefficient.
grcoef:	Gradient energy coefficient.
dtime:	Time step for numerical integration.
etas(ntovt):	Array for nodal values of order parameters, ntovt = npoin × ngrain.
posgp(nngaus):	Position of integration points.

(continued)

weigp(ngaus):	Weights at the integration points.
coord(npoin, ndime):	Cartesian coordinates of the nodes.
lnods(nelem, nnode):	Element node connectivity list.
gstif(npoin, npoin):	Global stiffness matrix.

Listing:

```

1   function[gstif] = gg_stiff_1
2     (ngrain,npoin,nelem,nnode,nstre,
3      ndime,ndofn, ...
4      ngaus,nctype,lnods,coord,mobil,
5      grcoef, ...
6      dtim, posgp,weigp)
7      format long;
8
9      %%-- global and local variables:
10
11      ngaus2=ngaus;
12      if(nnode == 3)
13          ngaus2=1;
14      end
15
16      %%--
17
18      for ielem=1:nelem
19
20      %%-- initialize
21
22      for ievab=1:nevab
23          for jevab=1:nevab
24              estif(ievab,jevab)=0.0;
25          end
26      end
27
28
29      %%-- coords of the element nodes
30
31      for inode=1:nnode
32          lnode=lnods(ielem,inode);
33          for idime=1:ndime
34              elcod(idime,inode)=coord(lnode,
35                  idime);
36          end
37
38      %=====
39      % integrate elements stiffness %
40      %=====
41
42      kgasp=0;
43      for igaus=1:ngaus
44          exisp=posgp(igaus);
45          for jgaus=1:ngaus2
46              etasp =posgp(jgaus);
47              if(nnod ==3)
48                  etasp=posgp(ngaus+igaus);
49              end
50
51              kgasp =kgasp + 1;
52
53              [shape,deriv]=sfr2(exisp,etasp,
54                  nnod);
55
56              [cartd,djacb,gpcod]=jacob2(ielem,
57                  elcod,kgasp,shape, ...
58                  deriv,nnod,ndime);
59
60              dvol=djacb*weigp(igaus)*weigp
61                  (jgaus);
62
63
64      %== form element stiffness:
65
66      for inode=1:nnod
67          ievab=inode;
68          for jnode=1:nnod
69              jevab=jnode;
70
71              estif(ievab,jevab) = estif(ievab,
72                  jevab)+shape(inode)*shape(jnode)
73                  *dvol;
74
75              estif(ievab,jevab) = estif(ievab,
76                  jevab) + dtim*mobil*grcoef* ...
77                  (cartd(1,inode)*cartd(1,jnode)
78                  + cartd(2,inode)*cartd(2,jnode))
79                  *dvol;
80
81          end
82      end
83
84      end %jgaus

```

```

80 end %igaus
81
82 %=====
83 % form global stiffness
84 %=====
85
86 for inode=1:nnode
87 itotv=lnods( :,inode );
88 for jnode=1:nnode
89 jtovt=lnods(:,jnode);
90 gstif=gstif +sparse(lnods(ielem,
91 inode),lnods(ielem,jnode),estif
92 (inode,jnode),ntotv,ntotv);
93 end
94 end
95 end %ielem
96
97
98 end %endfunction

```

Line numbers:

8–11:	Order of numerical integration, depending on the element type.
13:	Total Number of variables.
14:	Total number of variables per element.
15:	Initialize global stiffness matrix.
19–95:	Loop over elements.
23–27:	Initialize element stiffness matrix.
31–36:	Cartesian coordinates of element nodes.
39–80:	Numerical integration of element.
44,	Position of integration points in local coordinate system.
46, 48:	
51:	Increment integration point counter.
53:	Calculate the shape functions and their derivative values.
55–56:	Cartesian derivatives of shape functions.
58–62:	Calculate element area/volume.
64–77:	Form element stiffness matrix.
71:	Calculate the first term in Eq. 6.75.
73–74:	Calculate the second term in Eq. 6.75.
83–93:	Assemble element stiffness matrix into the global stiffness matrix.

Function**gg_stiff_2.m**

This function calculates the global stiffness matrix. It is optimized for Matlab/Octave.

The function makes calls to the following functions:

- **sfr2.m**

Variable and array list:

ngrain:	Number of grains in the simulation.
npoin:	Total number of nodes.
nelem:	Total number of elements.
nnode:	Total number of nodes per element.
nstre:	Number of stress components.
ndime:	Number of Cartesian coordinate dimension.
ndofn:	Number of DOF per node.
ngaus:	Number of integration order.
ntype:	Solution type, ntype = 1 for plane-stress, ntype = 2 for plane-strain.
mobil:	Mobility coefficient.
grcoef:	Gradient energy coefficient.
dtime:	Time step for numerical integration.
etas(ntotv):	Array for nodal values of order parameters, ntotv = npoin × ngrain.
posgp(ngaus):	Position of integration points.
weigp(ngaus):	Weights at the integration points.
coord(npoin, ndime):	Cartesian coordinates of the nodes.
lnods(nelem, nnode):	Element node connectivity list.
gstif(npoin, npoin):	Global stiffness matrix.
dvolum(nelem, mgaus):	Elemental area/volume values at the integration points.
dgdx(nelem, mgaus,ndime, nnode):	Cartesian derivative of shape functions for all elements in the solution

Listing:

```

1 function[gstif] = gg_stiff_2
2   (ngrain,npoin,nelem,nnode,nstre,
3   ndime,ndofn, ...
4   ngaus,ntype,lnods,coord,
5   mobil,grcoef, ...
6   dtime,posgp,weigp,dgdx,
7   dvolum)
8   format long;
9
10  ngaus2=ngaus;
11  if (nnode == 3)
12    ngaus2=1;
13  end

```

```

11  %--- initialize global & elements
12  stiffness:
13  ntotv=npoin;
14  nevab=nnode;
15
16  gstif =sparse(ntotv,ntotv);
17
18  estif=zeros(nelem,nevab,nevab);
19
20
21 %=====
22 % integrate elements stiffness %
23 %=====
24
25  kgasp=0;
26  for igaus=1:ngaus
27    exisp=posgp(igaus);
28    for jgaus=1:ngaus2
29      etasp =posgp(jgaus);
30      if(nnode ==3)
31        etasp=posgp(ngaus+igaus);
32    end
33
34  kgasp =kgasp + 1;
35
36  [shape,deriv]=sfr2(exisp,etasp,
nnode);
37
38 %== form element stiffness:
39
40  for inode=1:nnode
41    ievab=inode;
42    for jnode=1:nnode
43      jevab=jnode;
44
45      estif( :,ievab,jevab) = estif( :,
ievab,jevab)+shape(inode)*shape
(jnode).*dvolum( :,kgasp);
46
47      estif( :,ievab,jevab) = estif( :,
ievab,jevab) + dtime*mobil*
grcoef* ...
48      (dgdx ( :,kgasp,1,inode).*dgdx ( :,
kgasp,1,jnode) + ...
49      dgdx ( :,kgasp,2,inode).*dgdx ( :,
kgasp,2,jnode)).*dvolum(:,kgasp);
50
51  end
52  end
53
54  end %jgaus
55  end %igaus
56
57 %=====
58 %   global stiffness
59 %=====
60
61  for inode=1:nnode
62    itotv=lnods( :,inode);
63    for jnode=1:nnode
64      jtotv=lnods( :,jnode);
65      gstif =gstif +sparse(lnods( :,
inode),lnods( :,jnode),estif
(:,inode,jnode),ntotv, ntotv);
66
67  end
68  end
69
70
71
72  end %endfunction

```

Line numbers:

6–9:	Order of numerical integration.
13:	Total number of variables.
14:	Total number of variables per element.
16:	Initialize global stiffness matrix.
18:	Initialize element stiffness matrices of all elements in the solution.
22–55:	Numerical integration of elements.
27,	Position of integration points in the local coordinate system.
29, 31:	Increment integration point counter.
34:	Calculate the shape functions and their derivative values.
36:	Form element stiffness matrices of all elements in the solution.
38–52:	Calculate the first term in Eq. 6.75.
45:	Calculate the second term in Eq. 6.75.
47–49:	Assemble element stiffness matrices of all elements into the global stiffness matrix.
58–68:	

Function

init_micro_grain_fem.m

This function is almost identical to the function *init_grain_micro.m* and generates the order parameters for the nodes in the FEM mesh. The iflag sets the initial grain microstructure either to a circular bicrystal or to a polycrystalline

microstructure consisting of 25 grains. Therefore, it is just listed and not annotated again in here.

Variable and array list:

iflag:	iflag = 1, for bicrystal simulation, iflag = 2 for polycrystal simulation.
npoin:	Total number of nodes in the solution.
ngrain:	Number of grains in the simulations
etas(ntotv):	Array containing the order parameters of grains, ntotv = npoin × ngrain.
coord(npoin, ndime):	Nodal coordinates.

Listing:

```

1   function [etas,ngrain] = init_
2       micro_grain_fem(npoin,coord,
3           iflag)
4
5   % iflag=1 two grain
6   % iflag=2 polycrystal
7
8   if(iflag == 2)
9       in=fopen('grain_25.inp','r');
10
11 end
12
13 %-----
14 % generate two grains
15 %-----
16
17 if(iflag == 1)
18
19 ngrain =2;
20
21 ntotv=ngrain*npoin;
22 etas=zeros(ntotv,1);
23
24 %center of mesh & size of second grain
25 x0=16.0;
26 y0=16.0;
27 radius= 7.0;
28
29 for ipoin=1:npoin
30     etas(ipoin)=1.0;
31     rad1=sqrt((x0-coord(ipoin,1))^2+
32     (y0-coord(ipoin,2))^2);
33     if(rad1 <= radius)
34         etas(ipoin)=0.0;
35     etas(npoin+ipoin)=1.0;
36     end
37 end
38
39 %-----
40 %-----%
41 %      generate polycrystal
42 %      microstructure%
43 if(iflag == 2)
44
45 %-----
46 % read data generate from voroni_1.m
47 %-----
48
49 twopi=8.0*atan(1.0);
50 epsilon=1.0e-4;
51 ndime=2;
52
53 nvpoint = fscanf(in,'%d',1);
54 nvnode = fscanf(in,'%d',1);
55 nvelem = fscanf(in,'%d',1);
56 ngrain = fscanf(in,'%d',1);
57
58 for ipoin=1:nvpoint
59     jpoint=fscanf(in,'%d',1);
60     dummy=fscanf(in,'%lf %lf',[2,1]);
61     for idime=1:ndime
62         vcord(jpoint,idime)=dummy(idime);
63     end
64 end
65
66 for ielem=1:nvelem
67     jelem=fscanf(in,'%d',1);
68     dummy=fscanf(in,'%d',[nvnode
69     +1,1]);
70     for inode=1:nvnode+1
71         vlnods(ielem,inode)=dummy(inode);
72     end
73
74 %-----
75 for ielem=1:nvelem
76
77     jnode=0;
78     for inode=1:nvnode
79         knode=vlnods(ielem,inode);
80         if(knode ~= 0)

```

```

81 jnode = jnode +1;
82 end
83 end
84 nnode2(ielem)=jnode;
85 end
86
87 %-----
88 % mark order parameters
89 %-----
90
91 for igrain=1:ngrain
92 for ipoin=1:npoint
93 itotv=(igrain-1)*npoint+ipoin;
94 etas(itotv)=0.0;
95 end
96 end
97
98 for ipoin=1:npoint
99
100 for ielem=1:nvelem
101
102 igrain=vlnods(ielem,nvnode+1);
103
104 itotv=(igrain-1)*npoint+ipoin;
105
106 theta=0.0;
107
108 mnode=nnode2(ielem);
109
110 for inode=1:mnode
111
112 knode=vlnods(ielem,inode);
113
114 xv1=vcord(knode,1);
115 yv1=vcord(knode,2);
116
117 jnode=vlnods(ielem,inode+1);
118 if(inode==mnode)
119 jnode=vlnods(ielem,1);
120 end
121
122 xv2=vcord(jnode,1);
123 yv2=vcord(jnode,2);
124
125 p1x=xv1-coord(ipoin,1);
126 p1y=yv1-coord(ipoin,2);
127
128 p2x=xv2-coord(ipoin,1);
129 p2y=yv2-coord(ipoin,2);
130
131 x1=sqrt(p1x*p1x+p1y*p1y);
132 x2=sqrt(p2x*p2x+p2y*p2y);
133
134 if(x1*x2 <= epsilon)
135 theta=twopi;
136 else
137 tx1=((p1x*p2x+p1y*p2y)/(x1*x2));
138 end
139
140 if(abs(tx1)>=1.0)
141 tx1=0.9999999999;
142 end
143
144 theta=theta+acos(tx1);
145
146 end %inode
147
148 if(abs(theta-twopi)<=epsilon)
149 etas(itotv)=1.0;
150 end
151
152 end %ielem
153
154 end %ipoin
155
156 %-----
157
158 end %iflag
159
160
161 end %endfunction

```

Reference

- Glodkov S, Svendsen B (2010) Towards coupled finite element modeling of grain growth with plasticity. In: Proceeding of 3rd International Conference on Nonlinear Dynamics, Ukraine, Sep 2010, p 478

6.8 Case Study-XIII

Phase-field modeling of instabilities in layered thin films

Objectives:

The objective of this case study is to demonstrate a numerical implementation of the FEM

for the solution of conserved Cahn–Hilliard equation combined with the elastic inhomogeneities in phase-field modeling. In this case, the resulting system of equations are solved in decoupled mode. The approach serves as a template for formulism of other phase-field models in which this separation of field variables is possible.

6.8.1 Background

The epitaxially grown films on substrates of different crystals have attracted considerable interest due to their important applications in the semiconductor and electro-optical device industry. Owing to the crystal lattice mismatch between the film and substrate, the generated internal stresses in the system inevitably lead to the defect formation. Two main defects in heteroepitaxial films are: First is the surface roughening and morphological instabilities which makes it very difficult to grow planar films that are highly desired in many electronic and optical applications. This stress-relief mechanism was investigated theoretically in [1–3]. The second is the generation of misfit dislocations which usually leads to poor device performance. Since the energy barrier for their formation is usually small, the misfit dislocations are almost always generated as another stress-relieving mechanism [4–6].

There are several simulation studies on the stress-induced morphological instabilities in thin films. Some of these studies can be found in references, at the atomistic level [7, 8], at mesoscale with phase-field method [9–13] and at continuum level as sharp interface models [14, 15].

6.8.2 Phase-Field Model

In the presence of elastic inhomogeneities, the total free energy is assumed to be additive and composed of chemical and elastic energies as studied in *Case Study-IX*. Recall Eq. 5.29, in

this case the standard evolution equation for Cahn–Hilliard equation is expressed as:

$$\frac{\partial c}{\partial t} = \nabla \cdot M \nabla \left(\frac{\delta F^{CH}}{\delta c} + \frac{\delta W^{EL}}{\delta c} \right) \quad (6.77)$$

F^{CH} is expressed, as in previous cases, as

$$F^{CH} = \int_V \left[\frac{\delta f(c)}{\delta c} + \frac{1}{2} \kappa |\nabla c|^2 \right] dv \quad (6.78)$$

again, a simple double-well potential for $f(c)$ is assumed.

$$f(c) = Ac^2(1 - c)^2 \quad (6.79)$$

The elastic energy W is defined as:

$$W(\varepsilon, c) = \frac{1}{2} [\varepsilon - \varepsilon^0(c)] : C(c) [\varepsilon - \varepsilon^0(c)] \quad (6.80)$$

in which the elasticity matrix $C(c)$ is assumed to be linearly varying between the phases.

$$C(c) = C_M + c(C_P - C_M) \quad (6.81)$$

where indices M and P represent the elasticity matrices of matrix and the evolving phase, respectively.

The linearized strains, ε , is related to the displacement field u by

$$\varepsilon(u) = \frac{1}{2} (\nabla u + (\nabla u)^T) \quad (6.82)$$

and the transformation-induced eigenstrains, $\varepsilon^0(c)$, is defined as:

$$\varepsilon^0(c) = c \varepsilon^0 \delta_{ij} \quad (6.83)$$

in which δ_{ij} is the Kronecker delta function, and ε^0 is the magnitude of eigenstrains resulting from the intrinsic lattice mismatch between the phases.

6.8.3 FEM Formulation

By rearranging Eq. 6.77, it reads

$$\begin{aligned} \frac{\partial c}{\partial t} - \nabla \cdot M \left(\nabla \left(\frac{\partial f}{\partial c} + \frac{\partial W}{\partial c} - \kappa \nabla^2 c \right) \right) &= 0 \text{ in } V \\ M \left(\nabla \left(\frac{\partial f}{\partial c} + \frac{\partial W}{\partial c} - \kappa \nabla^2 c \right) \right) &= 0 \text{ on } \partial V \\ M \kappa \nabla c \cdot n &= 0 \text{ on } \partial V \end{aligned} \quad (6.84)$$

where V is the volume, ∂V is the boundary, and n is the outward normal vector to the boundary. The formulism, in the absence of elastic strain energy, to solve Eq. 6.84 by introducing an auxiliary variable with standard isoparametric elements is detailed in *Case Study-XI*. Following the same steps, for implicit-Euler time integration the weak form Eq. 6.84 now reads as:

$$\begin{aligned} \int_V \frac{c^{n+1} - c^n}{dt} \eta dV + \int_V M \nabla \frac{\partial W}{\partial c} \cdot \nabla \eta dV + \int_V M \nabla \mu^{n+1} \cdot \nabla \eta dV = 0 \\ \int_V \mu^{n+1} \zeta dV - \int_V \frac{\partial f^{n+1}}{\partial c} \zeta dV - \int_V \kappa \nabla c^{n+1} \cdot \nabla \zeta dV = 0 \end{aligned} \quad (6.85)$$

where dt is the time increment, $dt = t^{n+1} - t^n$, μ is the auxiliary variable, η and ζ are the test functions.

$$R_i^e = \int_V \left[(c^{n+1} - c^n) N_i + dt M \frac{\partial W}{\partial c} \left(\frac{\partial N_i}{\partial x} \frac{\partial N_i}{\partial y} \right)^T \cdot \left(\frac{\partial N_j}{\partial x} \frac{\partial N_j}{\partial y} \right) N_j + dt M \mu^{n+1} \left(\frac{\partial N_i}{\partial x} \frac{\partial N_i}{\partial y} \right)^T \cdot \left(\frac{\partial N_j}{\partial x} \frac{\partial N_j}{\partial y} \right) N_j \right. \\ \left. \mu^{n+1} N_i - \frac{\partial f^{n+1}}{\partial c} N_i - \kappa c^{n+1} \left(\frac{\partial N_i}{\partial x} \frac{\partial N_i}{\partial y} \right)^T \cdot \left(\frac{\partial N_j}{\partial x} \frac{\partial N_j}{\partial y} \right) N_j \right] dV \quad (6.87)$$

in which $(\partial N / \partial x, \partial N / \partial y)$ are the derivatives of shape function in Cartesian coordinates and T stands for the transpose.

Corresponding element stiffness to Eq. 6.87 is:

$$K_{ij}^e = \frac{\partial R_i^e}{\partial c \text{ or } \partial \mu} = \begin{bmatrix} K_{ij}^{cc} & K_{ij}^{c\mu} \\ K_{ij}^{\mu c} & K_{ij}^{\mu\mu} \end{bmatrix} \quad (6.88)$$

and its components are calculated as:

6.8.4 Numerical Implementation

Because of dependence of $C(c)$ and $\varepsilon^0(c)$ to phase-field variable, c , in Eqs. 6.80 through 6.84, they are coupled set of time-dependent nonlinear equations. In the following implementation in here, however, the evolution of the stress-strain fields is decoupled from the Cahn–Hilliard equation and they are solved separately. Besides the computational efficiency, the physical justification for this decoupling is that the stress–strain fields evolve at a considerable much faster rate than the evolution of the phases by diffusional processes.

Again, by taking the nodal variables as c and μ and utilizing the shape functions of isoparametric elements, the values of c and μ can be determined at any Cartesian point within the element as:

$$c = \sum_i^n N_i c_i \quad \text{and} \quad \mu = \sum_i^n N_i \mu_i \quad (6.86)$$

where n is number of nodes of that element and N_i , c_i , and μ_i are the values of shape functions and the nodal values of c and μ , respectively. Then, the nodal residual vector at the element takes the form:

$$K_{ij}^{cc} = \int_V \left[N_i^T \cdot N_j + dt M \left(\frac{\partial^2 W}{\partial c^2} \right) \left(\frac{\partial N_i}{\partial x} \frac{\partial N_i}{\partial y} \right)^T \cdot \left(\frac{\partial N_j}{\partial x} \frac{\partial N_j}{\partial y} \right) \right] dV \quad (6.89)$$

$$K_{ij}^{c\mu} = \int_V \left[dt M \left(\frac{\partial N_i}{\partial x} \frac{\partial N_i}{\partial y} \right)^T \cdot \left(\frac{\partial N_j}{\partial x} \frac{\partial N_j}{\partial y} \right) \right] dV \quad (6.90)$$

$$K_{ij}^{\mu c} = \int_V \left[-\frac{\partial^2 f}{\partial c^2} N_i^T N_j - \kappa \left(\frac{\partial N_i}{\partial x} \frac{\partial N_i}{\partial y} \right) \cdot \left(\frac{\partial N_j}{\partial x} \frac{\partial N_j}{\partial y} \right) \right] dV \quad (6.91)$$

$$K_{ij}^{\mu\mu} = \int_V [N_i^T N_j] dV \quad (6.92)$$

After forming element stiffness and residuals, they are assembled into the global stiffness matrix and right-hand side vectors, and the resulting set of nonlinear equation is

$$[K^G]\{d\delta\} = \{R^G\} \quad (6.93)$$

Then, the algorithm to solve the evolution of concentration field in decoupled form follows the following steps:

Step-1

For each time increment:

Solve the stress-strain fields by utilizing concentration field from the previous time step as

internal stress problem in linear elasticity, and calculate the resulting derivatives of elastic strain energy in Eqs. 6.87 and 6.89.

Form the global stiffness matrix, $[K^G]$, in Eq. 6.93 only once for this time step.

Step-2

Newton-Raphson iterations:

Form the right-hand side residual vector, $\{R^G\}$, in Eq. 6.87

Solve Eq. 6.93

Update $\delta^{n+1} = \delta^n + d\delta$

Check convergence, if it satisfied go to step-1, else repeat step-2.

6.8.5 Results and Discussion

A single layer and a double layer film configurations, as shown in Fig. 6.9, were considered in the simulations. In both cases, the thickness of the films were about the same. The width-to-length aspect ratio of the simulation cell was 1:4. The simulation cell discretized with three-node isoparametric elements. There were 2145 nodes and 4096 elements in the FEM mesh. Only for two

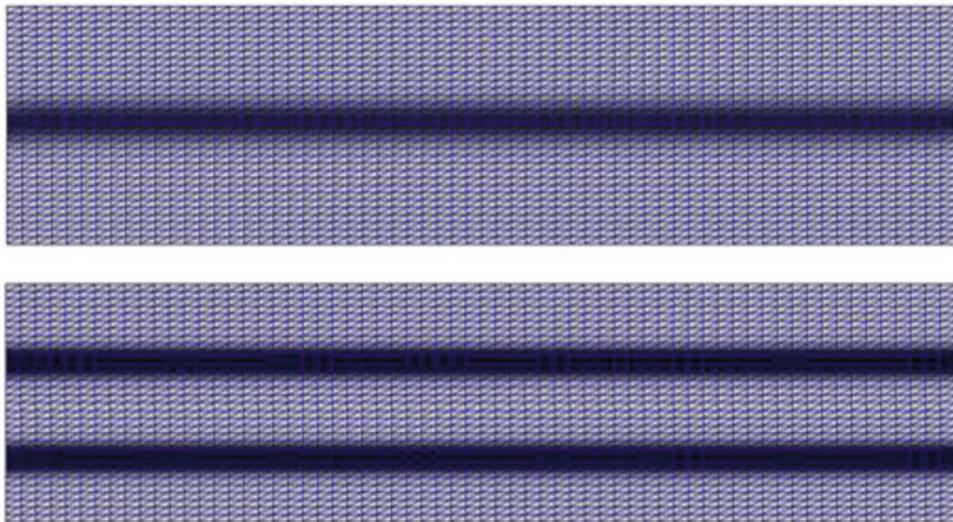
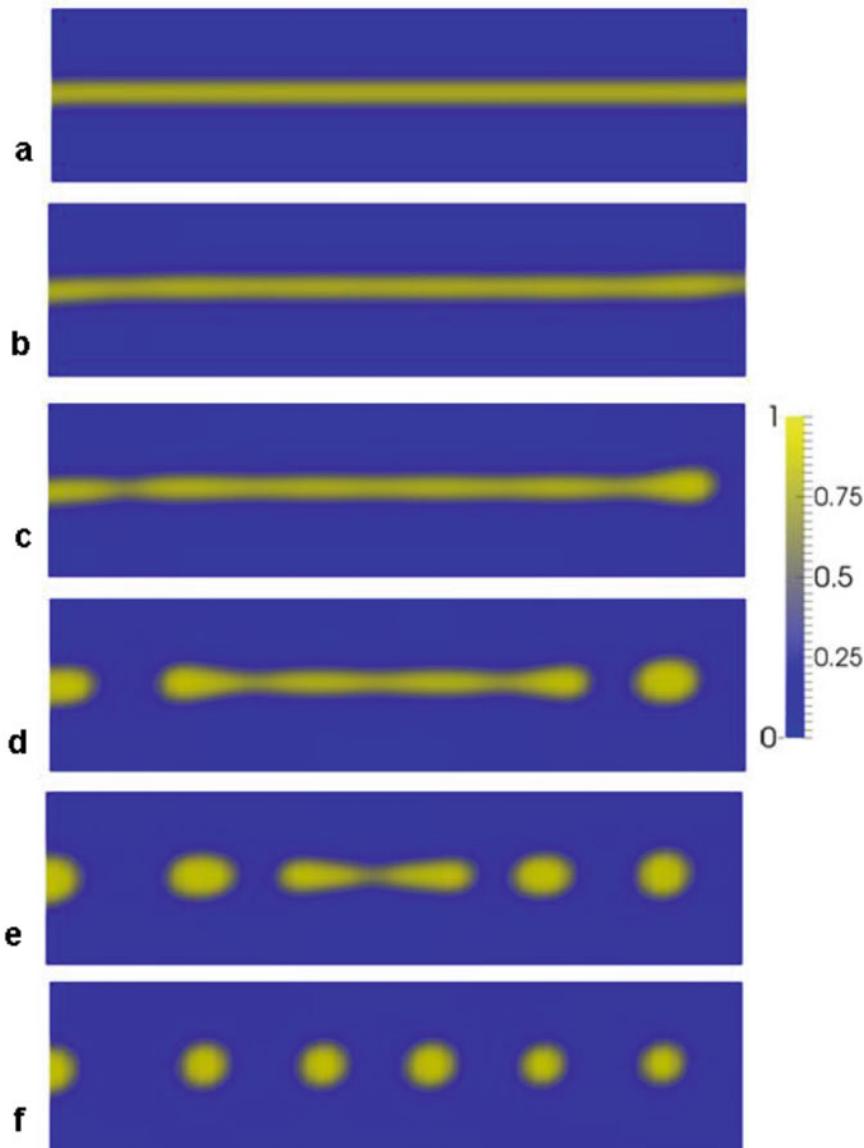


Fig. 6.9 FEM mesh and morphology of thin films in the simulations

Table 6.1 Parameters used in the simulations

M	κ	E_M	E_F	$\nu_M=\nu_F$	ε_{ij}	Dt
1.0	3.0×10^{-3}	50.0	100.0	0.3	5.0×10^{-3}	2.5×10^{-4}

**Fig. 6.10** Evolution of instability for single thin film layer in soft matrix. Nondimensional times in the figure are: (a) 0.025, (b) 1.5, (c) 1.75, (d) 2.0, (e) 2.25, and (f) 2.5

nodes that were residing at the bottom edge corners, their displacements in the both directions were constrained to be zero. There was no periodicity and no applied loads were present. The parameters used in the simulations

are summarized in Table 6.1. As can be seen from the table, the elastic constants of the film were twice of that of matrix and it has only dilatational misfit strains.

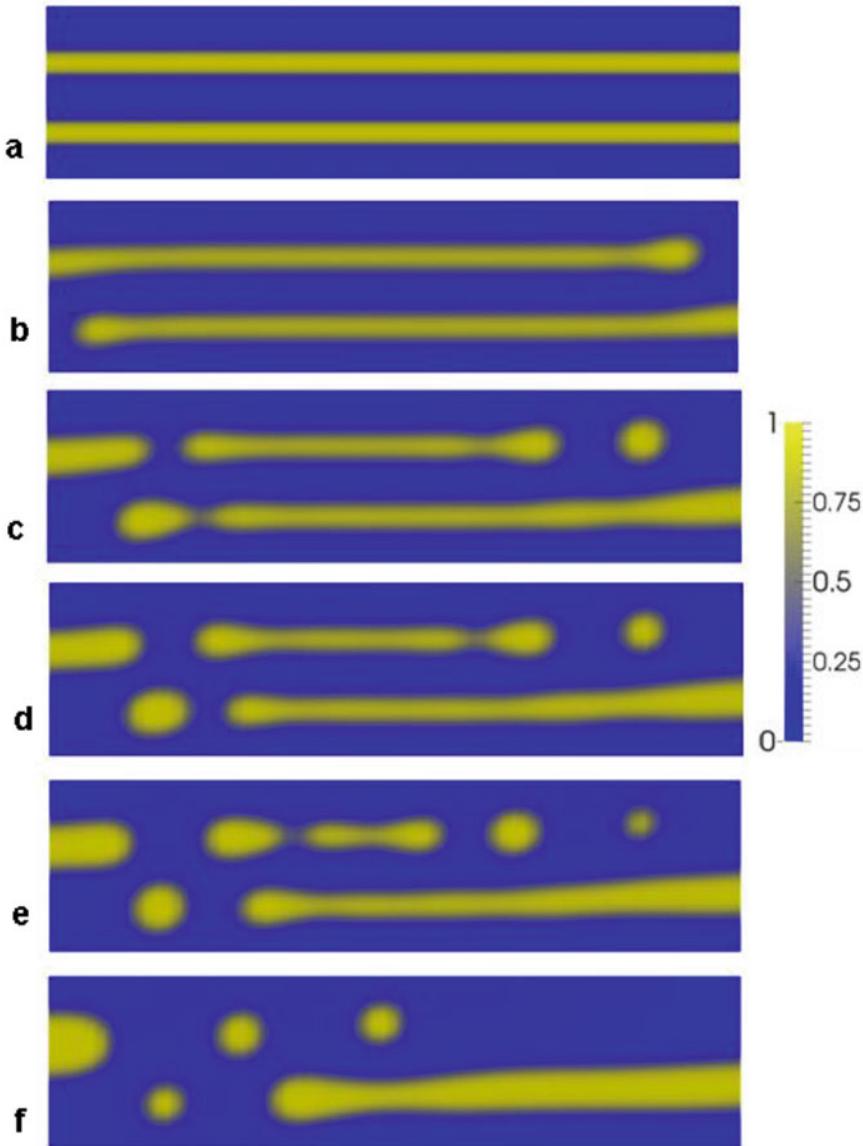


Fig. 6.11 Evolution of instability for double layer thin film in soft matrix. Nondimensional times in the figure are: (a) 0.025, (b) 1.125, (c) 1.375, (d) 1.625, (e) 1.875, and (f) 2.5

The matrix concentration was modulated to 0.001 and the concentration of the films was set to be 0.999 and they also modulated with same amount as the matrix in function *init_micro_thin_film.m*.

The time evolution of the single film in a soft matrix is summarized in Fig. 6.10. As can be seen, shortly after start of the simulation, the

instability starts near the edges driven by the stress state. Then, at multiple locations, the thinning, breakup, and spheroidization events follow. Eventually, the initially perfectly smooth layered composite form turns into particulate composite form.

The simulation results for double layer thin film are summarized in Fig. 6.11. Although, the

events taken place are the same to that seen in the case of mono layer film, however, final morphology is different resulting from the evolution of different stress state in this case. Even though very coarse FEM mesh is used in these simulations, results are very closely agreeing with the previous simulations and experiments listed in the references.

The input FEM mesh file and movie files resulting from the simulations are given in subdirectory *case_study_13* in downloadable file.

6.8.6 Source Codes

Program

fem_thin_film.m

This is the main program to solve Cahn–Hilliard equation with elastic inhomogeneities with FEM algorithm. Depending on the selection of the isolve parameter in the code: For isolve = 1, the code executes in longhand format and un-optimized mode. For isolve = 2, execution is for Matlab/Octave optimized mode. Therefore, for the desired execution mode, this parameter in line 21 should be modified in the program.

The program makes calls to the following functions:

- **input_fem_elast.m**
- **gauss.m**
- **cart_deriv.m**
- **loads.m**
- **init_micro_thin_film.m**
- **elastic_energ_v1.m**
- **elastic_energ_v2.m**
- **chem_stiff_v3.m**
- **chem_stiff_v4.m**
- **write_vtk_fem.m**

Listing:

```

1    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2    %                                %
3    % FEM PHASE-FIELD CODE FOR SOLVING %
4    % CAHN-HILLIARD EQUATION WITH      %
5    %                                %
6    % ELASTIC INHOMOGENEITY          %
7    %                                %
8    % get wall time:                %
9    %                                %
10   time0=clock();                 %
11   format long;                  %
12   %--                            %
13   global in;                   %
14   in=fopen('mesh_64_4.inp','r'); %
15   %--                            %
16   global out;                  %
17   out=fopen('result_1.out','w'); %
18   %--                            %
19   %--                            %
20   isolve=2;                     %
21   %--                            %
22   %--- Time integration parameters: %
23   %                                %
24   nstep= 10000;                 %
25   nprnt= 100;                   %
26   dtimer= 2.5e-4;               %
27   toler= 1.0e-4;               %
28   miter= 10;                    %
29   %--- Material specific parameters: %
30   %                                %
31   mobil= 1.0;                  %
32   grcoef= 3.0e-3;              %
33   %                                %
34   eigen(1) = 0.005;             %
35   eigen(2) = 0.005;             %
36   eigen(3) = 0.000;             %
37   %----- %
38   %input data:                   %
39   %----- %
40   [npoin,nelem,nvfix,ntype,           %
41   nnode,ndofn,ndime,ngaus, ...       %
42   nstre,nmats,nprop,lnods,           %
43   matno,coord,props,nofix, ...       %
44   iffix,fixed]=input_fem_elast();    %
45   %----- %
46   ntotv=npoin*ndofn;                %
47   nevab=nnode*ndofn;                %
48   %----- %
49   [posgp, weigp]=gauss(ngaus,nnode); %
50   %----- %
51   %----- %
52

```

```

53 if(isolve == 2)
54
55 [dgdx,dvolum] = cart_deriv(npoin,
      nelem,nnode,nstre,ndime,ndofn, ...
56     ngaus,ntype,lnods,coord,
      posgp,weigp);
57 end
58
59 %-- applied load vector:
60
61 [gapld] = loads(npoin,nelem,
      ndofn,nnode,ngaus,ndime,posgp, ...
62     weigp,lnods,coord);
63 %-----
64 %-- Prepare microstructure:
65 %-----
66 %
67 icase = 1;
68
69 [con] = init_micro_thin_film
      (npoin,coord,icase);
70
71 %-----&
72 %          EVOLVE          &
73 %-----&
74 %-----&
75 for istep=1:nstep
76
77 con_old=con;
78
79 %---- calculate elastic energy
80 % derivatives:
81
82 if(isolve == 1)
83
84 [dsendc,dsen2dc] = elastic_energ_
      v1(npoin,nelem,nnode,nstre,
      ndime,ndofn, ...
85     ngaus,ntype,lnods,matno,
      coord,props, ...
86     nvfix,nofix,iffix,fixed,
      posgp,weigp, ...
87     con,eigen,gapld);
88 end
89
90 if(isolve == 2)
91
92 [dsendc,dsen2dc] = elastic_energ_
      v2(npoin,nelem,nnode,nstre,
      ndime,ndofn, ...
93     ngaus,ntype,lnods,matno,
      coord,props, ...
94     nvfix,nofix,iffix,fixed,
      posgp,weigp, ...
95     dgdx,dvolum,con,eigen,
      gapld);
96
97 end
98
99 %-----
100 %--Newton iteration:
101 %-----
102 for iter = 1: miter
103
104 if(iter == 1)
105 gstif=sparse(ntotv,ntotv);
106 end
107
108 if(isolve == 1)
109
110 [gstif,gforce]=chem_stiff_v3
      (npoin,nelem,nnode,nstre,
      ndime, ...
111     ndofn,ngaus,ntype,lnods,
      coord, ...
112     mobil,grcoef,con,con_old,
      dtime, ...
113     posgp,weigp,istep,iter, ...
114     dsendc,dsen2dc,gstif);
115 end
116 %
117
118 if(isolve == 2)
119
120 [gstif,gforce]=chem_stiff_v4
      (npoin,nelem,nnode,nstre,
      ndime, ...
121     ndofn,ngaus,ntype,lnods,
      coord, ...
122     mobil,grcoef,con,con_old,
      dtime, ...
123     posgp,weigp,dgdx,dvolum,
      istep, ...
124     iter,dsendc,dsen2dc,
      gstif);

```

```

125 end
126
127 %-----
128 % solve equations and update
129 %-----
130
131 asdis = gstif\gforce;
132
133 %-- update concentration field
134
135 con = con + asdis';
136
137 %-- for small deviations:
138 if(isolve == 1)
139 for ipoin=1:npoint
140 if(con(ipoin) <= 0.0001)
141 con(ipoin)=0.0001;
142 end
143 if(con(ipoin) >= 0.9999)
144 con(ipoin)=0.9999;
145 end
146 end
147
148 else
149 inrange=(con > 0.9999);
150 con(inrange) = 0.9999;
151
152 inrange=(con < 0.0001);
153 con(inrange) = 0.0001;
154 end
155
156 %--- check norm for convergence
157
158 normF = norm(gforce, 2);
159
160 if(normF <= toler)
161 break
162 end
163
164 end %end of Newton
165
166 %--- print out
167
168 if(mod(istep,nprnt) == 0)
169
170 fprintf('Done step: %5d\n',istep);
171
172 %fname=sprintf('time_%d.out',
173 istep);
174 %out1=fopen(fname,'w');
175 %for ipoin=1:npoint
176 %fprintf(out1,'%14.6e %14.6e %
177 14.6e\n',coord(ipoin,1), coord
178 (ipoin,2),con(ipoin));
179 %fclose(out1);
180
181 write_vtk_fem(npoin,nelem,nnode,
182 lnods,coord,istep,con);
183
184 end %if
185
186 end %istep
187
188 compute_time = etime(clock(), time0)
189
190 fprintf(out,'compute time: %7d\n',
191 compute_time);

```

Line numbers:

10:	Get wall clock time beginning of the execution.
14–18:	Unit names for input and output files.
21:	Solution flag: isolve = 1 for un-optimized solution, isolve = 2 for Matlab/Octave optimized solution.
23–30:	Time integration parameters.
25:	Number of time steps.
26:	Print frequency to output results to file.
27:	Time increment for numerical integration.
28:	Tolerance value for iterative solution.
29:	Number of maximum iterations.
31–39:	Material-specific parameters.
33:	Mobility coefficient.
34:	Gradient energy coefficient.
36–38:	Components of eigenstrains.
44–46:	Input FEM mesh and the control parameters.
48:	Number of total variables in the solution.
49:	Number of variables per element.
51:	Parameters for numerical integration of elements.
53–58:	If the solution is in optimized mode, precalculate the Cartesien derivatives of shape functions and area/volume of all elements.
59–62:	Calculate the applied loads if there is any.
68:	Model parameter: icase = 1 for single layer, icase = 2 double layer thin film morphology.

(continued)

70:	Prepare the concentration field for the desired thin film microstructure.
73–186:	Evolve microstructure.
78	Transfer nodal values of concentration from previous time step, into the array con_old.
83–89:	If the solution is in un-optimized mode, calculate the derivatives of the elastic-strain energy at the beginning of the time step.
91–97:	If the solution is in optimized mode, calculate the derivatives of the elastic-strain energy at the beginning of the time step.
100–164:	Newton–Raphson iterations.
108–115:	If the solution is in un-optimized mode, calculate the global stiffness matrix and global rhs, load, vectors.
118–125:	If the solution is in optimized mode, calculate the global stiffness matrix and global rhs, load, vectors.
131:	Solve system equations.
135:	Update nodal values.
137–154:	For small deviations from max and min values, reset the limits.
138–146:	For solution in un-optimized mode.
148–154:	For solution in optimized mode.
156–162:	Check convergence, if converged solution is reached, exit from Newton–Raphson iteration.
168–184:	If print frequency is reached, output the results to file.
172–179:	Open an output file and print the coordinates of the nodes and the nodal values to file. These lines are commented but, they can be changed, including the output format.
181:	Write the results in vtk file format for contour plots to be viewed by using Paraview.
188–190:	Calculate the total execution time and print it.

Function

bmats2.m

This function forms the strain matrix. It is optimized for Matlab/Octave.

Variable and array list:

nelem:	Number of elements.
nnode:	Number of nodes per element.
nstre:	Number of stress components.
nevab:	Number of element variables.
kgasp:	Integration point counter.
bmatx(nelem,nstre,nevab):	Strain matrix.

dgdx(nelem,mgaus,nstre,nnode):	Precalculated element Cartesian derivatives of shape functions.
---------------------------------------	---

Listing:

```

1  function [bmatx]=bmats2(dgdx,
2    nelem,nnode,nstre,nevab,kgasp)
3  format long;
4
5  bmatx = zeros(nelem,nstre,nevab);
6
7  ngash=0;
8
9  for inode=1:nnode
10
11  mgash=ngash+1;
12  ngash=mgash+1;
13
14  bmatx( :,1,mgash)=dgdx( :,kgasp,1,
15  inode);
16  bmatx( :,1,ngash)=0.0;
17  bmatx( :,2,mgash)=0.0;
18  bmatx( :,2,ngash)=dgdx( :,kgasp,2,
19  inode);
20  bmatx( :,3,mgash)=dgdx( :,kgasp,2,
21  inode);
22  bmatx( :,3,ngash)=dgdx( :,kgasp,1,
23  inode);
24
25  end
26
27
28  end %endfunction

```

Line numbers:

5:	Initialize strain matrix for all elements in the solution.
7:	Initialize the counter.
9–21:	Loop over element nodes.
11–12:	Counters for the position of Cartesian derivatives of shape functions in strain matrix.
14–19:	Place the Cartesian derivatives of shape functions into strain matrix for all elements.

Function

boundary_cond_v1.m

This function modifies the global stiffness matrix and the global rhs, load, vector for boundary conditions.

<i>Variable and array list:</i>	
npoin:	Number of total nodes in the simulation.
nvfix:	Total number of nodes that have prescribed boundary conditions.
ndofn:	DOFs per node.
gforce(ntotv):	Global rhs, load, vector.
nofix(nvfix):	List of nodes that have prescribed boundary conditions.
iffix(nvfix,ndofn):	List of constrained DOFs.
fixed(nvfix,ndofn):	Prescribed values for constrained DOFs.
gstif(ntotv,ntotv):	Global stiffness matrix.

Listing:

```

1 function[gstif,gforce] =
2   boundary_cond_v1(npoin,nvfix,
3     nofix,iffix, ...
4     fixed,ndofn,gstif,gforce)
5
6 format long;
7
8 ntotv=npoin*ndofn;
9
10 for ivfix = 1:nvfix
11 lnode = nofix(ivfix);
12
13 for idofn =1:ndofn
14 if(iffix(ivfix,idofn) == 1)
15 itotv=(lnode - 1)*ndofn +idofn;
16 gstif(itotv,jtotv ) = 0.0;
17 end
18
19 gstif(itotv,itotv) = 1.0;
20
21 gforce(itotv) = fixed(ivfix,idofn);
22 end
23 end
24 end
25
26 end %endfunction

```

Line numbers:

6:	Total number of variables.
8–24:	Loop over number of constrained nodes.
16:	Zero the rows belong to the DOFs of the constrained nodes in the global stiffness matrix.

19:	Insert the value of one to the diagonals of the corresponding DOFs in the global stiffness matrix.
21:	Assign prescribed values of displacements in the corresponding DOF of constrained nodes in the global rhs, load, vector.

Function

boundary_cond_v2.m

This function modifies the global stiffness matrix and the global rhs, load, vector for boundary conditions. It is optimized for Matlab/Octave.

<i>Variable and array list:</i>	
npoin:	Number of total nodes in the simulation.
nvfix:	Total number of nodes that have prescribed boundary conditions.
ndofn:	DOFs per node.
gforce(ntotv):	Global rhs, load, vector.
nofix(nvfix):	List of nodes that have prescribed boundary conditions.
iffix(nvfix,ndofn):	List of constrained DOFs.
fixed(nvfix, ndofn):	Prescribed values for constrained DOFs.
gstif(ntotv,ntotv):	Global stiffness matrix.

Listing:

```

1 function[gstif,gforce] =
2   boundary_cond_v2(npoin,nvfix,
3     nofix,iffix, ...
4     fixed,ndofn,gstif,gforce)
5
6 format long;
7
8 ntotv=npoin*ndofn;
9
10 for ivfix = 1:nvfix
11 lnode = nofix(ivfix);
12
13 for idofn =1:ndofn
14 if(iffix(ivfix,idofn) == 1)
15 itotv=(lnode - 1)*ndofn +idofn;
16 gstif(itotv,:) = 0.0;
17 gstif(itotv,itotv) = 1.0;
18
19 gforce(itotv) = fixed(ivfix,idofn);

```

```

19 end
20 end
21 end
22
23 end %endfunction

```

Line numbers:

6:	Total number of variables.
11–21:	Loop over number of constrained nodes.
15:	Zero the rows belong to the DOFs of the constrained nodes in the global stiffness matrix.
16:	Insert the value of one to the diagonals of the corresponding DOFs in the global stiffness matrix.
18:	Assign prescribed values of displacements in the corresponding DOF of constrained nodes in the global rhs, load, vector.

Function**chem_stiff_v3.m**

This function forms the global stiffness matrix and global rhs, load, vector for solution of Cahn–Hilliard equation by taking into account the elastic in homogeneities. It is in longhand format and is not optimized.

The function makes calls to the following functions:

- **sfr2.m**
- **jacob2.m**
- **free_energ_fem_v1.m**

Variable and array list:

npoint:	Total number of nodes in the solution.
nelem:	Total number of elements in the solution.
nnode:	Number of nodes per element.
nstre:	Number of stress components.
ndime:	Number of Cartesian coordinate dimension.
ndofn:	Number of DOF per node.
ngaus:	Order of numerical integration.
ntype:	Solution type, ntype = 1 for plane-stress, ntype = 2 for plane-strain.
mobil:	Mobility coefficient.
grcoef:	Gradient energy coefficient.
dtime:	Time increment in numerical integration.
istep:	Current time integration step.
iter:	

	Current Newton–Raphson iteration number.
con(ntotv):	Nodal concentration values, ntotv = npoin × ndofn.
con_old (ntotv):	Nodal concentration values from previous time step.
posgp(ngaus):	Position of integration points.
weigp(ngaus):	Weights for numerical integration.
dsende(ntotv):	First derivative of elastic strain energy.
dsen2dc (ntotv):	Second derivative of elastic strain energy
gforce(ntotv):	Global rhs, load, vector.
lnods(nelem, nnode):	Element nodal connectivity list.
coord(npoin, ndime):	Cartesian coordinates of nodes.
gstif(ntotv, ntotv):	Global stiffness matrix.

Listing:

```

1 function [gstif,gforce]=chem_
stiff_v3(npoin,nelem,nnode,
nstre,ndime, ...
2           ndofn,ngaus,ntype,lnods,
coord, ...
3           mobil,grcoef,con,con_old,
dtime, ...
4           posgp,weigp,istep,iter,
dsende, ...
5           dsen2dc,gstif)
6 format long;
7
8 %=====
9 %   global and local variables: &
10 %=====
11
12 ntotv=npoin*ndofn;
13 nevab=nnode*ndofn;
14 %--
15 ngaus2=ngaus;
16 if(nnode == 3)
17 ngaus2=1;
18 end
19 gforce = zeros(ntotv,1);
20 for ielem=1:nelem
21
22 %=====
23 %   initialize elements stiffness %
24 %           & rhs : %
25 %=====

```

```

26
27 %--- stiffness matrices:
28
29 if(iter == 1)
30
31 for inode=1:nnode
32 for jnode=1:nnode
33
34 kcc(inode,jnode)=0.0;
35 kcm(inode,jnode)=0.0;
36 kmm(inode,jnode)=0.0;
37 kmc1(inode,jnode)=0.0;
38 kmc(inode,jnode)=0.0;
39
40 end
41 end
42 end %if
43
44 %--- rhs
45 for inode =1:nnode
46 eload1(inode) = 0.0;
47 eload2(inode) = 0.0;
48 end
49 for ievab=1:nevab
50 eload(ievab) =0.0;
51 end
52
53
54
55 %=====&
56 %----- elemental values &
57 %=====&
58
59 for inode =1:nnode
60
61 lnode = lnods(ielem,inode);
62 cv(inode) = con(lnode);
63 cm(inode) = con(npoin+lnode);
64 cv_old(inode) = con_old(lnode);
65
66 end
67
68 %--- coords of the element nodes
69 for inode=1:nnode
70 lnode=lnods(ielem,inode);
71 for idime=1:ndime
72 elcod(idime,inode)=coord(lnode,
idime);
73 end
74 end
75
76 %=====
77 % integrate element stiffness %
78 % & rhs :
79 %=====
80
81 kgasp=0;
82
83 for igaus=1:ngaus
84 exisp=posgp(igaus);
85 for jgaus=1:ngaus2
86 etasp =posgp(jgaus);
87 if(nnode ==3)
88 etasp=posgp(ngaus+igaus);
89 end
90
91 kgasp=kgasp+1;
92 [shape,deriv]=sfr2(exisp,etasp,
nnode);
93 [cartd,djacb,gpcod]=jacob2
(ielem,elcod,kgasp,shape, ...
deriv,nnode,ndime);
94
95
96 dvolu=djacb*weigp(igaus)*weigp
(jgaus);
97
98 if(nnode == 3)
99 dvolu=djacb*weigp(igaus);
100 end
101
102 %== values at the gauss points:
103
104 cvgp = 0.0;
105 cmgp = 0.0;
106 cv_ogp = 0.0;
107
108 for inode=1:nnode
109
110 cvgp = cvgp + cv(inode)*shape
(inode);
111 cmgp = cmgp + cm(inode)*shape
(inode);
112 cv_ogp = cv_ogp + cv_old(inode)*sha
pe(inode);
113
114 end
115
116 %== chemical potential:
117
118 [dfdc,df2dc]=free_energ_fem_v1
(cvgp);

```

```

119
120 %
121
122 if(iter == 1)
123
124 %--- kcc matrix:
125
126 for inode=1:nnode
127 for jnode=1:nnode
128
129 kcc(inode,jnode) = kcc(inode,
130 jnode) + ...
131 shape(inode)*shape(jnode)*dvolu;
132 end
133
134 %--- add strain energy contribution
135 to kcm matrix
136 for inode=1:nnode
137 for jnode=1:nnode
138
139 kcc(inode,jnode) =kcc (inode,
140 jnode) + dtim*mobil*dsen2dc
141 (ielem,kgas... *
142 (cartd(1,inode)*cartd(1,
143 jnode)+...
144 cartd(2,inode)*cartd(2,
145 jnode)).*d volu;
146 end
147
148 %--- kcm matrix:
149 for inode=1:nnode
150 for jnode=1:nnode
151 for idime=1:ndime
152
153 kcm(inode,jnode) = kcm (inode,
154 jnode) + ...
155 dtim*mobil*cartd(idime,
156 inode)* ...
157 cartd(idime,jnode)*dvolu;
158 end
159
160
161 %--- kmm matrix:
162
163 for inode=1:nnode
164 for jnode=1:nnode
165 kmm(inode,jnode) = kmm(inode,
166 jnode) + ...
167 shape(inode)*shape(jnode)
168 *dvolu;
169 end
170
171 %--- kmc matrix:
172 for inode=1:nnode
173 for jnode=1:nnode
174 for idime=1:ndime
175
176 kmc(inode,jnode) = kmc(inode,
177 jnode) - ...
178 grcoef*cartd(idime,
179 inode)* ...
180 cartd(idime,jnode)*dvolu;
181 end
182
183 for inode=1:nnode
184 for jnode=1:nnode
185
186 kmc(inode,jnode) = kmc(inode,
187 jnode) - ...
188 df2dc*shape(inode)*shape
189 (jnode)*dvolu;
190
191
192 end %if iter
193
194 %-----
195 % element rhs
196 %-----
197
198 for inode=1:nnode
199 eload1(inode) = eload1(inode) - ...
200 shape(inode)*(cvgp-cv_ogp)*
201 dvolu;
202

```

```

203 for inode=1:nnode           244 end %jgaus
204 for jnode=1:nnode          245
205 for idime=1:ndime          246 %-----
206
207 eload1(inode) = eload1(inode) - 247 %%assemble element stiffness
208   dtimes*mobil...           248 and rhs:
209   *cmgp*shape(jnode)*cartd 249 %-----
210   (idime,inode)*...          250 if(iter == 1)
211   cartd(idime,jnode)*dvolu; 251
212 end                         252 for inode=1:nnode
213 end                         253 ievab=nnode+inode;
214
215 %--- add strain energy contribution: 254 for jnode=1:nnode
216
217 for inode=1:nnode           255 jevab=nnode+jnode;
218 for jnode=1:nnode          256
219
220 eload1(inode) = eload1      257 estif(inode,jnode)=kcc(inode,
221   (inode) - dtimes*mobil*dsemdc 258   jnode);
222   (ielem,kgas)*...             259 estif(inode,jevab)=kcm(inode,
223   (cartd(1,inode)*cartd(1, 260   jnode);
224   jnode) + ...                261 estif(ievab,jnode)=kmc(inode,
225   cartd(2,inode)*cartd(2, 262   jnode);
226   jnode))*shape(jnode)*dvolu; 263 end
227 end                         264 end %if
228 end                         265
229
230 for inode=1:nnode           266 %rhs
231 for jnode=1:nnode          267 for inode=1:nnode
232 for idime=1:ndime          268 ievab=nnode+inode;
233
234 eload2(inode) = eload2(inode) - 269 eload(inode)=eload1(inode);
235   shape(inode)*(cmgp-dfdc)    270 eload(ievab)=eload2(inode);
236   *dvolu;                     271 end
237
238 end                         272
239 end                         273 %=====
240 end                         274 % form global stiffness and rhs
241 end                         275 %=====
242
243 end %igaus                  276
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287

```

```

288     (ielem,jnode)), ...
289     estif(ievab,jevab),
290     ntotv,ntotv);
291 end
292 end
293
294 end %if iter
295
296 %-- rhs
297
298 for idofn=1:ndofn
299 for inode=1:nnode
300 ievab=(idofn-1)*nnode+inode;
301
302 gforce = gforce + sparse(((idofn-1)
303 *npoin+lnods(ielem,inode)),1...
304 ,eload(ievab),ntotv,1);
305 end
306 end
307
308 end % ielem
309
310 end %endfunction
311

```

102–114:	Calculate the values of variables at the integration point, Eq. 6.86.
118:	Calculate derivatives of free energy.
122–192:	If iter = 1, form submatrices of element stiffness matrix.
124–132:	Form first term of K_{ij}^{cc} matrix, Eq. 6.89.
136–143:	Form second term of K_{ij}^{cc} matrix, Eq. 6.89.
149–158:	Form K_{ij}^{cu} matrix, Eq. 6.90.
163–168:	Form $K_{ij}^{\mu\mu}$ matrix, Eq. 6.92.
172–181:	Form second term of $K_{ij}^{\mu c}$ matrix, Eq. 6.91.
183–190:	Form first term of $K_{ij}^{\mu c}$ matrix, Eq. 6.91.
198–201:	Form first term of first row of rhs, load, vector, Eq. 6.87.
203–213:	Form last term of first row of rhs, load, vector, Eq. 6.87.
215–224:	Form second term of first row of rhs, load vector, Eq. 6.87.
227–230:	Form first and second terms of second row of rhs, load, vector, Eq. 6.87.
232–241:	Form last term of second row of rhs, load, vector, Eq. 6.87.
250–264:	If iter = 1, assemble element stiffness matrix from submatrices, Eq. 6.88.
267–271:	Assemble element rhs, load, vector.
277–294:	If iter = 1, assemble element stiffness matrix into global stiffness matrix.
296–306:	Assemble element rhs, load, vector to global rhs, load vector.

Line numbers:

12:	Total number of variables.
13:	Total number of variables per element.
15–18:	Order of numerical integration, depending on the element type.
19:	Initialize global, rhs, load, vector.
20–308:	Loop over elements.
29–42:	If iter = 1, initialize element stiffness submatrices.
45–51:	Initialize element rhs, load, vector.
59–66:	Calculate element nodal values.
68–74:	Calculate Cartesian coordinates of element nodes.
77–244:	Integrate element stiffness matrix and rhs, load, vector.
84, 86, 88:	Coordinates of integration points in the local coordinate system.
91:	Increment integration point counter.
92:	Calculate shape functions and their derivative values.
93–94:	Calculate Cartesian derivatives of shape functions.
96–100:	Calculate element area/volume.

Function**chem._stiff_v4.m**

This function forms the global stiffness matrix and global rhs, load, vector for solution of Cahn–Hilliard equation by taking into account the elastic in homogeneities. It is optimized for Matlab/Octave.

The function makes calls to the following functions:

- **sfr2.m**
- **free_energ_fem_v2.m**

Variable and array list:

npoin:	Total number of nodes in the solution.
nelem:	Total number of elements in the solution.
nnode:	Number of nodes per element.
nstre:	Number of stress components.

(continued)

ndime:	Number of Cartesian coordinate dimension.
ndofn:	Number of DOF per node.
ngaus:	Order of numerical integration.
ntype:	Solution type, ntype = 1 for plane-stress, ntype = 2 for plane-strain.
mobil:	Mobility coefficient.
grcoef:	Gradient energy coefficient.
dtime:	Time increment in numerical integration.
istep:	Current time integration step.
iter:	Current Newton–Raphson iteration number.
con(ntotv):	Nodal concentration values, ntotv = npoin × ndofn.
con_old(ntotv):	Nodal concentration values from previous time step.
posgp(ngaus):	Position of integration points.
weigp(ngaus):	Weights for numerical integration.
dsendc(ntotv):	First derivative of elastic strain energy.
dsen2dc(ntotv):	Second derivative of elastic strain energy
gforce(ntotv):	Global rhs, load, vector.
lnods(nelem, nnodes):	Element nodal connectivity list.
coord(npoin, ndime):	Cartesian coordinates of nodes.
gstif(ntotv, ntotv):	Global stiffness matrix.
dvolum(nelem, mgaus):	Precalculated area/volume of elements at their integration points.
dgdx(nelem, mgaus,nstre, nnodes):	Precalculated Cartesian derivatives of shape functions for all elements in the solution.

```

13
14 ntotv=npoin*ndofn;
15 nevab=nnode*ndofn;
16 %---
17 ngaus2=ngaus;
18 if(nnod == 3)
19 ngaus2=1;
20 end
21
22 %=====
23 % initialize elements stiffness %
24 % & rhs: %
25 %=====
26
27 if(iter == 1)
28
29 %--- stiffness matrices:
30 estif1 = zeros(nelem,nnode,nevab);
31 estif2 = zeros(nelem,nnode,nevab);
32 estif = zeros(nelem,nevab,nevab);
33
34 kcc = zeros(nelem,nnode,nnode);
35 kcm = zeros(nelem,nnode,nnode);
36 kmm = zeros(nelem,nnode,nnode);
37 kmc = zeros(nelem,nnode,nnode);
38 kmc1 = zeros(nelem,nnode,nnode);
39
40 end %iter
41
42 %--- rhs
43 eload1 = zeros(nelem,nnode);
44 eload2 = zeros(nelem,nnode);
45 eload = zeros(nelem,nevab);
46 gforce= zeros(ntotv,1);
47
48 %---composition:
49 cv = zeros(nelem,nnode);
50 cm = zeros(nelem,nnode);
51 cv_old = zeros(nelem,nnode);
52
53 %=====
54 %--- elemental values &
55 %=====
56
57 for inode=1:nnode
58 lnode = lnods( :,inode);
59 cv( :,inode) = con(lnode);
60 cm( :,inode) = con(npoin+lnode);
61 cv_old( :,inode) = con_old(lnode);
62 end

```

Listing:

```

1 function [gstif,gforce]=chem_
stiff_v4(npoin,nelem,nnode,
nstre, ...
2 ndime,ndofn,ngaus,ntype, ...
3 lnods,coord,mobil,grcoef, ...
4 con,con_old,dtime,posgp, ...
5 weigp,dgdx,dvolum,istep, ...
6 iter,dsendc,dsen2dc,gstif)
7
8 format long;
9
10 %=====
11 % global and local variables: &
12 %=====

```

```

63
64 %=====
65 %   integrate element stiffness %
66 %           & rhs: %
67 %=====
68
69 kgasp=0;
70 for igaus=1:ngaus
71 exisp=posgp(igaus);
72 for jgaus=1:ngaus
73 etasp =posgp(jgaus);
74 if(nnode ==3)
75 etasp=posgp(ngaus+igaus);
76 end
77
78 kgasp=kgasp+1;
79 [shape,deriv]=sfr2(exisp,etasp,
nnode);
80
81 %== values at the gauss points:
82
83 cvgp=zeros(nelem,1);
84 cmgp=zeros(nelem,1);
85 cv_ogp=zeros(nelem,1);
86
87 for inode=1:nnode
88
89 cvgp=cvgp + cv( :,inode)*shape
(inode);
90 cmgp =cmgp + cm( :,inode)*shape
(inode);
91 cv_ogp =cv_ogp + cv_old( :,inode)
*shape(inode);
92
93 end
94
95 %== chemical potential:
96
97 [dfdc,df2dc]=free_energ_fem_v2
(nelem,cvgp);
98
99 %
100
101 if(iter == 1)
102
103 for inode = 1:nnode
104 for jnode = 1:nnode
105
106 %--- kcc matrix:
107
108 kcc( :,inode,jnode)=kcc( :,inode,
jnode) + ...
109 shape(inode)*shape(jnode).*dv
olum( :,kgasp);
110
111
112 %--- add strain energy contribution
to kcm matrix
113
114 kcc( :,inode,jnode) =kcc( :,inode,
jnode) + dtime*mobil*dsen2dc( :,,
kgasp) ...
115 .* (dgdx( :,kgasp,1,inode) .
*dgdx( :,kgasp,1,jnode)+...
dgdx( :,kgasp,2,inode) .
*dgdx( :,kgasp,2,jnode)) .
*dvolum( :,kgasp);
116
117
118
119 %--- kcm matrix:
120
121 kcm( :,inode,jnode) =kcm( :,inode,
jnode) + dtime*mobil* ...
122 (dgdx( :,kgasp,1,inode) .
*dgdx( :,kgasp,1,jnode)+...
dgdx( :,kgasp,2,inode) .
*dgdx( :,kgasp,2,jnode)) .
*dvolum( :,kgasp);
123
124
125
126 %--- kmm matrix:
127
128 kmm( :,inode,jnode)=kmm( :,,
inode,jnode) + ...
129 shape(inode)*shape(jnode) .
*dvolum( :,kgasp);
130
131 %--- kmc matrix:
132
133 kmc( :,inode,jnode)=kmc( :,inode,
jnode) -grcoef* ...
134 (dgdx( :,kgasp,1,inode) .
*dgdx( :,kgasp,1,jnode)+...
dgdx( :,kgasp,2,inode) .
*dgdx( :,kgasp,2,jnode)) .
*dvolum( :,kgasp);
135
136
137 kmc( :,inode,jnode) =kmc( :,inode,
jnode) - df2dc( : )* ...
138 shape(inode)*shape(jnode) .

```

```

          *dvolum( :,kgasp) ;
139      end
140      end
141      end %iter
142      %-----
143      % element rhs
144      %-----
145
146
147
148      for inode = 1:nnode
149
150      eload1( :,inode)=eload1( :,inode)
151      - shape(inode)* ...
152          (cvgp( :) - cv_ogp( :)) .
153          *dvolum( :,kgasp) ;
154
155
156      for jnode = 1:nnode
157
158      eload1( :,inode) = eload1( :,inode)
159      - dtim*mobil*cmgp( :).* ...
160          (dgdx( :,kgasp,1,inode).*dgdx
161          ( :,kgasp,1,jnode) + ...
162          dgdx( :,kgasp,2,inode).*dgdx
163          ( :,kgasp,2,jnode))* ...
164
165          shape(jnode).*dvolum( :,
166          kgasp);
167
168
169
170      eload2( :,inode) = eload2( :,inode)
171      + grcoef*cvgp( :).* ...
172          (dgdx( :,kgasp,1,inode).*dgdx( :,
173          kgasp,1,jnode) + ...
174          dgdx( :,kgasp,2,inode) .
175          *dgdx( :,kgasp,2,jnode))* ...
176
177      end %igaus
178      end %jgaus
179
180
181      % assemble element stiffness
182      %-
183
184      if(iter == 1)
185
186          for inode=1:nnode
187              ievab=nnode+inode;
188
189          jevab=nnode+jnode;
190
191          estif( :,inode,jnode)=kcc
192          ( :,inode,jnode);
193
194          estif( :,inode,jevab)=kcm
195          ( :,inode,jnode);
196
197          estif( :,ievab,jnode)=kmc
198
199          estif( :,ievab,jevab)=kmm
200          ( :,inode,jnode);
201
202
203
204
205
206
207
208
209
210
211
212      if( iter == 1)
213
214
215
216
217

```

```

218 for jnode=1:nnode
219   jevab = (jdofn-1)*nnode+jnode;
220
221   gstif = gstif + sparse(((idofn-1)
222     *npoin+lnods( :, inode)), ...
223       ((jdofn-1)*npoin+lnods
224         ( :, jnode)), ...
225       estif( :, ievab, jevab), ntotv,
226         ntotv);
227   end
228 end
229 end %iter
230
231 %--- rhs
232
233 for idofn=1:ndofn
234 for inode=1:nnode
235   ievab=(idofn-1)*nnode+inode;
236
237   gforce = gforce +sparse(((idofn-1)
238     *npoin+lnods( :, inode)), 1 ...
239       ,eload( :, ievab), ntotv, 1);
240   end
241 end
242
243 end %endfunction
244

```

Line numbers:

14:	Number of total variables.
15:	Number of variables per element.
17–20:	Order of numerical integration, depending on the element type.
27–40:	If iter = 1, initialize element stiffness matrix and submatrices for all elements in the solution.
42–47:	Initialize elements rhs, load, vector.
48–52:	Initialize array for nodal variables for all elements.
57–62:	Calculate the nodal variable values for all elements.
65–178:	Integrate stiffness matrix and rhs, load, vector of all elements.
71, 73, 75:	Coordinates of the integration points in local coordinate system.
78:	Increment the integration point counter.

81–93:	Calculate the variable values at the integration point, Eq. 6.86.
97:	Calculate the derivative of free energy for all elements.
101–142:	If iter = 1, form element stiffness matrix submatrices.
108–109:	Form the first term of K_{ij}^{cc} matrix, Eq. 6.89.
114–116:	Form the second term of K_{ij}^{cc} matrix, Eq. 6.89.
121–123:	Form the $K_{ij}^{c\mu}$ matrix, Eq. 6.90.
128–129:	Form the $K_{ij}^{\mu\mu}$ matrix, Eq. 6.92.
133–135:	Form the second term of $K_{ij}^{\mu c}$ matrix, Eq. 6.91.
137–138:	Form the first term of $K_{ij}^{\mu c}$ matrix, Eq. 6.91.
150–151:	Form the first term of first row of rhs, load, vector, Eq. 6.87.
153–154:	Form the first two terms of second row of rhs, load, vector, Eq. 6.87.
158–161:	Form the last term of first row of rhs, load, vector, Eq. 6.87.
165–168:	Form the second term of first row of rhs, load, vector, Eq. 6.87.
170–173:	Form the second term of second row of rhs, load, vector, Eq. 6.87.
184–199:	If iter = 1, assemble element stiffness matrix, for all elements, Eq. 6.88.
202–206:	Assemble rhs, load, vector of all elements.
212–229:	If iter = 1, assemble element stiffness matrix of all elements into the global stiffness matrix.
231–241:	Assemble element rhs, load vector of all elements into the global rhs, load vector.

Function**dbe2.m**

This function multiplies the elasticity matrix and the strain matrix. It is optimized for Matlab/Octave.

Variable and array list:

nelem:	Total number of elements in the solution.
nevab:	Total number of variables per element.
nstre:	Number of stress components.
bmatx(nelem, nstre,nevab):	Strain matrix.
dmatx(nelem, nstre,nstre):	Elasticity matrix.
dbmat(nelem, nstre,nevab):	Result of strain and elasticity matrix multiplication.

Listing:

```

1 function [dbmat] = dbe2(nelem,
2 nevab,nstre,bmatx,dmatx)
3 format long;
4
5 %multiply bmatx with dmatx
6
7 dbmat = zeros(nelem,nstre,nevab);
8
9 for istre=1:nstre
10 for ievab=1:nevab
11 %dbmat( :,istre,ievab)=0.0;
12 for jstre=1:nstre
13 dbmat( :,istre,ievab)=dbmat( :,istre,
14 ievab)+dmatx( :,istre,
15 jstre) ...
16 .*bmatx( :,jstre,ievab);
17 end
18 end
19 end %endfunction

```

Line numbers:

7: Initialize matrix that holds multiplication results for all elements in the solution.

9–17: Carry out the multiplication of strain matrix and elasticity matrix for all elements.

Function

elastic energ v1.m

This function solves displacements at the nodal points, evaluates the stress-strain fields, and calculates the derivatives of the elastic strain energy. It is longhand format and is not optimized for Matlab/Octave.

The function makes calls to the following functions:

- stiffness2.m
 - boundary_cond_v1.m
 - stress2.m

Variable and array list:

npoint:	Total number of nodes.
nelem:	Total number of elements.

nnode:	Number of nodes per element.
nstre:	Number of stress components.
ndime:	Number of Cartesian coordinate dimension.
ndofn:	Number of DOF per node.
ngaus:	Order of numerical integration.
nvfix:	Number nodes with prescribed displacements.
ntype:	Solution type, ntype = 1 for plane-stress, ntype = 2 for plane-strain.
nofix(nvfix):	List of nodes with prescribed displacements.
posgp(ngaus):	Position of integration points.
weigp(ngaus):	Weights for numerical integration.
matno(nelem):	Material property types.
ncomp(npoin):	Nodal values of concentration.
eigen(nstre):	Eigenstrain values.
gforce(ntotv):	Global force, rhs, vector.
props(nmats, nprop):	Material properties, nmats is number of materials and nprop is number of properties.
lnods(nelem, nnode):	Element connectivity list.
coord(npoin, ndime):	Cartesian coordinates of nodes.
iffix(nvfix, ndime):	List of constrained DOFs.
fixed(nvfix, ndofn):	Prescribed values for constrained DOFs.
dsendc(nelem, mgaus):	First derivative of elastic strain energy at element integration points, mgaus total number of integration points per element.
dsen2dc (nelem, meaus):	Second derivative of elastic strain energy at element integration points.

Listing:

```

1 function [dsendc,dse2dc] =elastic_
energ_v1(npoin,nelem,nnode,nstre,
ndime,ndofn, ...
2       ngaus,ntype,lnods,matno,
coord,props, ...
3       nvfix,nofix,iifix,fixed, posgp,
weigp, ...
4       ncomp,eigen,gforce)
5
6 format long;
7
8 [gstif,iforce]=stiffness2(npoin,
nelem,nnode,nstre,ndime,ndofn, ...

```

```

9      ngaus,ntype,lnods,matno,
10     coord,props, ...
11
12 tforce = gforce + iforce;
13
14 % apply boundary conditions:
15
16 [gstif,tforce] =boundary_cond_v1
17 (npoin,nvfix,nofix,iffix, ...
18     fixed,ndofn,gstif,tforce);
19
20 %--- solve for displacements:
21
22 asdis = gstif\ tforce;
23
24
25 %% calculate derivatives of strain
26 %% energy:
27 [dsendc,dse2dc] = stress2(asdis,
28 nelem,npoin,nnode,ngaus,nstre,
29 props, ...
30     ntype,ndofn,ndime,lnods,
31     matno,coord,posgp,weigp, ...
32     ncomp,eigen);
33 end %endfunction

```

Line numbers:

8–10:	Calculate the global stiffness matrix and rhs, load vector resulting from internal stress field.
12:	Add internal rhs, load, vector to global rhs, load, vector.
16–17:	Modify the global stiffness matrix and global rhs, load, vector for prescribed boundary conditions.
22:	Solve system of equations for nodal displacements.
27–29:	Calculate stress–strains fields and the derivatives of elastic strain energy.

Function**elastic_energ_v2.m**

This function solves displacements at the nodal points, evaluates the stress–strain fields, and calculates the derivatives of the elastic strain

energy. The code is optimized for Matlab/Octave.

The function makes calls to the following functions:

- **stiffness3.m**
- **boundary_cond_v2.m**
- **stress3.m**

Variable and array list:

npoint:	Total number of nodes.
nelem:	Total number of elements.
nnode:	Number of nodes per element.
nstre:	Number of stress components.
ndime:	Number of Cartesian coordinate dimension.
ndofn:	Number of DOF per node.
ngaus:	Order of numerical integration.
nvfix:	Number nodes with prescribed displacements.
ntype:	Solution type, ntype = 1 for plane-stress, ntype = 2 for plane-strain.
nofix(nvfix):	List of nodes with prescribed displacements.
posgp(ngaus):	Position of integration points.
weigp(ngaus):	Weights for numerical integration.
matno(nelem):	Material property types.
ncomp(npoin):	Nodal values of concentration.
eigen(nstre):	Eigenstrain values.
gforce(ntotv):	Global force, rhs, vector.
props(nmats, nprop):	Material properties, nmats is number of materials and nprop is number of properties.
lnods(nelem, nnode):	Element connectivity list.
coord(npoin, ndime):	Cartesian coordinates of nodes.
iffix(nvfix, ndime):	List of constrained DOFs.
fixed(nvfix, ndofn):	Prescribed values for constrained DOFs.
dse2dc(nelem, mgaus):	First derivative of elastic strain energy at element integration points, mgaus total number of integration points per element.
dse2dc(nelem, mgaus):	Second derivative of elastic strain energy at element integration points.
dvolum(nelem, mgaus):	Precalculated area/volume of elements at their integration points.
dgdx(nelem, mgaus,nstre, nnode):	Precalculated Cartesian derivatives of shape functions of all elements in the solution.

Listing:

```

1 function [dsendc,dse2dc] =elastic_
energ_v2(npoin,nelem,nnode,nstre,
ndime,ndofn, ...
2     ngaus,ntype,lnods,matno,
coord,props, ...
3     nvfix,nofix,iffix,fixed,
posgp,weigp, ...
4     dgdx,dvolum,ncomp,eigen,
gforce);
5 format long;
6
7
8 [gstif,iforce] =stiffness3(npoin,
nelem,nnode,nstre,ndime,ndofn, ...
9     ngaus,ntype,lnods,matno,
coord,props, ...
10    nvfix,nofix,iffix,fixed,
posgp,weigp, ...
11    dgdx,dvolum,ncomp,eigen);
12
13
14 tforce = gforce + iforce;
15
16 % apply boundary conditions:
17
18 [gstif,tforce] =boundary_cond_v2
(npoin,nvfix,nofix,iffix, ...
19     fixed,ndofn,gstif,tforce);
20
21 %--- solve for displacements:
22
23
24 asdis = gstif\ tforce;
25
26
27 %-- calculate derivatives of strain
energy:
28
29 [dsendc,dse2dc] = stress3(asdis,
nelem,npoin,nnode,ngaus,nstre,
props, ...
30     ntype,ndofn,ndime,lnods,
matno,coord,posgp,weigp, ...
31     dgdx,dvolum,ncomp,eigen);
32
33

```

34
35 end %endfunction

Line numbers:

8–10:	Calculate the global stiffness matrix and rhs, load vector resulting from internal stress field.
14:	Add internal rhs, load, vector to global rhs, load, vector.
18–19:	Modify the global stiffness matrix and global rhs, load, vector for prescribed boundary conditions.
24:	Solve system of equations for nodal displacements.
29–31:	Calculate stress–strains fields and the derivatives of elastic strain energy.

Function**stiffness2.m**

This function forms the global stiffness matrix and global rhs, load, vector resulting from internal stress field for the solution of nodal displacements. It is in longhand format and is not optimized.

The function makes calls to the following functions:

- **modps.m**
- **sfr2.m**
- **jacob2.m**
- **bmats.m**
- **dbe.m**

Variable and array list:

npoin:	Total number of nodes.
nelem:	Total number of elements.
nnode:	Number of nodes per element.
nstre:	Number of stress components.
ndime:	Number of Cartesian coordinate dimension.
ndofn:	Number of DOFs per node.
ngaus:	Order of numerical integration.
ntype:	Solution type, ntype = 1 for plane-stress, ntype = 2 for plane-strain.
posgp(ngaus):	Position of integration points.
weigp(ngaus):	Weights of integration points.

(continued)

ncomp (npoin):	Nodal values of composition.
matno (nelem):	Material number of elements.
eigen(nstres):	Eigenstrain components.
props(nmats, nprop):	Material properties, nmats is number of materials and nprop is number of properties.
coord(npoin, ndime):	Cartesian coordinates of nodes.
lnods(nelem, nnode):	Nodal connectivity list of elements.
iload(ntotv):	Global rhs, load vector resulting from internal stress field, ntotv = npoin \times ndofn.
gstif(ntotv, ntotv):	Global stiffness matrix

Listing:

```

1 function [gstif,iload]=stiffness2
2     (npoin,nelem,nnode,nstre,ndime,
3      ndofn, ...
4         ngaus,ntype,lnods,matno,
5          coord,props, ...
6          posgp,weigp,ncomp,eigen)
7      format long;
8
9      %-- order of integration:
10
11      ngaus2=ngaus;
12      if(nnnode == 3)
13          ngaus2=1;
14      end
15
16      %--- Initialize the global
17      stiffness:
18      nevab=nnode*ndofn;
19      ntotv=npoin*ndofn;
20
21      %--- Element stiffness & loads:
22      for ielem=1:nelem
23
24          %-- Initialize element stiffness
25          & load vector:
26          for ievab=1:nevab
27              eload(ievab)=0.0;
28              for jevab=1:nevab
29                  estif(ievab,jevab)=0.0;
30              end
31          end
32
33          %--- Form elasticity matrix
34          for phases:
35              mtype =1;
36              [dmatxm] =modps (mtype,ntype,nstre,
37              props);
38
39          %-- for the second phase:
40          mtype =2;
41          [dmatxp] =modps (mtype,ntype,
42          nstre,props);
43
44          %--- nodal values:
45          for inode=1:nnode
46              lnode =lnods(ielem,inode);
47              cv(inode)=ncomp(lnode);
48
49          %--- Coordinates of element nodes:
50
51          for inode=1:nnode
52              lnode=lnods(ielem,inode);
53              for idime=1:ndime
54                  elcod(idime,inode)=coord(lnode,
55                  idime);
56              end
57
58          %--- Integrate the element
59          stiffness :
60
61              kgasp=0;
62              for igaus=1:ngaus
63                  exisp=posgp(igaus);
64                  for jgaus=1:ngaus2
65                      etasp =posgp(jgaus);
66                      if(nnnode ==3)
67                          etasp=posgp(ngaus+igaus);
68                      end
69              kgasp=kgasp+1;
70
71              [shape,deriv]=sfr2(exisp,etasp,
72              nnode);
72              [cartd,djacb,gpcod]=jacob2(ielem,
73              elcod,kgasp,shape,deriv,nnode,
```

```

    ndime);
73 [bmatx]=bmats(cartd,shape,inode);
74
75
76 dvolu=djacb*weigp(igaus)*weigp
77 (jgaus);
78 if(nnode == 3)
79 dvolu=djacb*weigp(igaus);
80 end
81
82 %--- composition at the integration
83 %points and elasticity matrix:
84 cvgp = 0.0;
85
86 for inode=1:nnode
87 cvgp = cvgp +cv(inode)*shape
88 (inode);
89 end
90
91 %--- calculate composite elasticity
92 %matrix:
93
94 for istre=1:nstre
95 for jstre=1:nstre
96
97 dmatx(istre,jstre) = dmatxm(istre,
98 jstre) + cvgp *(dmatxp(istre,
99 jstre) -...
100 dmatxm(istre,jstre));
101 end
102 end
103
104 [dbmat] =dbe(nevab,nstre,bmatx,
105 dmatx);
106 %---Form element stiffness:
107
108 for ievab=1:nevab
109 for jevab=1:nevab
110 for istre=1:nstre
111
112 estif(ievab,jevab)=estif(ievab,
113 jevab)+bmatx(istre,ievab)*...
114 dbmat(istre,jevab)*dvolu;
115
116
117
118
119 %---- load vector due to eigenstress:
120
121
122 % extrapolate eigen strains from nodes
123 % to gauss points:
124 for istre=1:nstre
125 istran(istre) = -eigen(istre)
126 *cvgp;
127
128 %-- internal stress:
129
130 for istre=1:nstre
131 istres(istre) = 0.0;
132 for jstre=1:nstre
133 istres(istre) = istres(istre) +
134 dmatx(istre,jstre).*istran
135 (jstre);
136
137 %-- form load vector:
138
139 for inode = 1:nnode
140 ievab=(inode-1)*ndofn+1;
141 jevab=(inode-1)*ndofn+2;
142
143 eload(ievab)=eload(ievab) -(cartd
144 (1,inode)*istres(1) ...
145 *cvtd(2,inode)*istres(3))
146 *dvolu;
147
148 end
149 end %jgaus
150 end %igaus
151
152
153 %--- assemble element stiffness and
154 %load vectors into
155 %--- global stiffness matrix and

```

```

global load vectors
155
156
157 for inode =1:nnode
158 lnode =lnods(ielem,inode);
159 for idofn=1:ndofn
160 ievab=(inode-1)*ndofn+idofn;
161 itotv=(lnode-1)*ndofn+idofn;
162
163 iload= iload + sparse(itotv,1,eload
    (ievab),ntotv,1);
164
165 for jnode =1:nnode
166 knode = lnods(ielem,jnode);
167 for jdofn =1:ndofn
168 jevab=(jnode-1)*ndofn+jdofn;
169 jtov=(knode-1)*ndofn+jdofn;
170
171 gstif = gstif +sparse(itotv,jtov,
    estif(ievab,jevab),ntotv,ntotv);
172
173 end
174 end
175 end
176 end
177 end %ielem
178
179 end %endfunction

```

Line numbers:

8–11:	Order of integration, depending on the element type.
14:	Number of variables per element.
15:	Total number of variables.
17:	Initialize global stiffness matrix.
18:	Initialize global rhs, internal load, vector.
22–177:	Loop over elements.
26–31:	Initialize element rhs, load, vector and stiffness matrix.
35–37:	Form the elasticity matrix for the matrix phase.
38–41:	Form the elasticity matrix for the second phase.
44–47:	Get the nodal values of concentration.
51–56:	Calculate the Cartesian coordinates of element nodes.
58–150:	Numerical integration of element stiffness matrix and rhs, load, vector.
62,	Coordinates of the integration points in local coordinate system.
64, 66:	
69:	Increment the integration point counter.
71:	Calculate the shape functions and their derivative values.

72:	Calculate the Cartesian derivatives of shape functions.
73:	Form the strain matrix.
76–80:	Calculate the area/volume of element.
85–89:	Calculate the value of concentration at the current integration point, Eq. 6.86.
92–100:	Calculate the composite elasticity matrix, Eq. 6.81.
104:	Multiply the elasticity matrix and strain matrix.
106–116:	Form the element stiffness matrix, Eq. 6.46.
119–148:	Form the rhs, internal load, vector.
124–126:	Calculate the eigenstrain values at the current integration point.
130–135:	Calculate the stress components resulting from the eigenstrains.
137–148:	Form the element rhs, internal load, vector.
157–176:	Assemble element rhs, internal load vector and elements stiffness matrix into the global rhs, internal load vector and global stiffness matrix.

Function**stiffness3.m**

This function forms the global stiffness matrix and global rhs, load, vector resulting from internal stress field for the solution of nodal displacements. It is optimized for Matlab/Octave.

The function makes calls to the following functions:

- **modps.m**
- **sfr2.m**
- **bmats2.m**
- **dbe2m**

Variable and array list:

npoint:	Total number of nodes.
nelem:	Total number of elements.
nnode:	Number of nodes per element.
nstre:	Number of stress components.
ndime:	Number of Cartesian coordinate dimension.
ndofn:	Number of DOF per node.
ngaus:	Order of numerical integration.
ntype:	Solution type, ntype = 1 for plane-stress, ntype = 2 for plane-strain.
posgp(ngaus):	Position of integration points.

(continued)

weigp(ngaus):	Weights of integration points.
ncomp(npoin):	Nodal values of composition.
matno(nelem):	Material number of elements.
eigen(nstres):	Eigenstrain components.
iload(ntotv):	Global rhs, load vector resulting from internal stress field, ntotv = npoin × ndofn.
props(nmats, nprop):	Material properties, nmats is number of materials and nprop is number of properties.
coord(npoin, ndime):	Cartesian coordinates of nodes.
lnods(nelem, nnode):	Nodal connectivity list of elements.
gstif(ntotv,ntotv):	Global stiffness matrix.
dvolum(nelem, mgaus):	Precalculated element area/volume at their integration points.
dgdx(nelem, mgaus,nstre, nnode):	Precalculated Cartesian derivatives of shape functions of elements at their integration points.

Listing:

```

1 function [gstif,iload] =stiffness3
2 (npoin,nelem,nnode,nstre,ndime,
3 ndofn, ...
4 ngaus,ntype,lnods,matno,
5 coord,props, ...
6 nvfix,nofix,iffix,fixed,
7 posgp,weigp, ...
8 dgdx,dvolum,ncomp,eigen)
9 format long;
10 %--- order of integration
11
12 ngaus2=ngaus;
13 if(nnode == 3)
14 ngaus2=1;
15 end
16 mgaus = ngaus*ngaus2;
17
18 %--initialize global and local
19 % stiffness and rhs vectors
20 ntotv = npoin*ndofn;
21 nevab = nnode*ndofn;
22 gstif = sparse(ntotv,ntotv);
23 iload = sparse(ntotv,1);
24 estif = zeros(nelem,nevab,nevab);
25
26 %--- elemental values
27
28 for inode =1:nnode
29 lnode = lnods( :,inode);
30 cv( :,inode) = ncomp(lnode);
31 end
32
33 %--- integrate element stiffness
34
35 kgasp=0;
36 for igaus=1:ngaus
37 exisp=posgp(igaus);
38 for jgaus=1:ngaus2
39 etasp =posgp(jgaus);
40 f(nnode ==3)
41 etasp=posgp(ngaus+igaus);
42 end
43
44 kgasp=kgasp+1;
45 [shape,deriv]=sfr2(exisp,etasp,
46 nnode);
47
48 %--- composition at the integration
49 % points and elasticity matrix:
50
51 cvgp=zeros(nelem,1);
52
53 for inode=1:nnode

```

```

68 cvgp = cvgp +cv( :,inode)*shape
69 (inode);
70 end
71 %--- calculate composite elasticity
72 matrix:
73 dmatx=zeros(nelem,3,3);
74
75 for istre=1:nstre
76 for jstre=1:nstre
77
78 dmatx( :,istre,jstre) = dmatxm
79 (istre,jstre) + cvgp *(dmatxp
80 (istre,jstre) - ...
81 dmatxm(istre,jstre));
82 end
83 end
84
85 [bmatx]=bmats2(dgdx,nelem,nnode,
86 nstre,nevab,kgasp);
87 [dbmat] =dbe2(nelem,nevab,nstre,
88 bmatx,dmatx);
89
90 %Form element stiffness:
91 for ievab=1:nevab
92 for jevab=1:nevab
93 for istre=1:nstre
94 estif( :,ievab,jevab)=estif( :,
95 ievab,jevab)+bmatx( :,istre,
96 ievab).* ...
97 dbmat( :,istre,jevab).*dvolum
98 ( :,kgasp);
99 end
100 end
101 %---- load vector due to
102 eigenstress:
103
104 % extraplot egen strains from
105 nodes to gauss points:
106 for istre=1:nstre
107 istran( :,istre) = -eigen(istre)
108 *cvgp;
109 end
110 %-- internal stress:
111 for istre =1:nstre
112 for jstre =1:nstre
113 istres( :,istre) = istres( :,istre)
114 + dmatx( :,istre,jstre).*istran( :,
115 jstre);
116 end
117 end
118 for inode = 1:nnode
119 ievab =(inode-1)*ndofn+1;
120 jevab =(inode-1)*ndofn+2;
121
122 eload( :,ievab)=eload( :,ievab)
123 -(dgdx( :,kgasp,1,inode).*
124 *istres( :,1) ...
125 +(dgdx( :,kgasp,2,inode).*
126 *istres( :,3)).*dvolum( :,
127 kgasp);
128 eload( :,jevab)=eload( :,jevab)
129 -(dgdx( :,kgasp,1,inode).*
130 *istres( :,3) ...
131 +(dgdx( :,kgasp,2,inode).*
132 *istres( :,2)).*dvolum
133 ( :,kgasp);
134
135 %--- assemble element stiffness and
136 load vectors into
137
138 %--- global stiffness matrix and
139 for inode =1:nnode
140 lnode = lnods( :,inode);
141 for idofn =1:ndofn
142 ievab =(inode-1)*ndofn+idofn;
143 itotv =(lnode-1)*ndofn+idofn;

```

```

144
145 iload = iload + sparse(itotv,1,
   eload( :,ievab),ntotv,1);
146
147 for jnode =1:nnode
148 knode = lnods( :,jnode);
149 for jdofn =1:ndofn
150 jevab =(jnode-1)*ndofn+jdofn;
151 jtotv =(knode-1)*ndofn+jdofn;
152
153 gstif = gstif +sparse(itotv,jtotv,
   estif( :,ievab,jevab),ntotv,
   ntotv);
154
155 end
156 end
157 end
158 end
159
160
161 end %endfunction

```

Line numbers:

8–11:	Order of numerical integration, depending on the element type.
12:	Total integration points per element.
16:	Total number of variables.
17:	Total number of variables per element.
19:	Initialize global stiffness matrix.
20:	Initialize global rhs, load, vector.
30–33:	Calculate the elasticity matrix of the matrix phase.
34–37:	Calculate the elasticity matrix of the second phase.
40:	Initialize array for nodal compositions for all elements.
44–47:	Get nodal composition values for all elements.
40–132:	Integrate the element stiffness matrix and rhs, load, vector of all elements.
53, 55, 57:	Coordinates of integration points in local coordinate system.
60:	Increment integration point counter.
65–69:	Calculate the composition values at the integration point. Eq. 6.86.
71–81:	Calculate the composite elasticity matrix, Eq. 6.81.
85:	Calculate the strain matrix for all elements.

86:	Multiply the strain matrix with elasticity matrix for all elements.
88–98:	Form the stiffness matrix for all elements, Eq. 6.46.
101–127:	Calculate the rhs, internal load, vector resulting from the internal stress field for all elements.
106–108:	Calculate the eigenstrain values at the current integration point.
112–116:	Calculate the stress components resulting from the eigenstrains.
118–127:	Form the element rhs, internal load, vector for all elements.
135–159:	Assemble element stiffness matrices and rhs, internal load vectors into the global stiffness matrix and global rhs, load, vector.

Function**stress2.m**

This function calculates the stress and strain fields from the nodal displacements and the derivatives of elastic strain energy at the elements integration points. It is in longhand format and is not optimized for Matlab/Octave.

For compact coding of the derivatives of elastic-strain energy with respect to concentration, three different stress values are evaluated at the element integration points. These are:

$$\begin{aligned}\sigma &= (C_M + c(C_p - C_M))(\epsilon - \epsilon^0(c)) \\ \bar{\sigma} &= (C_p - C_M)(\epsilon - \epsilon^0(c)) \\ \sigma^0 &= (C_M + c(C_p - C_M))\epsilon^0\end{aligned}\quad (6.94)$$

in which c is the concentration, C_M is the elasticity matrix of the matrix phase, C_p is the elasticity matrix of the second phase, $\epsilon^0(c)$ is the composition dependent eigenstrain values, and ϵ^0 is the intrinsic values of the eigenstrains resulting from the lattice mismatch.

The function makes calls to the following functions:

- **modps.m**
- **sfr2.m**
- **jacob2.m**

- **bmats.m**

Variable and array list:

nelem:	Total number of elements in the solution.
npoint:	Total number of nodes in the solution.
nnode:	Number of nodes per element.
ngaus:	Order of numerical integration.
nstre:	Number of stress components.
ndofn:	Number of DOFs per node.
ntype:	Solution type, ntype = 1 for plane-stress, ntype = 2 for plane-strain.
ndime:	Number of Cartesian coordinate dimension.
matno(nelem):	Material number of elements.
posgp(ngaus):	Position of integration points.
weigp(ngaus):	Weights for numerical integration points.
asdis(ntotv):	Nodal displacements, ntotv = npoin \times ndofn.
ncomp(npoin):	Nodal values of composition.
eigen(nstre):	Eigenstrain components.
props(nmats, nprop):	Material properties, nmats is number of materials and nprop is number of properties.
coord(npoin, ndime):	Cartesian coordinates of nodes.
lnods(nelem, nnode):	Nodal connectivity list of elements.
dsendc(nelem, mgaus):	First derivative of elastic energy at element integration points.
dsen2dc(nelem, mgaus):	Second derivative of elastic strain energy at element integration points.

Listing:

```

1 function [dsendc,dse2dc] = stress2
2 (asdis,nelem,npoin,nnode,ngaus,
3 nstre,props, ...
4 ntype,ndofn,ndime,lnods,matno,
5 coord,posgp, ...
6 weigp,ncomp,eigen)
7 format long;
8
9 %--- order of integration
10 if(nnode == 3)
11 ngaus2=1;
12 end
13
14 %--- element node coordinates.
15
16 for inode=1:nnode
17 lnode = lnods(ielem,inode);
18 for idime=1:ndime
19 elcod(idime,inode)=coord(lnode,
20 idime);
21 end
22 end
23
24 %---composition and displacements:
25
26 for inode =1:nnode
27 lnode = lnods(ielem,inode);
28 cv(inode) = ncomp(lnode);
29 for idofn=1:ndofn
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57

```

```

58 ievab=(inode-1)*ndofn+idofn;      98 for jstre=1:nstre
59 nposn=(lnods(ielem,inode)-1)      99
60 *ndofn+idofn;
61 eldis(ievab)=asdis(nposn);
62 end
63
64 %--- calculate strains and stresses
65 at integration points:
66 kgasp=0;
67 for igaus=1:ngaus
68 for jgaus=1:ngaus2
69
70 kgasp=kgasp+1;
71 exisp=posgp(igaus);
72 etasp=posgp(jgaus);
73
74 [shape,deriv]=sfr2(exisp,etasp,
75 nnodes);
75 [cartd,djacb,gpcod]=jacob2(ielem,
76 elcod,kgasp,shape,deriv,nnodes,
77 ndime);
76 [bmatx]=bmats(cartd,shape,inode);
77
78 %-- initialize stress & strain
79 components:
80 for istre=1:nstre
81 stran(istre)=0.0;
82 stres(istre)=0.0;
83 stresb(istre)=0.0;
84 stres0(istre)=0.0;
85 end
86
87 %--- composition at the integration
88 points and elasticity matrix:
89 cvgp=0.0;
90
91 for inode=1:nnodes
92 cvgp=cvgp+cv(inode)*shape
93 (inode);
94 end
95 %--- calculate composite elasticity
96 matrix:
97 for istre=1:nstre
98 for jstre=1:nstre
99
100 dmatx(istre,jstre)=dmatxm(istre,
101 jstre)+cvgp*(dmatxp(istre,jstre)
102 -...
103 dmatxm(istre,jstre));
104
105 %--- calculate the strains:
106
107 for istre=1:nstre
108 for ievab=1:nevab
109
110 stran(istre)=stran(istre)+bmatx
111 (istre,ievab)*eldis(ievab);
112 end
113 end
114
115 %calculate stresses:
116
117 for istre=1:nstre
118 for jstre=1:nstre
119
120 stres(istre)=stres(istre)+dmatx
121 (istre,jstre).*...
122 (stran(jstre)-cvgp*eigen
123 (jstre));
124 end
125
126 %--- calculate stressb
127
128 for istre=1:nstre
129 for jstre=1:nstre
130
131 stresb(istre)=stresb(istre)++
132 dmatx_diff(istre,jstre)*(stran
133 (jstre)-...
134 cvgp*eigen(jstre));
135 end
136
137 %-- calculate stres0
138
139 for istre=1:nstre

```

```

140 for jstre=1:nstre
141 stres0(istre) = stres0(istre) +
dmatx(istre,jstre)*eigen(jstre);
142 end
143 end
144
145
146 %--- derivatives of strain energy:
147
148 for istre = 1:nstre
149
150 dsendc(ielem,kgasp) = dsendc
(ielem,kgasp) + 0.5*(stran(istre)
-cvgp*eigen(istre))*stresb(istre)
- eigen(istre)*stres(istre);
151
152 dsen2dc (ielem,kgasp) = dsen2dc
(ielem,kgasp) + eigen(istre)*
(stres0(istre) -2.0*stresb
(istre));
153
154 end
155
156 end %igaus
157 end %jgaus
158
159 end %ielem
160
161
162 end %endfunction
163

```

Line numbers:

9–12:	Order of numerical integration, depending on the element type.
14:	Total number of integration points per element.
15:	Number of variables per element.
20–23:	Calculate the elasticity matrix of matrix phase.
24–27:	Calculate the elasticity matrix of second phase.
28–35:	Calculate the differential elasticity matrix, Eq. 6.94.
36–159:	Loop over elements.
38–41:	Initialize derivatives of elastic strain energy.
45–50:	Calculate Cartesian coordinates of element nodes.
54–62:	Calculate the elemental nodal values of displacements and concentration.
67–157:	Loop over integration points.
70:	Increment integration point counter.

71–72:	Coordinates of integration points in local coordinates.
74:	Calculate shape functions and their derivative values.
75:	Calculate Cartesian derivatives of shape functions.
76:	Calculate strain matrix.
80–85:	Initialize strain and stress components.
89–93:	Calculate the value of the concentration at the integration point, Eq. 6.86.
97–103:	Calculate composite elasticity matrix, Eq. 6.94.
105–113:	Calculate strains resulting from the displacements.
115–143:	Calculate stress terms given in Eq. 6.94.
146–154:	Calculate derivatives of elastic strain energy with respect to composition.
150:	First derivative of elastic-strain energy.
152:	Second derivative of elastic-strain energy.

Function**stress3.m**

This function calculates the stress and strain fields from the nodal displacements and the derivatives of elastic strain energy at the elements integration points. It is optimized for Matlab/Octave. The function utilizes Eq. 6.94 to evaluate the stress components.

The function makes calls to the following functions:

- **modps.m**
- **sfr2.m**
- **bmats2.m**

Variable and array list:

nelem:	Total number of elements in the solution.
npoin:	Total number of nodes in the solution.
nnode:	Number of nodes per element.
ngaus:	Order of numerical integration.
nstre:	Number of stress components.
ndofn:	Number of DOFs per node.
ntype:	Solution type, ntype = 1 for plane-stress, ntype = 2 for plane-strain.
ndime:	Number of Cartesian coordinate dimension.

(continued)

matno(nelem):	Material number of elements.
posgp(ngaus):	Position of integration points.
weigp(ngaus):	Weights for numerical integration points.
asdis(ntotv):	Nodal displacements, ntotv = npoin \times ndofn.
ncomp(npoin):	Nodal values of composition.
eigen(nstre):	Eigenstrain components.
props(nmats, nprop):	Material properties, nmats is number of materials and nprop is number of properties.
coord(npoin, ndime):	Cartesian coordinates of nodes.
lnods(nelem, nnodes):	Nodal connectivity list of elements.
dsendc(nelem, mgaus):	First derivative of elastic energy at element integration points.
dsen2dc(nelem, mgaus):	Second derivative of elastic strain energy at element integration points.
dvolum(nelem, mgaus):	Precalculated element area/volume at their integration points.
dgdx(nelem, mgaus,nstre, nnodes):	Precalculated Cartesian derivatives of shape functions of elements at their integration points.

Listing:

```

1 function [dsendc,dSEN2dc] = stress3
2 %----- input parameters -----
3 %----- order of integration
4 %----- calculate strains and stresses
5 %----- output parameters -----
6 %----- calculate strains and stresses
7 %----- calculate strains and stresses
8 %----- calculate strains and stresses
9 %----- calculate strains and stresses
10 %----- calculate strains and stresses
11 %----- calculate strains and stresses
12 %----- calculate strains and stresses
13 %----- calculate strains and stresses
14 %----- calculate strains and stresses
15 %----- calculate strains and stresses
16 %----- calculate strains and stresses
17 %----- calculate strains and stresses
18 %----- calculate strains and stresses
19 %----- calculate strains and stresses
20 %----- calculate strains and stresses
21 %----- calculate strains and stresses

```

```

66
67 [bmatx]=bmats2(dgdx,nelem,nnode,
  nstre,nevab,kgasp);
68
69
70 %--- composition at the integration
  points and elasticity matrix:
71
72 cvgp=zeros(nelem,1);
73
74 for inode=1:nnode
  cvgp=cvgp+cv(:,inode)*shape
  (inode);
75 end
76
77 %--- calculate composite elasticity
  matrix:
78
79 dmatx=zeros(nelem,3,3);
80
81 for istre=1:nstre
  for jstre=1:nstre
82
83 dmatx(:,istre,jstre)=dmatxm
  (istre,jstre)+cvgp*(dmatxp
  (istre,jstre)-...
84
85 dmatxm(istre,jstre));
86
87 end
88
89 end
90
91 %-- initialize stress &strain
  components:
92
93 stran=zeros(nelem,nstre);
94 stres=zeros(nelem,nstre);
95 stresb=zeros(nelem,nstre);
96 stres0=zeros(nelem,nstre);
97
98
99 %--- calculate the strains:
100
101 for istre=1:nstre
  for ievab=1:nevab
102
103 stran(:,istre)=stran(:,istre)
  +bmatx(:,istre,ievab).*eldis
  (:,ievab);
104
105
106 end
107 end
108
109 %calculate stresses:
110
111 for istre=1:nstre
  for jstre=1:nstre
112
113 stres(:,istre)=stres(:,istre) +
  dmatx(:,istre,jstre).*...
114
115 (stran(:,jstre)-cvgp*eigen
  (jstre));
116
117 end
118 end
119
120 %--- calculate stressb
121
122 for istre=1:nstre
  for jstre=1:nstre
123
124 stresb(:,istre)=stresb(:,istre) +
  dmatx_diff(istre,jstre)*(stran
  (:,jstre)-...
125
126 cvgp*eigen(jstre));
127
128 end
129 end
130
131 %-- calculate stres0
132
133 for istre=1:nstre
  for jstre=1:nstre
134
135 stres0(:,istre)=stres0(:,istre) +
  dmatx(:,istre,jstre)*eigen
  (jstre);
136
137 end
138
139
140 %--- derivatives of strain energy:
141
142 for istre=1:nstre
143
144 dsenc(:,kgasp)=dsenc(:,kgasp) +
  0.5*(stran(:,istre)-cvgp*eigen
  (istre)).*stresb(:,istre)-eigen
  (istre)*stress(:,istre);

```

```

145
146 dsen2dc (:,kgasp) = dsen2dc
147   (:,kgasp) + eigen(istre)*(stres0
148     (:,istre) -2.0*stresb (:,istre));
149 end
150
151
152 end %igaus
153 end %jgaus
154
155
156
157 end %endfunction

```

Line numbers:

9–12:	Order of numerical integration.
14:	Total number of integration points per element.
15:	Number of variables per element.
17–18:	Initialize derivatives of free energy for all elements.
20:	Initialize composite elasticity matrix for all elements.
24–26:	Calculate elasticity matrix for matrix phase.
28–31:	Calculate elasticity matrix for second phase.
32–34:	Calculate differential elasticity matrix, Eq. 6.94.
38–51:	Calculate nodal displacements and concentration values for all elements.
55–153:	Loop over integration points.
59:	Increment integration point counter.
60–61:	Coordinates of integration points in local coordinate system.
63:	Calculate the shape functions and their derivative values.
72–76:	Calculate the concentration value at the integration point for all elements, Eq. 6.86.
82–89:	Calculate the composite elasticity matrix for all elements, Eq. 6.94.
91–97:	Initialize strain and stress components for all elements.
101–107:	Calculate the strain values for all elements.
109–138:	Calculate stress components given in Eq. 6.84 for all elements.
140–149:	Calculate the derivatives of elastic-strain energy for all elements.
144:	First derivative of the elastic energy.
146:	Second derivative of the elastic energy.

Function**init_micro_thin_film.m**

This function generates either single layer or double layer thin film microstructure, depending on the value of `icase`. `icase = 1` is for single layer and `icase = 2` for double layers. The function is specifically designed for mesh input file, `mesh_64_4.inp`. For different mesh configurations, it should be modified.

Variable and array list:

npoint:	Number of nodes.
icase:	<code>icase = 1</code> for single layer, <code>icase = 2</code> double layer thin film morphology.
nodco(npoint):	Nodal concentration values.
coord(npoin, ndime):	Nodal Cartesian coordinates.

Listing:

```

1 function [nodco] = init_micro_
thin_film(npoin,coord,icase)
2
3
4 if (icase == 1)
5
6 for ipoin=1:npoin
7
8 nodco(ipoin) = 0.001+0.001*
(0.5-rand);
9 nodco(ipoin + npoin) =0.0;
10
11 if(coord(ipoin,2) >= 0.23 && coord
(ipoin,2) <= 0.29)
12
13 nodco(ipoin) = 0.999+0.001*
(0.5-rand);
14 end
15
16 end
17
18 end
19
20
21 if(icase == 2)
22

```

```

23 for ipoin=1:npoint
24
25 nodco(ipoin) = 0.001+0.001*
(0.5-rand);
26 nodco(ipoin + npoin) = 0.0;
27
28 if(coord(ipoin,2) >= 0.1 && coord
(ipoin,2) <= 0.17185)
29
30 nodco(ipoin) = 0.999+0.001*
(0.5-rand);
31 end
32
33 if(coord(ipoin,2) >= 0.3 && coord
(ipoin,2) < 0.375)
34
35 nodco(ipoin) = 0.999+0.001*
(0.5-rand);
36 end
37
38 end
39
40 end %if
41
42 end %end function

```

Line numbers:

4–18:	If icase = 1, introduce a single layer film into the FEM mesh and modulate the concentration values of the nodes with noise term of which value is 0.001.
21–40:	If icase = 2, introduce double layer film into the FEM mesh and modulate the concentration values of the nodes with noise term of which value is 0.001.

5. Jagannadham K, Narayan J (1991) Critical thickness during two and three dimensional epitaxial growth in semiconductor heterostructures. Mater Sci Eng: B 8:107
6. Zhou J, Cockayne DJH (1996) Nucleation of semi-circular misfit dislocation loops from the epitaxial surface of strained layer heterostructures. J Appl Phys 79:7632
7. Dong L, Schnitker J, Smith RW, Srolovitz DJ (1998) Stress relaxation and misfit dislocation nucleation in the growth of misfitting films: a molecular dynamics study. J Appl Phys 83:217
8. Lam CH, Lee CK, Sander LM (2002) Competing roughening mechanisms in strained heteroepitaxy: a fast kinetic Monte Carlo study. Phys Rev Lett 89:216102
9. Kassner K, Misbach C (1999) A phase-field approach for stress-induced instabilities. Europhys Lett 46:217
10. Kassner K, Misbach C, Muller J, Kappey J, Kohlert P (2001) Phase-field modeling of stress induced instabilities. Phys Rev E 63:036117
11. Wang YU, Jin YM, Khachaturyan AG (2004) Phase field microelasticity modeling of surface instability of heteroepitaxial thin films. Acta Mater 52:81
12. Seol DJ, Hu SY, Liu ZK, Chen LQ, Kim SG, Oh KH (2005) Phase-field modeling of stress-induced surface instabilities in heteroepitaxial thin films. J Appl Phys 98:044910
13. Chiranjeevi BG, Abinandanan TA, Gururajan MP (2009) A phase-field study of morphological instabilities in multilayer thin films. Acta Mater 57:1060
14. Zhang YW (2000) Self-organization, shape transition and stability of epitaxially strained islands. Phys Rev B 61:10338
15. Zhang YW, Bower AF (2001) Three-dimensional analysis of shape transitions in strained-heteroepitaxial islands. Appl Phys Lett 78:2706

References

1. Asaro RJ, Tiller WA (1972) Interface development during stress corrosion cracking. Metall Trans A 3:1789
2. Grinfeld MA (1986) Instability of interface between non-hydrostatically stressed elasticity body and melts. Sov Phys Doklady 290:1358
3. Srolovitz DJ (1989) On the stability of surfaces of stressed solids. Acta Metall 37:621
4. Kamat SV, Hirth JP (1990) Dislocation injection in strained multilayer structures. J Appl Phys 67:6844

6.9 Case Study-XIV

Phase-field modeling of multi-variant martensitic transformations

Objective:

The objective of this case study is to demonstrate a FEM phase-field algorithm in which phase-field equations are fully coupled and solved simultaneously. Although, in this case, the fully coupled equations are elasticity and non-conserved order parameters; however, the approach serves as a template for fully coupled formulism of other phase-field models.

6.9.1 Background

Martensite, named after German metallurgist Adolf Marten (1850–1914), most commonly refers to formation of very hard body-centered tetragonal, *bct*, phase with fast cooling from the temperatures where the austenite phase with face-centered cubic, *fcc*, crystal structure is stable in Fe–C system [1, 2]. The martensitic transformations are diffusionless, first-order, solid-to-solid phase transformations. Materials undergoing this phase transformation have many important technological implications. Crystals undergoing a thermo-elastic martensitic phase transformation often exhibit shape memory effects and they are widely used in sensor-actuator applications [3]. Martensitic transformations are also observed in biological systems, the cylindrical protein crystals are found to perform life functions of primitive cells with these transformations [4].

Owing to its technological importance, the martensitic phase transformations are studied at the atomistic level [5–8] and extensively with phase-field models [9–15]. A review on phase-field modeling of martensitic transformation can be found in [16].

6.9.2 Phase-Field Model

Since the martensitic transformations are diffusionless, the stress-strain fields and the martensitic phases evolve with the same rate. Therefore, the evolution equations cannot be separated, contrary to *Case Study-XIII*, and they need to be solved with fully coupled mode. The phase-field formulism for this case study follows the work of Schmitt et al. [14]. The formulism given below is for two martensitic invariants as a general case in 2D. In the model, martensitic phases are described by non-conserved order parameters. The order parameters, c_i , take the value of one for the i th martensite orientation variant, and the value of zero in the austenitic matrix and in the other orientation variant. The total potential energy, ψ , is additively composed of three components; the elastic energy, W , the

phase separation energy, ψ^{Sep} , and the gradient energy, ψ^{Grad} , [17]. Thus,

$$\psi(\varepsilon, c_i, \nabla c_i) = W(\varepsilon, c_i) + \psi^{Sep}(c_i) + \psi^{Grad}(\nabla c_i) \quad (6.95)$$

The elastic energy W is defined as:

$$W(\varepsilon, c_i) = \frac{1}{2} [\varepsilon - \varepsilon^0(c_i)] : C(c_i) [\varepsilon - \varepsilon(c_i)] \quad (6.96)$$

in which the elasticity matrix, $C(c_i)$, is assumed to be linearly varying between the austenitic matrix and the martensitic phases as

$$C(c_i) = C_A + c_1(C_M - C_A) + c_2(C_M - C_A) \quad (6.97)$$

where indices A and M represent the elasticity matrices of austenitic and martensitic phases, respectively.

The linearized strains ε is related to the displacement field u by

$$\varepsilon(u) = \frac{1}{2} (\nabla u + (\nabla u)^T) \quad (6.98)$$

and the transformation-induced eigenstrains, $\varepsilon^0(c_i)$, is defined as:

$$\varepsilon^0(c_i) = c_1 \varepsilon_1^0 + c_2 \varepsilon_2^0 \quad (6.99)$$

where ε_1^0 and ε_2^0 are the eigenstrains associated with each martensite variants. Neglecting inertia effects and in the absence of volumetric forces, the equilibrium condition requires:

$$\operatorname{div} \sigma = 0 \quad (6.100)$$

and the stresses, σ , are determined by the constitutive relationship as

$$\sigma = \frac{\partial \psi}{\partial \varepsilon} = C(c_i)(\varepsilon - \varepsilon^0(c_i)) \quad (6.101)$$

The gradient and phase separation energies are defined as:

$$\psi^{Grad}(\nabla c_i) = \frac{1}{2} \kappa_g GL \sum_i^2 |\nabla c_i|^2 \quad (6.102)$$

and

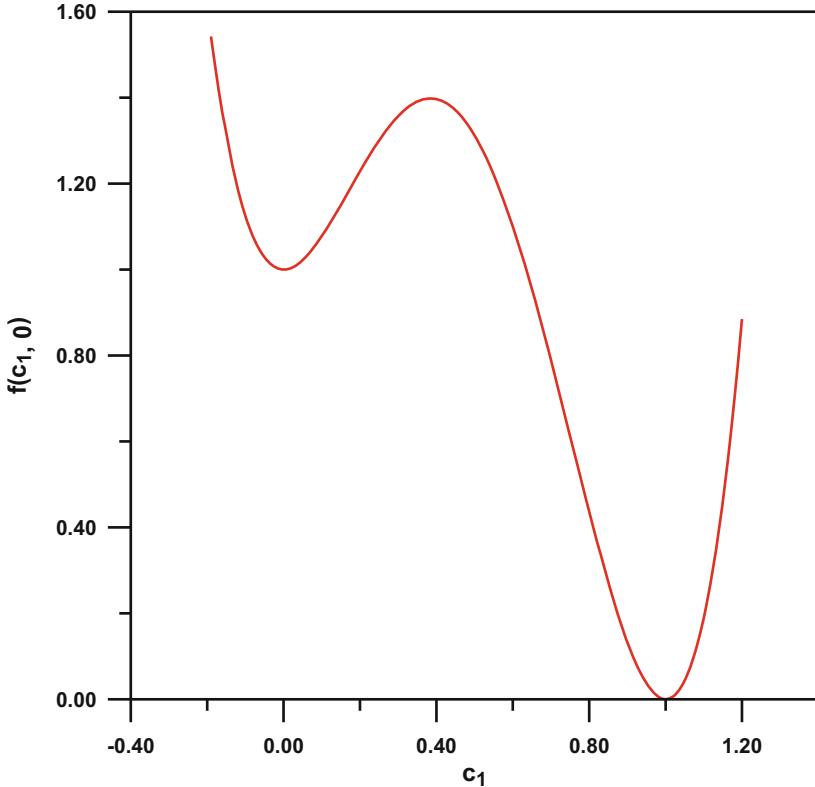


Fig. 6.12 Values of $f(c_1, 0)$, Eq. 6.104 with coefficients $\alpha = 20$, $\beta = 3\alpha + 12$, and $\gamma = 2\alpha + 12$ for single martensite invariant

$$\psi^{Sep}(c_i) = \kappa_s \frac{G}{L} f(c_i) \quad (6.103)$$

where κ_s and κ_g are the positive coefficients, the G appearing in both equations can be related to the characteristics interface energy and L is the parameter that controls the width of the interface region. The phase separation potential, $f(c_i)$, is formulated using the Landau polynomial expansion as [18]

$$f(c_i) = 1 + \frac{\alpha}{2}(c_1^2 + c_2^2) - \frac{\beta}{3}(c_1^3 + c_2^3) + \frac{\gamma}{4}(c_1^2 + c_2^2)^2 \quad (6.104)$$

The plot of Eq. 6.104 for single martensite invariant is given in Fig. 6.12. The function has a value of 1 at $c_1 = 0$ which corresponds to metastable austenite phase and has value of 0 at $c_1 = 1$ which corresponds to stable martensite phase. In order to system transform from metastable austenite phase, it has to overcome

the energy barrier seen in the figure. The dimensionless parameters α , β , and γ in Eq. 6.104 set the height of this energy barrier.

The evolution of order parameters is assumed to be proportional to the variational derivative of the total potential energy, which is defined as:

$$\frac{\partial c_i}{\partial t} = -M \frac{\delta \psi}{\delta c_i} = -M \left[\frac{\partial W}{\partial c_i} + G \left(\frac{\kappa_s}{L} \frac{\partial f}{\partial c_i} - \kappa_g L \nabla c_i \right) \right] \quad (6.105)$$

which is time-dependent Ginzburg–Landau or Allen–Cahn equation and M is the mobility coefficient.

Because of dependence of $C(c_i)$ and $e^0(c_i)$ to phase-field variables, c_i , in Eqs. 6.96 through 6.101, they are fully coupled set of time-dependent nonlinear equations which will be solved with finite element method below.

6.9.3 FEM Formulation

The weak form of Eqs. 6.100 and 6.105 for two martensite invariants ($i = 1, 2$) with test functions η_u and η_{ci} are:

$$\int_V \nabla \eta_u \sigma dV = \int_V \eta_u t^* dA \quad (6.106)$$

and

$$\begin{aligned} & \int_V \eta_{ci} \frac{\dot{c}_i}{M} dV - \int_V \nabla \eta_{ci} q_i dV + \int_V \eta_{ci} \left(\frac{\partial W}{\partial c_i} + G \frac{\kappa_s}{L} \frac{\partial f}{\partial c_i} \right) dV \\ &= - \int_V \eta_{ci} q_i^* dA \end{aligned} \quad (6.107)$$

in which $q_i = -\kappa_g GL \nabla c_i$. The boundary conditions for stress is the tractions, $t^* = \sigma n$ and for q_i normal flux $q_i^* = q_i n$ with $q_i^* = 0$, where n is the outer normal vector to the boundary of domain V .

As before, by taking nodal variables as displacements, u , and the order parameters, c_i , and utilizing the element shape functions, u , c_i , ∇c_i , and ε are discretized at element level as:

$$u = \sum_i^n N_i u_i, \quad \varepsilon = \sum_i^n B_i^u u_i \quad (6.108)$$

and

$$c_i = \sum_j^n N_j c_{ij}, \quad \nabla c_i = \sum_j^n B_j^c c_{ij} \quad (6.109)$$

where n is the number of nodes of the element and N_i is the element shape functions corresponding to the nodes. The B_i^u is the usual strain matrix and B_i^c is the Cartesian derivative matrices, respectively, which they are expressed as:

$$B_i^u = \begin{bmatrix} \frac{\partial N_i}{\partial x} & 0 \\ 0 & \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial y} & \frac{\partial N_i}{\partial x} \end{bmatrix} \quad \text{and} \quad B_i^c = \begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \end{bmatrix} \quad (6.110)$$

By taking the nodal values as $\delta = (u_1, u_2, c_1, c_2)^T$ in which u_1 and u_2 are the displacement components in the Cartesian coordinate system and c_1 and c_2 are the order parameters for the martensite invariants, the nodal residuals at element level becomes:

$$R_i^e = \begin{bmatrix} \int_V (B_i^u)^T \sigma dV \\ \int_V N_i \frac{\dot{c}_i}{M} dV - \int_V (B_i^c)^T q_i dV + \int_V N_i \left(\frac{\partial W}{\partial c_1} + G \frac{\kappa_s}{L} \frac{\partial f}{\partial c_1} \right) dV \\ \int_V N_i \frac{\dot{c}_2}{M} dV - \int_V (B_i^c)^T q_2 dV + \int_V N_i \left(\frac{\partial W}{\partial c_2} + G \frac{\kappa_s}{L} \frac{\partial f}{\partial c_2} \right) dV \end{bmatrix} \quad (6.111)$$

The corresponding element stiffness,

$$K_{ij}^e = \frac{\partial R_i^e}{\partial \delta_j} = \begin{bmatrix} K_{ij}^{uu} & K_{ij}^{uc_1} & K_{ij}^{uc_2} \\ K_{ij}^{uc_1} & K_{ij}^{c_1 c_1} & K_{ij}^{c_1 c_2} \\ K_{ij}^{uc_2} & K_{ij}^{c_2 c_1} & K_{ij}^{c_2 c_2} \end{bmatrix} \quad (6.112)$$

by defining,

$$\tilde{\sigma} = (C_M - C_A)(\varepsilon - \varepsilon^0(c_i)) \quad \text{and} \quad \sigma_i^0 = C(c_i)\varepsilon_i^0 \quad (6.113)$$

the matrices in Eq. 6.140 are calculated as:

$$K_{ij}^{uu} = \int_V (B_i^u)^T C(c_i) (B_j^u) dV \quad (6.114)$$

$$K_{ij}^{uc_1} = \int_V (B_i^u)^T (\tilde{\sigma} - \sigma_1^0) N_j dV \quad (6.115)$$

$$K_{ij}^{c_1 c_1} = \int_V \left[N_i \left(\frac{1}{M\Delta t} \right) N_j + \kappa_g GL (B_i^u)^T B_j^u + N_i \left((\varepsilon_1^0)^T (\sigma_1^0 - 2\tilde{\sigma}) + \kappa_s \frac{G}{L} \frac{\partial^2 f}{\partial c_1^2} \right) N_j \right] dV \quad (6.117)$$

$$K_{ij}^{c_1 c_2} = \int_V \left[N_i \left(-\tilde{\sigma} (\varepsilon_1^0 + \varepsilon_j^0) + \varepsilon_j^0 \sigma_i^0 + \kappa_s \frac{G}{L} \frac{\partial^2 f}{\partial c_1 c_2} \right) N_j \right] dV \quad (6.118)$$

$$K_{ij}^{c_2 c_2} = \int_V \left[N_i \left(\frac{1}{M\Delta t} \right) N_j + \kappa_g GL (B_i^u)^T B_j^u + N_i \left((\varepsilon_2^0)^T (\sigma_2^0 - 2\tilde{\sigma}) + \kappa_s \frac{G}{L} \frac{\partial^2 f}{\partial c_2^2} \right) N_j \right] dV \quad (6.119)$$

6.9.4 Numerical Implementation

The code developed in this section only considers the single invariant in order to reduce the computational cost. It can be easily expended for the second invariant. In the case of single invariant, the element stiffness is formed as:

$$K_{ij}^e = \begin{bmatrix} K_{ij}^{uu} & K_{ij}^{uc_1} \\ K_{ij}^{uc_1} & K_{ij}^{c_1 c_1} \end{bmatrix} \quad (6.120)$$

and the last row in residual equation, Eq. 6.111 pertaining the second invariant is deleted.

After forming element stiffness and residuals, they are assembled into the global stiffness and right-hand side vectors, and the resulting set of nonlinear equation.

$$[K^G] \{d\delta\} = \{R^G\} \quad (6.121)$$

Note that the assembly process is carried out in such a away that the unknown vector $d\delta$ in Eq. 6.121 takes the form below. This is just a numerical convince for bookkeeping purposes.

$$d\delta = (u_1^1, u_2^1, u_1^2, u_2^2, \dots, u_1^N, u_2^N, c_1^1, c_1^2, \dots, c_1^N)^T \quad (6.122)$$

$$K_{ij}^{uc_2} = \int_V (B_i^u)^T (\tilde{\sigma} - \sigma_2^0) N_j dV \quad (6.116)$$

where N is the total number of nodes in the FEM mesh, u_1 and u_2 are the components of nodal displacements, and c_1 is the value of the order parameter for the first invariant at the nodes.

As in previous case studies, the set of nonlinear equation are again solved with the modified Newton–Raphson scheme. Thus, the steps in the algorithm are:

Step-1

For each time increment:

Form the global stiffness matrix, $[K^G]$, in Eq. 6.121 only once.

Step-2

Newton–Raphson iterations:

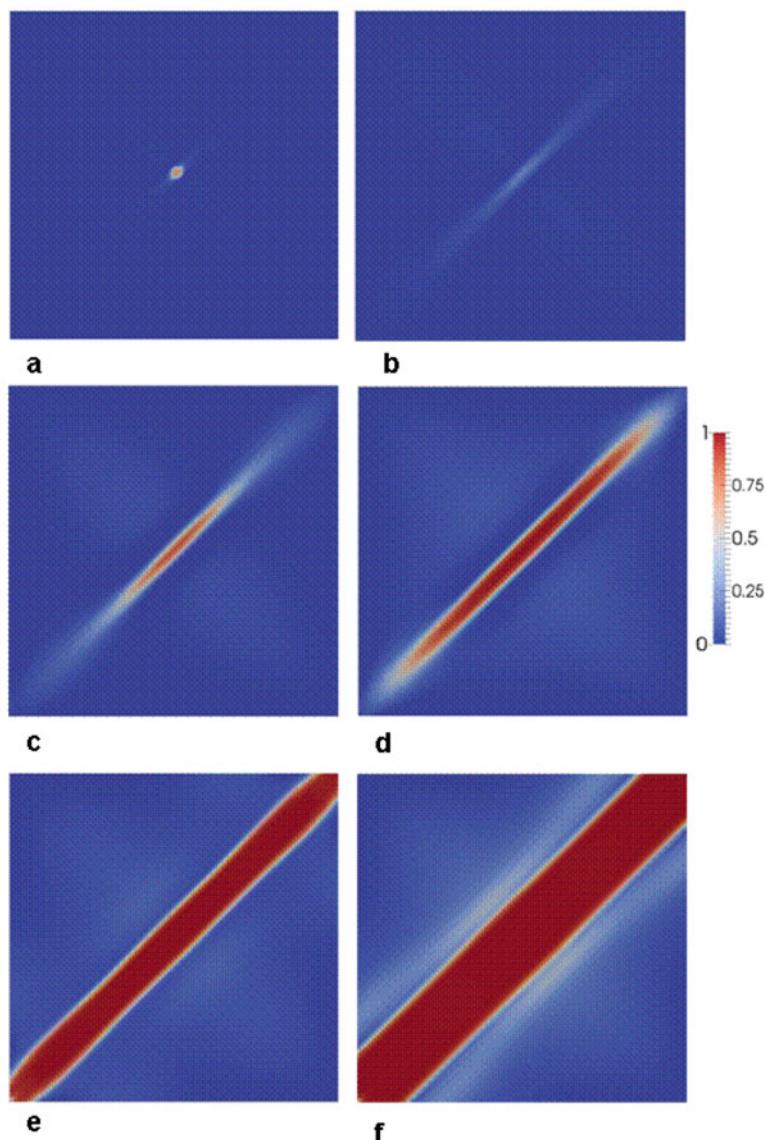
Form the right-hand side residual vector, $\{R^G\}$, in Eq. 6.121.

Solve Eq. 6.121.

Update $\delta^{n+1} = \delta^n + d\delta$, where n is the iteration number.

Check convergence, if it satisfied go to step-1, else repeat step-2.

Fig. 6.13 Formation of single martensite invariant from a nucleus. The nondimensional times in the figure are: (a) 0.2, (b) 4.0, (c) 10.0, (d) 12.0, (e) 15.0, and (f) 20.0



6.9.5 Results and Discussion

Two sets of simulations were carried out with the code developed in this section. The parameters used in the simulations are listed in function *material_parameters.m*, therefore it is not repeated in here. The simulation domain was a square plate in $1 \mu\text{m} \times 1 \mu\text{m}$ in dimension and discretized with tri-node isoparametric elements. There were 4225 nodes and 8192 elements in the FEM mesh and no periodicity is imposed. The evolution of single martensite invariant from a nucleus is explored in the first simulation. For

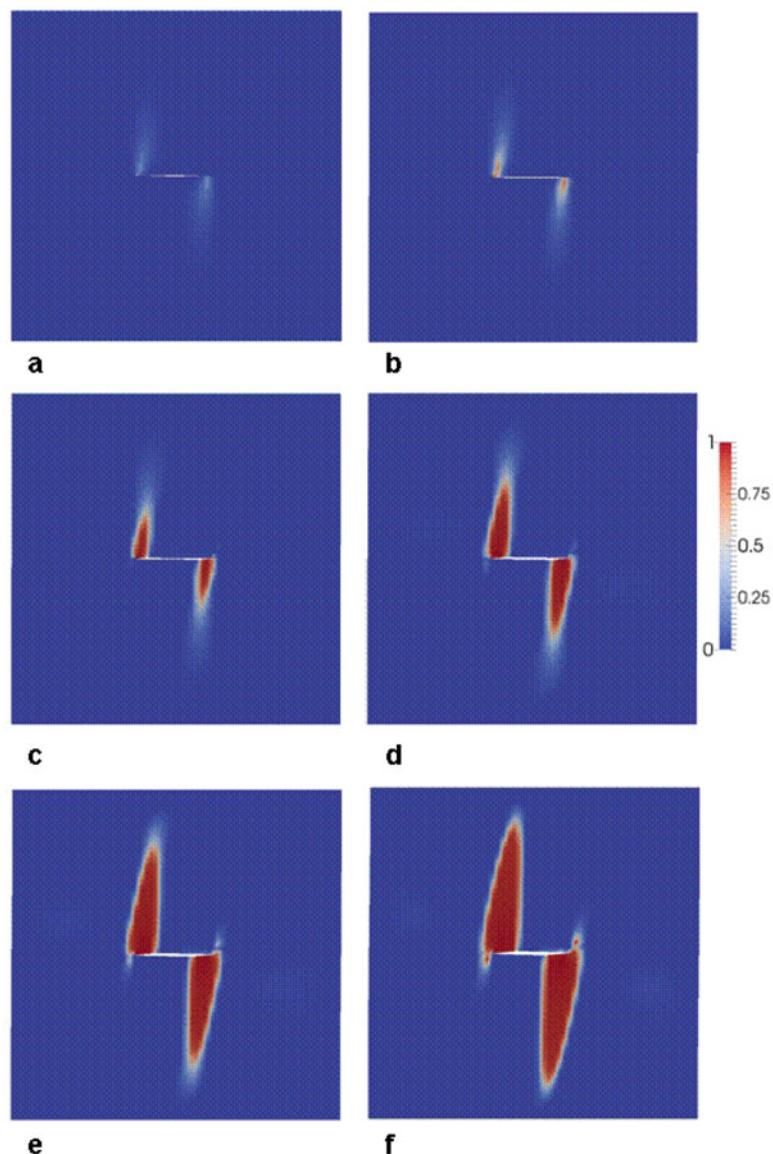
this purpose the nodal values, the order parameters, of two elements in the center of the mesh were set to be 0.9. Since the stable martensite phase has the value of 1 at the equilibrium, the nucleus was not in fully stable state. Also, there was no noise term to lead other possible nucleation in the system. The time evolution of the system is summarized in Fig. 6.13.

As can be seen from the figure and as described in [14], initially the nucleus starts shrinking and its order parameter values start to decrease towards to zero, in order to minimize the resulting total energy by minimizing the strain energy.

However, this results in an increase in the separation potential. When the system overcomes the energy barrier (Fig. 6.12), it becomes energetically favorable for the system to minimize the separation energy instead of strain energy. This leads to martensitic phase start to grow again. It is also noticeable thin needle-like growth of the martensite phase which agrees very favorably with the experiments. If simulations were carried out further than shown in here, the full domain transforms into the martensitic phase.

In the second simulation, the evolution of martensite phase at the crack tip is elucidated. In order to introduce a geometric crack into the FEM mesh, the elements in the center of the mesh were disconnected by introducing double nodes at the same Cartesian coordinates. Thus in this case, while the number of elements remained the same, there were 4242 nodes in the mesh. In addition, the nodes on the top and bottom edges have the prescribed displacements. While the axial displacements of the bottom nodes were

Fig. 6.14 The evolution of martensite phase ahead of the crack in a square plate under axial loading. The axial displacements, in μm , in the figure are:
 (a) 1.5×10^{-3} ,
 (b) 1.8×10^{-3} ,
 (c) 2.1×10^{-3} ,
 (d) 2.4×10^{-3} , (e)
 2.7×10^{-3} , and
 (f) 3.0×10^{-3}



constrained to be zero, the axial displacements of the nodes on the top edge were incremented $0.75 \times 10^{-6} \mu\text{m}$ per time step to provide the applied loading to the system. In addition, the eigenstrain of the martensite also contains the shear term for this case, as given in [15]. Again, there was no noise term to cause any nucleation in the domain. The resulting time evolution is shown in Fig. 6.14. The opening of the crack with increasing applied displacements is also apparent in the figure. As the applied loading increases, a stress concentration develops at the crack tip, resulting in an increase in the strain energy around this region. Owing to the continuous loading and no other relaxation mechanism is present in the system such as crack propagation, the system minimizes its total energy through the separation energy. As a result, martensite phase start to nucleate at the both ends of the crack and again they grow in needle-like shape with the identical rate in opposite direction to each other.

The mesh input files and the movies resulting from these simulations are given in subdirectory *case_study_14* in downloadable file.

6.9.6 Source Codes

Program

fem_mart_v1_2.m

This is the main program to solve Allen–Cahn equation with elastic inhomogeneities using FEM algorithm. The algorithm solves the evolution of the non-conserved order parameters and the evolution of stress–strain fields simultaneously in fully coupled mode.

Depending on the selection of the *isolve* parameter in the code: For *isolve* = 1, the code executes in longhand format and un-optimized mode. For *isolve* = 2, execution is for Matlab/Octave optimized mode. Therefore, for the desired execution mode, this parameter in line 37 should be modified in the program.

The program makes calls to the following functions:

- **input_fem_pf.m**
- **input_fem_elast.m**
- **gauss.m**
- **cart_deriv.m**
- **marten_stiff_v1.m**
- **marten_stiff_v2.m**
- **boundary_cond2_v1.m**
- **boundary_cond2_v2.m**
- **write_vtk_fem.m**

Listing:

```

1 %%%%%%%%%%%%%%%%
2 %
3 %      FEM PHASE-FIELD CODE FOR      %
4 %      MARTENSITIC TRANSFORMATIONS %
5 %%%%%%%%%%%%%%%%
6
7 % get initial wall time:
8
9 time0=clock();
10 format long;
11
12 global in;
13 global out;
14
15 icrack=0;
16
17 %--- Open input output files:
18 if(icrack == 0)
19 in = fopen('mesh64_3t.inp','r');
20 else
21 in = fopen('mesh64_3tcc.inp','r');
22 end
23
24 out = fopen('result_1.out','w');
25
26 %% Time integration parameters:
27
28 nstep = 2000;
29 nprnt = 40;
30 miter = 10;
31 dtime = 1.0e-2;
32 toler = 1.0e-2;
33
34 tfact = 0.0;
35 facto = 0.0000075;
36
37 isolve = 2;
```



```

117 [gstif,gforce,treac]           164 if(isolve == 2)
=boundary_cond2_v2(npoin,nvfix,      165 dummy =tdisp(ntotv2+1:ntotv);
nofix,iffix,fixed,ndofn,tfact,...  166 inrange = (dummy > 0.9999);
118     gstif,gforce,tdisp);        167 dummy(inrange) = 0.9999;
119 end                           168 inrange = ( dummy <= 5.0e-4 );
120                               169 dummy(inrange) = 5.0e-4;
121 end %if icrack               170 tdisp(ntotv2+1:ntotv) = dummy;
122                               171 end
123 %-----                   172
124 -- Solve & update variables: 173 %--- check norm for convergence:
125 %-----                   174
126                               175 dummy = gforce(ntotv2+1: ntotv);
127 asdis = gstif\gforce;          176
128                               177 normF = norm(dummy, 2);
129 %-- Update:                  178 if(iter > 1)
130                               179 if(abs(normF-oldnorm) <= toler)
131 for ipoin=1:npoin            180 break;
132 itotv = ntotv2 +ipoin;        181 end
133 asdis(itotv) = asdis(itotv)*dtime; 182 end
134 end                         183 oldnorm =normF;
135                               184
136 if(isolve == 1)              185 end %end of Newton
137                               186
138 for itotv=1:ntotv           187 %--- print results:
139 tdisp(itotv) =tdisp(itotv) +asdis 188
(itotv);                      189 if(mod(istep,nprnt) == 0 )
140 end                         190
141 end                         191 fprintf('done step: %5d\n',istep);
142 %                           192
143 if(isolve == 2)              193 %fname=sprintf('time_%d.out',
144                               194 istep);
145 tdisp = tdisp + asdis;       195 %out1=fopen(fname,'w');
146                               196 %for ipoin = 1:npoin
147 end                           197 %itotv =ntotv2 + ipoin;
148 %--- adjust small deviations: 198
149                               199 %fprintf(out1,'%14.6e %14.6e
150 if(isolve == 1)              200 %14.6e\n',coord(ipoin,1), coord
151                               201 (ipoin,2),tdisp(itotv));
152 for ipoin=1:npoin            202
153 itotv=ntotv2+ipoin;          203 %-- write_vtk
154 if(tdisp(itotv) > 0.9999)   204
155 tdisp(itotv) = 0.9999;       205 if(icrack == 1)
156 end                           206 for ipoin=1:npoin
157                               207 for idofn=1:ndofn2
158 if(tdisp(itotv) <= 5.0e-4)   208 itotv=(ipoin-1)*ndofn2+idofn;
159 tdisp(itotv) =5.0e-4;         209 cord2(ipoin,idofn) =coord(ipoin,
160 end                           210 idofn) + tdisp(itotv);
161 end                           211
162 end                           212
163 %

```

```

210 end
211 end
212 end
213
214
215 if(isolve == 1)
216 for ipoin=1:npoint
217 itotv=ntotv2+ipoin;
218 cont1(ipoin) =tdisp(itotv);
219 end
220
221 if(icrack == 1)
222 write_vtk_fem(npoint,nelem,nnode,
lnodes,coord2,istep,cont1);
223 else
224 write_vtk_fem(npoint,nelem,nnode,
lnodes,coord,istep,cont1);
225 end
226 end
227
228 if(isolve == 2)
229 cont1=tdisp(ntotv2+1:ntotv);
230
231 if(icrack == 1)
232 write_vtk_fem(npoint,nelem,nnode,
lnodes,coord2,istep,cont1);
233 else
234 write_vtk_fem(npoint,nelem,nnode,
lnodes,coord,istep,cont1);
235 end
236 end
237
238 end %if
239
240 end %end istep
241
242 compute_time = etime(clock(), time0)
243
244 fprintf(out,'compute time: %7d\n',
compute_time);

```

Line numbers:

9:	Get wall clock time beginning of the execution.
12–13:	Assign unit names for the input and output files.
15:	icrack = 0, solution is for single martensite invariant evolution; icrack = 1, martensite evolution at the crack tip (see results and discussion).

17–25:	Open FEM mesh file and output file depending on the value of icrack.
26–35:	Time integration parameters.
28:	Number of time integration steps.
29:	Print frequency to output the results to file.
30:	Number of maximum iterations for Newton–Raphson solution.
31:	Time increment for numerical integration.
32:	Tolerance value for iterative solution.
34:	Total value of boundary displacement increments.
35:	Factor for boundary displacement increments per time steps.
37:	Solution flag: isolve = 1 for un-optimized solution mode, isolve = 2 for optimized solution mode.
40–50:	Depending on the icrack value read the FEM mesh and the control parameters.
52:	Total DOF for displacements per node.
53:	Total DOF for order parameters per node.
54:	Total number of variables in the solution.
55:	Total number of displacement variables in the solution.
56:	Total number of order parameter variables in the solution.
58:	Initialize microstructure (for icrack = 0, introduce a martensite nucleus to the solution vector).
60:	Parameters for numerical integration.
62–66:	If solution is in optimized mode (isolve = 2), precalculate the Cartesian derivatives of shape functions and area/volume of all elements in the solution.
69–240:	Evolve microstructure.
74:	Transfer the values from previous time step to vector tdisp_old.
76–78:	If icrack = 1, increase the displacement increment factor.
80–87:	Depending on the solution type, initialize to global stiffness matrix.
89–185:	Newton–Raphson iterative solution.
91–96:	If solution is in un-optimized mode (isolve = 1), form the global stiffness matrix and global rhs, load, vector.
98–103:	If solution is in optimized mode (isolve = 2), form the global stiffness matrix and global rhs, load, vector.
107–121:	For crack solution (icrack = 1), modify the global stiffness matrix and global rhs, load vector depending on the solution type.
127:	Solve system of equations.
129–134:	Update the unknown vector.
136–147:	Add the incremental order parameter values to unknown vector.

(continued)

136–141:	Update the unknown vector for optimized mode.
143–147:	Add the incremental values to the solution vector, for optimized solution mode.
149–171:	If there are any small deviations from max and min values for order parameters, reset the limits.
151–162:	For un-optimized solution mode.
164–171:	For optimized mode.
173–182:	Check convergence. If convergence is reached, exit from Newton–Raphson iteration.
189–238:	If print frequency is reached, output the results to file.
193–201:	Open and output file and print the coordinates of the nodes and the nodal values to file. These lines are commented out, but they can be changed including the output format.
205–212:	If solution is for crack case, add the current values of the nodal displacements to the coordinates of nodes, in order to see the results in deformed mode.
215–226:	If the solution is in un-optimized mode, write the results in vtk file format for contour plots to be viewed by using Paraview.
228–236:	If the solution is in optimized mode, write the results in vtk file format for contour plots to be viewed by using Paraview.
242–244:	Calculate the total execution time and print it.

Function

bmats1.m

This function rearranges the Cartesian derivatives of shape functions at the current integration point for all elements in the solution.

Variable and array list:

nelem:	Total number of elements.
nnode:	Number of nodes per element.
nstre:	Number of stress components.
nevab:	Total number of variables per element.
kgasp:	Integration point counter.
bmatx1(nelem, ndime,nnode):	Cartesian derivatives of shape function, ndime is the number of Cartesian coordinate dimension.
dgdx(nelem, mgaus,ndime, nnode):	Precalculated Cartesian derivatives of shape functions, mgaus = total integration points per element.

Listing:

```

1  function [bmatx1]=bmats1(dgdx,
2      nelem,nnode,nstre,nevab,kgasp)
3      format long;
4
5      bmatx1 = zeros(nelem,nstre,nevab);
6
7      ngash=0;
8
9      for inode=1:nnode
10
11      bmatx1( :,1,inode)=dgdx( :,kgasp,1,
12      inode);
13      bmatx1( :,2,inode)=dgdx( :,kgasp,2,
14      inode);
15  end %endfunction

```

Line numbers:

5:	Initialize array bmats1.
9–14:	Transfer Cartesian derivatives of shape functions from global array dgdx to array bmats1 for the current integration point.

Function

boundary_cond2_v1.m

This function modifies the global stiffness matrix and global rhs, load, vector for incremental loading, resulting from applied displacements. It is in longhand format and not optimized.

Variable and array list:

npoint:	Total number of nodes.
nvfix:	Total number of nodes that have prescribed boundary conditions.
ndofn:	Number of DOF per node.
facto:	Displacement increment factor.
nofix(nvfix):	List of nodes that have the prescribed displacements.
gforce (ntotv):	Global rhs, load, vector; ntotv = npoin × ndofn.
tdisp(ntotv):	Solution vector.
iffix(nvfix, ndofn):	List of constrained DOFs.
fixed(nvfix, ndofn):	Prescribed values for constrained DOFs.

(continued)

gstif(ntotv, ntotv):	Global stiffness matrix.
treac(nvfix, ndofn):	Reaction force vector resulting from the applied loads/displacements at the constrained boundary nodes.

Listing:

```

1 function[gstif,gforce,treac]
=boundary_cond2_v1(npoin,nvfix,
nofix,iffix, ...
2     fixed,ndofn,facto,gstif, ...
3     gforce,tdisp)
4 format long;
5
6 ndofn2=2;
7 ntotv=npoin*ndofn;
8
9 for ivfix=1:nvfix
10 for idofn=1:ndofn2
11 treac(ivfix,idofn) =0.0;
12 end
13 end
14
15 for ivfix = 1:nvfix
16 lnode = nofix(ivfix);
17
18 for idofn =1:ndofn2
19 if(iffix(ivfix,idofn) == 1)
20 itotv=(lnode - 1)*ndofn2 +idofn;
21 for jtotv = 1:ntotv
22
23 treac(ivfix,idofn) =treac(ivfix,
idofn)- gstif(itotv,jtotv)*tdisp
(jtotv);
24 gstif(itotv,jtotv) = 0.0;
25
26 end
27
28 gstif(itotv,itotv) = 1.0;
29 gforce(itotv) = fixed(ivfix,idofn)
*facto-tdisp(itotv);
30 end
31 end
32 end
33
34 end %endfunction

```

Line numbers:

6:	Number of DOFs for displacements per node.
7:	Total number of variables.

9–13:	Initialize total reaction force vector.
15–32:	Modify global stiffness matrix and global rhs, load, vector.
15:	Loop over number of constrained boundary nodes.
23:	Calculate reaction forces. Multiply row of global stiffness matrix to that corresponding to DOFs with the solution vector.
24:	Zero the row in global stiffness matrix corresponding to constrained DOFs.
28:	Insert value of one to diagonals in the global stiffness matrix corresponding to constrained DOFs.
29:	Put prescribed displacement values to corresponding constrained DOFs and subtract the current total displacement values.

Function

boundary_cond2_v2.m

This function modifies the global stiffness matrix and global rhs, load, vector for incremental loading, resulting from applied displacements. The code is optimized for Matlab/Octave.

Variable and array list:

npoin:	Total number of nodes.
nvfix:	Total number of nodes that have prescribed boundary conditions.
ndofn:	Number of DOFs per node.
facto:	Displacement increment factor.
nofix(nvfix):	List of nodes that have the prescribed displacements.
gforce (ntotv):	Global rhs, load, vector; ntotv = npoin × ndofn.
tdisp(ntotv):	Solution vector.
iffix(nvfix, ndofn):	List of constrained DOFs.
fixed(nvfix, ndofn):	Prescribed values for constrained DOFs.
gstif(ntotv, ntotv):	Global stiffness matrix.
treac(nvfix, ndofn):	Reaction force vector resulting from the applied loads/displacements at the constrained boundary nodes.

Listing:

```

1 function[gstif,gforce,treac]
=boundary_cond2_v2(npoin,nvfix,
nofix,iffix, ...

```

```

2   fixed,ndofn,facto, ...
3   gstif,gforce,tdisp)
4
5   format long;
6
7   ndofn2=2;
8   ntotv=npoin*ndofn;
9   treac=zeros(nvfix,ndofn2);
10
11  for ivfix = 1:nvfix
12    lnode = nofix(ivfix);
13
14  for idofn =1:ndofn2
15    if(iffix(ivfix,idofn) == 1)
16      itotv=(lnode - 1)*ndofn2 +idofn;
17
18    treac(ivfix,idofn) =treac(ivfix,
19      idofn)- gstif(itotv,:)
20      *tdisp(1:ntotv);
21
22    gstif(itotv,: ) = 0.0;
23
24  end
25  end
26  end
27  end
28
29 end %endfunction

```

Line numbers:

7:	Number of DOF for displacements per node.
8:	Total number of variables.
9:	Initialize total reaction force vector.
11–27:	Loop over number of constrained boundary nodes.
18:	Calculate reaction forces. Multiply row of global stiffness matrix to that corresponding to DOFs with the solution vector.
20:	Zero the row in global stiffness matrix corresponding to constrained DOFs.
22:	Insert value of one to diagonals in the global stiffness matrix corresponding to constrained DOFs.
23:	Put prescribed displacement values to corresponding constrained DOFs and subtract the current total displacement values.

Function**init_mart_micro.m**

This function introduces a single martensite nucleus into the FEM mesh, depending on the value of icrack.

Variable and array list:

npoin:	Total number of nodes.
ndofn:	Total DOFs per node.
icrack:	Flag for crack simulation.
tdisp (ntotv):	Solution vector, ntotv = npoin × ndofn.

Listing:

```

1  function[tdisp]=init_mart_micro
2    (npoin,ndofn,icrack)
3
4  format long;
5
6  ndofn2 = 2;
7  ntotv2 = npoin*ndofn2;
8  ntotv = npoin*ndofn;
9  tdisp = zeros(ntotv,1);
10
11 for itotv=ntotv2:ntotv
12   tdisp(itotv)=1.0e-5;
13 end
14
15 %-- Introduce martensite nuclues:
16
17 if(icrack == 0)
18
19 nmarten=4;
20 mrnode(1)=2113;
21 mrnode(2)=2114;
22 mrnode(3)=2178;
23 mrnode(4)=2179;
24 for imarten = 1 : nmarten
25   itotv = ntotv2 +mrnode(imarten);
26   tdisp(itotv) = 0.9;
27 end
28
29 end
30
31 end %endfunction

```

Line numbers:

5:	Number of DOFs for displacements per node.
6:	Total number of displacement variables in the solution.
7:	Total number of variables in the solution.
8:	Initialize the solution vector.
10–12:	Introduce a small number for the order parameters in the solution vector.
15–29:	If icrack = 0, introduce martensite nucleus to the element that is in the center of the FEM mesh.
20–24:	Node numbers of the element.
24–27:	Give the order parameter value of 0.9 to DOFs of the nodes listed in lines 20–25.

Function**marten_free_energ_v1.m**

This function evaluates the derivatives of free energy with respect to martensite invariants. It is in un-optimized mode.

Variable and array list:

eta1gp:	Order parameter value of the first martensite invariant at the integration point.
eta2gp:	Order parameter value of the second martensite invariant at the integration point.
consta:	Value of constant α in the free energy function.
constb:	Value of constant β in the free energy function.
constc:	Value of constant γ in the free energy function.
dfdeta1:	Derivative of free energy with respect to first invariant.
dfdeta2:	Derivative of free energy with respect to second invariant.
df2deta1:	Second derivative of free energy with respect to first invariant.
df2deta2:	Second derivative of free energy with respect to second invariant.
def2deta12:	Second derivative of first derivative of free energy with respect to first invariant with second invariant.
def2deta12:	Second derivative of first derivative of free energy with respect to second invariant with first invariant.

Listing:

```

1  function [dfdeta1,dfdeta2,df2deta1,
2   df2deta2,df2eta12,df2eta21] =
3   marten_free_energ_v1(eta1gp,
4   eta2gp, ...
5   consta,constb,constc)
6   format long;
7
8   dfdeta1 = constc*eta1gp*(eta1gp^2
9   +eta2gp^2) - constb*eta1gp^2 +
10  consta*eta1gp;
11  dfdeta2 = constc*eta2gp*(eta1gp^2 +
12  eta2gp^2) - constb*eta2gp^2 +
13  consta*eta2gp;
14  df2deta1 = constc*(3.0*eta1gp^2+
15  eta2gp^2) - 2.0*constb*eta1gp +
16  consta;
17  df2deta2 = constc*(eta1gp^2+
18  3.0*eta2gp^2) - 2.0*constb*eta2gp
19  + consta;
20  df2eta12 = 2.0*constc*eta1gp
21  *eta2gp;
22  df2eta21 = 2.0*constc*eta1gp
23  *eta2gp;
24
25  end %endfunction

```

Line numbers:

5:	First derivative of free energy with respect to first invariant.
6:	First derivative of free energy with respect to second invariant.
8:	Second derivative of free energy with respect to first invariant.
9:	Second derivative of free energy with respect to second invariant.
11–12:	Cross derivatives for the second derivatives of free energy.

Function**marten_free_energ_v2.m**

This function evaluates the derivatives of free energy with respect to martensite invariants

simultaneously for all elements. It is in optimized mode.

Variable and array list:

nelem:	Number of elements in the solution.
consta:	Value of constant α in the free energy function.
constb:	Value of constant β in the free energy function.
constc:	Value of constant γ in the free energy function.
eta1gp (nelem):	Order parameter value of the first martensite invariant at the integration point.
eta2gp (nelem)	Order parameter value of the second martensite invariant at the integration point.
dfdeta1 (nelem):	Derivative of free energy with respect to first invariant.
dfdeta2 (nelem):	Derivative of free energy with respect to second invariant.
df2deta1 (nelem):	Second derivative of free energy with respect to first invariant.
df2deta2 (nelem):	Second derivative of free energy with respect to second invariant.
df2deta12 (nelem):	Second derivative of first derivative of free energy with respect to first invariant with second invariant.
df2deta12 (nelem):	Second derivative of first derivative of free energy with respect to second invariant with first invariant.

Listing:

```

1  function [dfdeta1,dfdeta2,df2deta1,
2    df2deta2,df2eta12,df2eta21]
3    marten_free_energ_v2(nelem,
4      eta1gp,eta2gp, ...
5      consta,constb,constc)
6    format long;
7
8    dfdeta1 = zeros(nelem,1);
9    dfdeta2 = zeros(nelem,1);
10
11   df2deta1deta2 = zeros(nelem,1);
12   df2deta2deta1 = zeros(nelem,1);
13
14
15   dfdeta1 = constc*eta1gp.* (eta1gp.^2
+ eta2gp.^2) - constb*eta1gp.^2 +
consta*eta1gp;
```

```

16   dfdeta2 = constc*eta2gp.* (eta1gp.^2
+ eta2gp.^2) - constb*eta2gp.^2 +
consta*eta2gp;
17
18   df2deta1 = constc*(3.0*eta1gp.^2+
eta2gp.^2) - 2.0*constb*eta1gp +
consta;
19   df2deta2 = constc*(eta1gp.^2+3.0
*eta2gp.^2) - 2.0*constb*eta2gp +
consta;
20
21   df2eta12 = 2.0*constc*eta1gp.
*eta2gp;
22   df2eta21 = 2.0*constc*eta1gp.
*eta2gp;
23
24 end %endfunction
```

Line numbers:

5-12:	Initialize the arrays holding the derivatives of free energy for all elements in the solution.
15:	First derivative of free energy with respect to first invariant for all elements.
16:	First derivative of free energy with respect to second invariant for all elements.
18:	Second derivative of free energy with respect to first invariant for all elements.
19:	Second derivative of free energy with respect to second invariant for all elements.
21-22:	Cross derivatives for the second derivatives of free energy for elements.

Function

marten_stiff_v1.m

This function calculates the global stiffness matrix and the global rhs, load vector. It is in longhand format and is not optimized.

The function makes calls to the following functions:

- **sfr2.m**
- **jacob2.m**
- **bmats.m**
- **marten_free_energ_v1.m**
- **stress_strain_v1.m**

Variable and array list:

npoint:	Number of nodes.
nelem:	Number of elements.

(continued)

nnode:	Number of nodes per element.
nstre:	Number of stress components.
ndime:	Number of Cartesian coordinate dimension.
ndofn:	Number of DOF per nodes.
ngaus:	Order of numerical integration.
ntype:	Solution type; ntype = 1 for plane-stress and ntype = 2 for plane-strain.
dtime:	Time increment for numerical integration.
iter:	Iteration number.
icrack:	Flag for crack simulation.
tdisp(ntotv):	Solution vector, ntotv = npoin \times ndofn.
tdisp_old (ntotv):	Solution values from previous time step.
posgp(ngaus):	Position of integration points.
weigp(ngaus):	Weights for numerical integration.
matno(nelem):	Element material identity list.
gforce(nelem):	Global rhs, load, vector.
lnods(nelem, nnode):	Element nodal connectivity list.
coord(npoin, ndime):	Cartesian coordinates of the nodes.
gstif(ntotv, ntotv):	Global stiffness matrix.

Listing:

```

1 function [gstif,gforce] =marten_
stiff_v1(npoin,nelem,nnode,nstre,
ndime,ndofn, ...
2     ngaus,ntype,lnods,matno,
3     coord, ...
4     posgp,weigp, ...
5     dtime,tdisp,tdisp_old,iter,
6     gstif,icrack)
7
8 format long;
9
10 %-- Material Parameters:
11
12 [consta,constb,constc,M,G,L,ks,
13 kg,eigen1,cmat_fcc,cmat_bcc] =
14 material_parameters(nstre,icrack);
15
16 ngaus2=ngaus;
17 if(nnode == 3)
18     ngaus2=1;
19
20 end
21
22 mgaus = ngaus*ngaus2;
23
24 ndofn2 = 2;
25 ndofn1 = 1;
26
27
28 ntotv = npoin*ndofn;
29 ntotv1 = npoin*ndofn1;
30 ntotv2 = npoin*ndofn2;
31
32 %--- global rhs
33
34 gforce=zeros(ntotv,1);
35
36 for ielem=1:nelem
37
38 %--initialize global and local
39 % stiffness and rhs vectors
40
41 for ievab =1:nevab2
42     eforce1(ievab) = 0.0;
43
44 for jevab=1:nevab2
45     estif1(ievab,jevab) =0.0;
46
47 end
48
49 for ievab=1:nevab2
50     for jevab=1:nevab1
51         estif2(ievab,jevab) = 0.0;
52
53     end
54
55     for ievab=1:nevab1
56         for jevab=1:nevab1
57             estif3(ievab,jevab)=0.0;
58
59         end
60
61 %--- elemental nodal values
62
63 for inode=1:nnode
64     lnode =lnods(ielem,inode);
65     for idofn=1:ndofn2

```

```

66 ievab=(inode-1)*ndofn2+idofn;      109
67 itotv=(lnode-1)*ndofn2+idofn;      110 [bmatx]=bmats(cartd,shape,inode);
68 eldis(ievab)=tdisp(itotv);          111
69 end                                112 %--- eta values at gauss points:
70 end                                113
71                                         114 eta1gp=0.0;
72 %--- Coordinates of element nodes: 115 eta2gp=0.0;
73                                         116 eta1rgp=0.0;
74 for inode=1:nnode                  117
75 lnode=lnods(ielem,inode);           118 for inode=1:nnode
76 for idime=1:ndime                 119
77 elcod(idime,inode)=coord          120 eta1gp=eta1gp+etal(inode)*shape
     (lnode,idime);                   (inode);
78 end                                121
79                                         122 eta1rgp=eta1rgp+etalr(inode)
80 for idofn=1:ndofn1                123 *shape(inode);
81 ievab=(inode-1)*ndofn1+idofn;      124 end
82 itotv=(lnode-1)*ndofn1+          125
     ntotv2+idofn;
83 etal(ievab)=tdisp(itotv);          126 %---derivatives of free_energy:
84 etalr(ievab)=(tdisp(itotv)        127
     -tdisp_old(itotv));
85 end                                128 [dfdeta1,dfdeta2,df2deta1,
86 end%inode                           df2deta2,df2eta12,df2eta21] =
87                                         129 marten_free_energ_v1(eta1gp,
88 %--- integrate element stiffness:   eta2gp, ...
89                                         130 consta,constb,constc);
90 kgasp=0;                            131 %---Calculate stress terms, strain
91 for igaus=1:ngaus                 132 energy derivatives
92 exisp=posgp(igaus);               133 %--- and effective elasticity
93 for jgaus=1:ngaus2               134 matrix:
94 etasp =posgp(jgaus);             135 [stres,stresb,stres01,dsengdeta1,
95 if(nnode==3)                      cmatx]=stress_strain_v1
96 etasp=posgp(ngaus+igaus);        136 (cmat_bcc,cmat_fcc, ...
97 end                                137
98                                         138 eldis,nstre,ndime,nevab2, ...
99 kgasp=kgasp+1;                    139 bmatx,eigen1,eta1gp);
100                                         140 if(iter==1)
101 [shape,deriv]=sfr2(exisp,etasp,    141 %-- estif1:
     nnode);                         142 for istre=1:nstre
102 [cartd,djacb,gpcod]=jacob2(ielem, 143 for ievab=1:nevab2
     elcod,kgasp,shape,deriv,nnode,    144 dummy(istre,ievab)=0.0;
     ndime);                         145 end
103 dvolu=djacb*weigp(igaus)*weigp 146 end
     (jgaus);                         147
104 if(nnode==3)                     148 for istre=1:nstre
105 dvolu=djacb*weigp(igaus);       149 for ievab=1:nevab2
106 end

```

```

150 for jstre = 1: nstre          194
151 dummy(istre, ievab) = dummy(istre,      195 for ievab = 1: nevab1
    ievab) + cmatx(istre, jstre) ...      196 for jevab = 1: nevab1
152     * bmatx(istre, ievab);           197 for idime = 1: ndime
153 end                           198
154 end                           199 estif4(ievab, jevab) = estif4
155 end                           (ievab, jevab) + kg*G*L*cartd
156                               (idime, ievab)* ...
157 for ievab=1:nevab2           200     cartd(idime, jevab)* dvolu;
158 for jevab=1:nevab2           201 end
159 for istre=1:nstre            202 end
160                               203 end
161 estif1(ievab, jevab)=estif1(ievab, 204
    jevab)+ bmatx(istre, ievab)* ...      205 temp = 0.0;
162 dummy(istre, jevab)*dvolu;       206
163 end                           207 for istre=1:nstre
164 end                           208 temp = temp + eigen1(istre)*
165 end                           (stres01(istre)-2.0*stresb
166                               (istre));
167 %--- form estif2--for eta1      209 end
168                               210
169 for istre=1:nstre            211 temp = temp + ks*(G/L)*df2det1;
170 for ievab=1:nevab1           212
171 dummy(istre, ievab)=0.0;       213 for ievab=1:nevab1
172 end                           214 for jevab=1:nevab1
173 end                           215 estif4(ievab, jevab) = estif4
174                               (ievab, jevab) + shape(ievab)*temp
175 for istre = 1:nstre           *shape(jevab)*dvolu;
176 for ievab = 1:nevab1           216 end
177 dummy(istre, ievab) = dummy(istre, 217 end
    ievab) + (stresb(istre) - ...
178     stres01(istre))*shape      218
        (ievab);                  219 %---
179 end                           220
180 end                           221 for ievab=1:nevab1
181                               222 for jevab=1:nevab1
182 for istre = 1:nstre           223
183 for ievab = 1:nevab2           224 estif4(ievab, jevab) = estif4
184 for jevab = 1:nevab1           (ievab, jevab) + (1.0/M)
185                               *shape(ievab)* ...
186 estif2(ievab, jevab) = estif2      225     shape(jevab)*dvolu;
    (ievab, jevab) + bmatx(istre, 226
    ievab)* ...                   227 end
187     dummy(istre, jevab)*dvolu;   228 end
188                               229
189 end                           230 end %if iter
190 end                           231
191 end                           232 ===== RHS
192                               233
193 %--- form estif4              234 %-- eforce1
194                               235

```

```

236 for istre=1:nstre           276 if(iter == 1)
237 for ievab=1:nevab2          277
238
239 eforce1(ievab) = eforce1(ievab)
+ bmatx(istre, ievab)*stres
(istre)*dvolu;
240 end
241 end
242
243 %--eforce2
244
245 for ievab=1:nevab1
246 eforce2(ievab) = eforce2(ievab) +
shape(ievab)*(etalrgp/(M*dtime))
*dvolu;
247 end
248
249 for idime=1:ndime
250 dummy(idime)=0.0;
251 end
252
253 for idime=1:ndime
254 for ievab=1:nevab1
255 dummy(idime) = dummy(idime)
- kg*G*L*cartd(idime, ievab)
*etalgp;
256 end
257 end
258
259 for idime =1:ndime
260 for ievab =1:nevab1
261 eforce2(ievab) = eforce2(ievab)
- cartd(idime, ievab)*dummy(idime)
*dvolu;
262 end
263 end
264
265 temp = (dsengdata1 +ks*(G/L)
*dfdata1);
266
267 for ievab=1:nevab1
268 eforce2(ievab) = eforce2(ievab)
+ temp*shape(ievab)*dvolu;
269 end
270
271 end %jgaus
272 end %igaus
273
274 %--- assemble element stiffness
275

276 if(iter == 1)
277
278 % assemble estif1:
279
280 for inode=1:nnode
281 lnode = lnods(ielem,inode);
282 for idofn=1:ndofn2
283 ievab =(inode-1)*ndofn2+idofn;
284 itotv =(lnode-1)*ndofn2+idofn;
285 %
286 for jnode=1:nnode
287 knode = lnods(ielem,jnode);
288 for jdofn=1:ndofn2
289 jevab =(jnode-1)*ndofn2+jdofn;
290 jtovt =(knode-1)*ndofn2+jdofn;
291
292 gstif(itotv,jtovt) = gstif(itotv,
jtovt) + estif1(ievab,jevab);
293
294 end
295 end
296 end
297 end
298
299 % assemble estif2
300
301 for inode=1:nnode
302 lnode = lnods(ielem,inode);
303 for idofn=1:ndofn2
304 ievab =(inode-1)*ndofn2+idofn;
305 itotv =(lnode-1)*ndofn2+idofn;
306 %
307 for jnode=1:nnode
308 knode = lnods(ielem,jnode);
309 for jdofn=1:ndofn1
310 jevab =(jnode-1)*ndofn1+jdofn;
311 jtovt =(knode-1)*ndofn1+ntovt2
+jdofn;
312
313 gstif(itotv,jtovt) = gstif(itotv,
jtovt) + estif2(ievab,jevab);
314
315 end
316 end
317 end
318 end
319
320 %--- assemble estif3 as transpose of
estif2:
321

```

```

322 for inode=1:nnode          365
323 lnode = lnods(ielem,inode); 366 %--eforce1
324 for idofn=1:ndofn1         367
325 ievab=(inode-1)*ndofn1+idofn; 368 for inode=1:nnode
326 itotv=(lnode-1)*ndofn1+ntotv2 369 lnode = lnods(ielem,inode);
327 +idofn;                      370 for idofn=1:ndofn2
328 %                           371 ievab=(inode-1)*ndofn2+idofn;
329 for jnode=1:nnode           372 itotv=(lnode-1)*ndofn2+idofn;
330 knode = lnods(ielem,jnode); 373 gforce(itotv) = gforce(itotv)
331 for jdofn=1:ndofn2         374 -eforce1(ievab);
332 jevab=(jnode-1)*ndofn2+jdofn; 375 end
333                               376
334 gstif(itotv,jtotv) = gstif(itotv, 377 %--eforce2
335   jtotv) + estif2(jevab,ievab); 378
336 end                         379 for inode=1:nnode
337 end                         380 lnode = lnods(ielem,inode);
338 end                         381 for idofn=1:ndofn1
339 end                         382 ievab=(inode-1)*ndofn1+idofn;
340                               383 itotv=(lnode-1)*ndofn1+ ntotv2
341 %-- assemble estif4          384 + idofn;
342                               385 gforce(itotv) = gforce(itotv)
343 for inode=1:nnode           386 -eforce2(ievab);
344 lnode = lnods(ielem,inode); 387 end
345 for idofn=1:ndofn1          388 end % ielem
346 ievab=(inode-1)*ndofn1+idofn; 389
347 itotv=(lnode-1)*ndofn1+ntotv2 390 end %endfunction
348 +idofn;
349 %
350 for jnode=1:nnode
351 knode = lnods(ielem,jnode);
352 for jdofn=1:ndofn1
353 jevab=(jnode-1)*ndofn1+jdofn;
354 jtotv=(knode-1)*ndofn1+ntotv2
355 +jdofn;
356 gstif(itotv,jtotv) = gstif(itotv,
357   jtotv) + estif4(ievab,jevab);
358 end
359 end
360 end
361
362 end %if iter
363
364 %--- assemble rhs:

```

Line numbers:

10:	Get material-specific parameters.
14–17:	Order of numerical integration, depending on the element type.
19:	Total integration points per element.
21:	Number of DOF for displacements per node.
22:	Number of DOF for order parameters per node.
24:	Total number of variables in the solution.
25:	Total number of variables for order parameters in the solution
26:	Total number of variables for displacements in the solution.
28:	Total number of variables per element.
29:	Total number of order parameter variables per element.
30:	Total number of displacement variables per element.
34:	Initialize global rhs, load vector.

(continued)

36–388:	Loop over elements.
38–59:	Initialize element stiffness matrix, element stiffness submatrices and rhs, load vector.
63–70:	Get element nodal displacement values.
72–78:	Get element nodal Cartesian coordinates.
80–86:	Get element nodal order parameter values.
88–272:	Numerical integration of elements.
92, 94, 96:	Coordinates of the integration points in local coordinate system.
99:	Increment integration point counter.
101:	Calculate shape functions and their derivative values.
102:	Calculate Cartesian derivatives of shape functions.
104–108:	Calculate element area/volume.
110:	Calculate strain matrix.
112–124:	Calculate the values of the order parameters at the current integration point, Eq. 6.108.
128–129:	Calculate the derivatives of free energy.
134–136:	Calculate the stress terms and derivative of elastic strain energy.
138–230:	If iter = 1, form submatrices of element stiffness matrix, Eq. 6.120.
140–165:	Form submatrix, K_{ij}^{uu} , Eq. 6.114.
167–191:	Form submatrix, $K_{ij}^{uc_1}$, Eq. 6.115.
193–228:	For submatrix, $K_{ij}^{c_1 c_1}$, Eq. 6.117.
232–269	Form rhs, load, vector.
236–241:	Form first row of rhs, load, vector, Eq. 6.111.
243–269:	Form second row of rhs, load, vector, Eq. 6.111.
276–362:	If iter = 1, assemble element sub stiffness matrix from submatrices into the global stiffness matrix.
364–386:	Assemble element rhs, load vector into the global rhs, load vector.

<i>Variable and array list:</i>	
npoin:	Number of nodes.
nelem:	Number of elements.
nnode:	Number of nodes per element.
nstre:	Number of stress components.
ndime:	Number of Cartesian coordinate dimension.
ndofn:	Number of DOF per nodes.
ngaus:	Order of numerical integration.
ntype:	Solution type; ntype = 1 for plane-stress and ntype = 2 for plane-strain.
ndtime:	Time increment for numerical integration.
iter:	Iteration number.
icrack:	Flag for crack simulation.
tdisp(ntotv):	Solution vector, ntotv = npoin × ndofn.
tdisp_old(ntotv):	Solution values from previous time step.
posgp(ngaus):	Position of integration points.
weigp(ngaus):	Weights for numerical integration.
matno(nelem):	Element material identity list.
gforce(nelem):	Global rhs, load, vector.
lnods(nelem, nnode):	Element nodal connectivity list.
coord(npoin, ndime):	Cartesian coordinates of the nodes.
gstif(ntotv,ntotv):	Global stiffness matrix.
volume(nelem, mgaus):	Area/volume of the elements.
dgdx(nelem, mgaus,ndime, nnode):	Precalculated Cartesian derivatives of shape functions of all elements in the simulation.

Listing:

```

1 function [gstif,gforce] =marten_
stiff_v2(npoin,nelem,nnode,
nstre,ndime,ndofn, ...
2      ngaus,ntype,lnods,matno,
      coord, ...
3      posgp,weigp, ...
4      dgdx,dvolum,dtime,tdisp,
      tdisp_old, ...
5      iter,gstif,icrack)
6
7 format long;
8
9 %% Material Parameters:
10

```

Function

marten_stiff_v2.m

This function calculates the global stiffness matrix and the global rhs, load vector. It is optimized for Matlab/Octave.

The function makes calls to the following functions:

- **sfr2.m**
- **bmats1.m**
- **bmats2.m**
- **marten_free_energ_v2.m**
- **stress_strain_v2.m**

```

11 [consta,constb,constc,M,G,L,ks,
12   kg,eigen1,cmat_fcc,cmat_bcc] =
13 material_parameters(nstre,icrack);
14
15 ngaus2=ngaus;
16 if(nnnode == 3)
17 ngaus2=1;
18 end
19
20 mgaus = ngaus*ngaus2;
21
22 %--initialize global and local
23 stiffness and rhs vectors
24 ndofn2 = 2;
25 ndofn1 = 1;
26
27 ntotv = npoin*ndofn;
28 ntotv1 = npoin*ndofn1;
29 ntotv2 = npoin*ndofn2;
30
31 nevab = nnnode*ndofn;
32 nevab1 = nnnode*ndofn1;
33 nevab2 = nnnode*ndofn2;
34
35 %--- elasticity matrix:
36
37 estif1 = zeros(nelem,nevab2,
38   nevab2);
39 estif2 = zeros(nelem,nevab2,
40   nevab1);
41 estif3 = zeros(nelem,nevab2,
42   nevab1);
43 estif4 = zeros(nelem,nevab1,
44   nevab1);
45
46 %--- global rhs
47 gforce = sparse(ntotv,1);
48
49 %--- elemental nodal values
50
51 eldis = zeros(nelem,nevab2);
52 eta1 = zeros(nelem,nevab1);
53 eta1r = zeros(nelem,nevab1);
54
55 for inode=1:nnnode
56
57 lnode = lnods( :,inode );
58
59 for idofn=1:ndofn2
60 ievab = (inode-1)*ndofn2+idofn;
61 itotv = (lnode-1)*ndofn2+idofn;
62 eldis( :,ievab ) = tdisp(itotv);
63 end
64
65 for idofn=1:ndofn1
66 ievab = (inode-1)*ndofn1+idofn;
67 itotv = (lnode-1)*ndofn1+ntotv2
68 +idofn;
69 eta1( :,ievab ) = tdisp(itotv);
70 eta1r( :,ievab ) = (tdisp(itotv)
71 -tdisp_old(itotv));
72 end
73 end%inode
74
75 %--- integrate element stiffness:
76 kgas=0;
77 for igaus=1:ngaus
78 exisp=posgp(igaus);
79 etasp =posgp(jgaus);
80 if(nnnode ==3)
81 etasp=posgp(ngaus+igaus);
82 end
83
84 kgas=kgas+1;
85
86 [shape,deriv]=sfr2(exisp,etasp,
87 nnnode);
88 [bmatx1]=bmats1(dgdx,nelem,nnnode,
89 ndime,nevab1,kgas);
90 [bmatx2]=bmats2(dgdx,nelem,
91 nnnode,nstre,nevab2,kgas);
92 %--- eta values at gauss points:
93
94 eta1gp = zeros(nelem,1);
95 eta2gp = zeros(nelem,1);
96 eta1rgp = zeros(nelem,1);

```

```

97
98   for inode =1:nnode
99
100  eta1gp = eta1gp + eta1( :,inode)
101    *shape(inode);
102  eta1rgp = eta1rgp + eta1r( :,inode)
103    *shape(inode);
104 end
105
106 %---derivatives of free_energy:
107
108 [dfdeta1,dfdeta2,df2deta1,
109  df2deta2,df2eta12,df2eta21] =
110  marten_free_energ_v2(nelem,
111  eta1gp,eta2gp, ...
112    consta,constb,constc);
113 %---Calculate stress terms,
114  strain energy derivatives
115 %--- and effective elasticity
116 matrix:
117 if(iter == 1)
118 %-- estif1:
119 dummy = zeros(nelem,nstre,nevab2);
120
121 for istre = 1:nstre
122 for ievab = 1:nevab2
123 for jstre = 1: nstre
124 dummy( :,istre,ievab) = dummy( :,
125  istre,ievab) + cmatx( :,istre,
126  jstre) ...
127    .*bmatx2( :,istre,ievab);
128 end
129 end
130 end
131 for ievab=1:nevab2
132 for jevab=1:nevab2
133 for istre=1:nstre
134 for ievab=1:nevab1
135
136 estif1( :,ievab,jevab)=estif1( :,
137  ievab,jevab)+ bmatx2( :,istre,
138  ievab).* ...
139      dummy( :,istre,jevab).
140      *dvolum( :,kgasp);
141 end
142 estif1 = estif1 +1.0e-6;
143
144 %--- form estif2--for eta1
145
146 dummy =zeros(nelem,istre,nevab1);
147
148 for istre = 1:nstre
149 for ievab = 1:nevab1
150 dummy( :,istre,ievab) =dummy( :,
151  istre,ievab) +(stresb( :,istre) -...
152  stres01( :,istre))*shape
153 (ievab);
154 end
155 end
156 for ievab = 1:nevab2
157 for jevab = 1:nevab1
158 estif2( :,ievab,jevab) =estif2( :,
159  ievab,jevab) + bmatx2( :,istre,
160  ievab).* ...
161      dummy( :,istre,jevab).
162      *dvolum( :,kgasp);
163 end
164 %--- form estif4
165
166 for ievab=1:nevab1
167 for jevab=1:nevab1
168 for idime=1:ndime
169
170 estif4( :,ievab,jevab)=estif4( :,
171  ievab,jevab) + kg*G*L*bmatx1( :,
172  idime,ievab).* ...
173      bmatx1( :,idime,jevab).
174      * dvolum( :,kgasp);
175 end
176 end
177 end

```

```

175
176 dummy=zeros(nelem,1);
177
178 for istre=1:nstre
179 dummy=dummy+eigen1(istre)*
180   (stres01(:,istre)-2.0*stresb
181   (:,istre));
182 end
183
184 dummy=dummy+ks*(G/L)*df2deta1;
185
186 for ievab=1:nevab1
187   estif4(:,ievab,jevab)=estif4
188   (:,ievab,jevab)+shape(ievab)*
189   dummy*shape(jevab).*...
190   dvolum(:,kgasp);
191 end
192 %---
193 for ievab=1:nevab1
194   for jevab=1:nevab1
195     estif4(:,ievab,jevab)=estif4
196     (:,ievab,jevab)+(1.0/M)*shape
197     (ievab)*...
198     shape(jevab).*dvolum(:,kgasp);
199 end
200 end
201
202 end %if iter
203
204 ===== RHS
205
206 %-- eforce1
207
208 for istre=1:nstre
209 for ievab=1:nevab2
210
211   eforce1(:,ievab)=eforce1
212   (:,ievab)+bmatx2(:,istre,ievab)*
213   stres(:,istre).*...
214   dvolum(:,kgasp);
215 end
216 %--eforce2
217
218 for ievab=1:nevab1
219   eforce2(:,ievab)=eforce2(:,ievab)
220   +shape(ievab)*(eta1rgp/
221   (M*dtime)).*...
222   dvolum(:,kgasp);
223 end
224
225 for idime=1:ndime
226 for ievab=1:nevab1
227   dummy(:,idime)=dummy(:,idime)
228   -kg*G*L*bmatx1(:,idime,ievab)*
229   eta1gp;
230 end
231 for idime=1:ndime
232 for ievab=1:nevab1
233   eforce2(:,ievab)=eforce2(:,ievab)
234   -bmatx1(:,idime,ievab).*...
235   dummy(:,idime).*...
236   dvolum(:,kgasp);
237 end
238 dummy=zeros(nelem,1);
239
240 dummy=(dsengdeta1+ks*(G/L)
241   *dfdeta1);
242 for ievab=1:nevab1
243   eforce2(:,ievab)=eforce2(:,ievab)
244   +dummy*shape(ievab).*...
245   dvolum(:,kgasp);
246 end %jgaus
247 end %igaus
248
249 %--- assemble element stiffness
250
251 if(iter==1)
252   % assemble estif1:
253
254 for inode=1:nnode
255   lnode=lnods(:,inode);
256   for idofn=1:ndofn2

```

```

258 ievab = (inode-1)*ndofn2+idofn;
259 itotv = (lnode-1)*ndofn2+idofn;
260 %
261 for jnode=1:nnode
262 knode = lnods( :,jnode);
263 for jdofn=1:ndofn2
264 jevab = (jnode-1)*ndofn2+jdofn;
265 jtotv = (knode-1)*ndofn2+jdofn;
266
267 gstif = gstif +sparse(itotv,jtotv,
268 estif1( :,ievab,jevab),ntotv,
269 ntotv);
270 end
271 end
272 end
273
274 % assemble estif2
275
276 for inode=1:nnode
277 lnode = lnods( :,inode);
278 for idofn=1:ndofn2
279 ievab = (inode-1)*ndofn2+idofn;
280 itotv = (lnode-1)*ndofn2+idofn;
281 %
282 for jnode=1:nnode
283 knode = lnods( :,jnode);
284 for jdofn=1:ndofn1
285 jevab = (jnode-1)*ndofn1+jdofn;
286 jtotv = (knode-1)*ndofn1+ntotv2
287 +jdofn;
288 gstif = gstif +sparse(itotv,jtotv,
289 estif2( :,ievab,jevab),ntotv,
290 ntotv);
291 end
292 end
293 end
294
295 %--- assemble estif2 as a transpose:
296
297 for inode=1:nnode
298 lnode = lnods( :,inode);
299 for idofn=1:ndofn1
300 ievab = (inode-1)*ndofn1+idofn;
301 itotv = (lnode-1)*ndofn1+ntotv2
302 +idofn;
303 for jnode=1:nnode
304 knode = lnods( :,jnode);
305 for jdofn=1:ndofn2
306 jevab = (jnode-1)*ndofn2+jdofn;
307 jtotv = (knode-1)*ndofn2+jdofn;
308
309 gstif = gstif +sparse(itotv,jtotv,
310 estif2( :,jevab,ievab),ntotv,
311 ntotv);
312 end
313 end
314 end
315
316 %-- assemble estif4
317
318 for inode=1:nnode
319 lnode = lnods( :,inode);
320 for idofn=1:ndofn1
321 ievab = (inode-1)*ndofn1+idofn;
322 itotv = (lnode-1)*ndofn1+ntotv2+id
323 ofn;
324 %
325 for jnode=1:nnode
326 knode = lnods( :,jnode);
327 for jdofn=1:ndofn1
328 jevab = (jnode-1)*ndofn1+jdofn;
329 jtotv = (knode-1)*ndofn1+ntotv2+jd
330 ofn;
331
332 gstif = gstif +sparse(itotv,jtotv,
333 estif4( :,ievab,jevab),ntotv,
334 ntotv);
335 end
336
337 end %if iter
338
339 %--- assemble rhs:
340
341 %--eforcel
342
343 for inode=1:nnode
344 lnode = lnods( :,inode);
345 for idofn=1:ndofn2

```

```

346 ievab=(inode-1)*ndofn2+idofn;
347 itotv=(lnode-1)*ndofn2+idofn;
348 gforce=gforce + sparse(itotv,1,
-eforce1( :,ievab ),ntotv,1);
349 end
350 end
351
352 %--eforce2
353
354 for inode=1:nnode
355 lnode=lnods( :,inode );
356 for idofn=1:ndofn1
357 ievab=(inode-1)*ndofn1+idofn;
358 itotv=(lnode-1)*ndofn1+ntotv2 +
idofn;
359 gforce=gforce + sparse(itotv,1,
-eforce2( :,ievab ),ntotv,1);
360 end
361 end
362
363 end %endfunction

```

Line numbers:

11:	Get material-specific parameters.
15–18:	Order of numerical integration, depending on the element type.
20:	Total integration points per element.
24:	Number of DOF for displacements per node.
25:	Number of DOF for order parameters per node.
27:	Total number of variables in the solution.
28:	Total number of variables for order parameters in the solution.
29:	Total number of variables for displacements in the solution.
31:	Total number of variables per element.
32:	Total number of variables for order parameters per element.
33:	Total number of variables for displacements per element.
35–43:	Initialize element stiffness submatrices and rhs, load, vectors for all elements in the solution.
47:	Initialize global rhs, load, vector.
49–54:	Initialize the arrays holding elemental nodal values for all elements.
55–71:	Get elemental nodal values for all elements.
59–63:	Nodal values for displacements.
65–70:	Nodal values for order parameters.
73–247:	Numerical integration of all elements.
77,	Coordinates of the integration points in the local coordinate system.
79, 81:	

84:	Increment integration point counter.
86:	Calculate the shape functions and their derivative values.
88:	Cartesian derivatives of shape functions for all elements.
90:	Form strain matrix for all elements.
92–104:	Calculate the order parameter values at the current integration point for all elements, Eq. 6.108.
108–109:	Calculate derivatives of free energy for all elements.
113–115:	Calculate the stress terms and derivative of elastic energy for all elements.
117–202:	If iter = 1, form stiffness submatrices of all elements, Eq. 6.120.
119–143:	Form submatrix, K_{ij}^{uu} , Eq. 6.114.
144–162:	Form submatrix, $K_{ij}^{uc_1}$, Eq. 6.115.
164–200:	For submatrix, $K_{ij}^{c_1 c_1}$, Eq. 6.117.
204–244:	Form rhs, load, vector.
206–214:	Form first row of rhs, load, vector, Eq. 6.111.
216–244:	Form second row of rhs, load, vector, Eq. 6.111.
251–337:	If iter = 1, assemble element stiffness submatrices, directly into global stiffness matrix.
339–361:	Assemble element rhs, load vector into the global rhs, load vector.

Function**material_parameters.m**

This function contains the material-specific parameters for martensitic transformations.

Variable and array list:

nstre:	Number of stress components.
icrack:	Parameter for crack solution.
M:	Mobility parameter.
G:	Interface energy density.
L:	Interface parameter.
ks:	Separation energy coefficient.
kg:	Gradient energy coefficient.
eigen1(nstre):	Eigen strains of first martensite invariant.
cmat_fcc(nstre, nstre):	Elasticity matrix for austenite phase.
cmat_bcc(nstre, nstre):	Elasticity matrix for martensite phase.

Listing:

```

1 function [consta,constb,constc,M,G,
L,ks/kg,eigen1,cmat_fcc,cmat_bcc] =
material_parameters(nstre,icrack)
2
3 format long;
4
5 %-- Free energy parameters:
6
7 consta = 0.15;
8 constb = 3.0*consta +12.0;
9 constc = 2.0*consta +12.0;
10
11 %-- Mobility & Gradient energy:
12
13 M = 0.001;
14 G = 1.0e-4;
15 L = 5.0e-5;
16
17 ks= 73.25;
18 kg= 39.5;
19
20 %-- Eigen strains:
21
22 eigen1(1) = 0.100;
23 eigen1(2) == -0.100;
24 eigen1(3) = 0.0;
25
26 if(icrack == 1)
27 eigen1(1) = 0.0100;
28 eigen1(2) == -0.0100;
29 eigen1(3) = 0.1;
30 end
31
32
33 %-- Elasticity matrix of Phases:
34
35 cmat_fcc = zeros(nstre,nstre);
36 cmat_bcc = zeros(nstre,nstre);
37
38 %-- fcc (Austinite Matrix):
39
40 cmat_fcc(1,1)=221880.0;
41 cmat_fcc(1,2)=149840.0;
42 cmat_fcc(2,1)=149840.0;
43 cmat_fcc(2,2)=221880.0;
44 cmat_fcc(3,3)= 36030.0;
45

```

```

46 %-- bcc (Martensite Phase):
47
48 cmat_bcc(1,1)=292700.0;
49 cmat_bcc(1,2)=106250.0;
50 cmat_bcc(2,1)=106250.0;
51 cmat_bcc(2,2)=292700.0;
52 cmat_bcc(3,3)= 93220.0;
53
54 end %endfunction

```

Line numbers:

7:	Coefficient α in the free energy.
8:	Coefficient β in the free energy.
9:	Coefficient γ in the free energy.
13:	Mobility parameter.
14:	Interfacial energy density.
15:	Parameter controls the interface width.
17:	Coefficient of separation energy.
18:	Coefficient of gradient energy.
20–30:	Components of eigenstrain, depending on the value of icrack.
35–36:	Initialize elasticity matrices of the phases.
40–44:	Elasticity matrix for austenite phase.
48–52:	Elasticity matrix for martensite phase.

Function**stress_strain_v1.m**

This function calculates stress components, composite elasticity matrix, and derivative of elastic strain energy at the integration points. It is in longhand format and is not optimized.

Variable and array list:

nstre:	Number of stress components.
ndime:	Number of Cartesian coordinate dimensions.
nevab2:	Total number of variables for displacements per element.
eta1gp:	Order parameter value at the current integration point.
dsendeta1:	Derivative of elastic strain energy.
stres(nstre):	Nominal stress components.
stresb(nstre):	Differential stress components
stres01(nstre):	Stress components from eigenstrain
eldis(ntotv2):	Elemental nodal values of displacements.
eigen1(nstre):	Components of eigenstrain.
bmatx(nstre,	Strain matrix.
nevab2):	

(continued)

cmatx_bcc (nstre,nstre):	Elasticity matrix for martensite phase.
cmatx_fcc (nstre,nstre):	Elasticity matrix for austenite phase.

Listing:

```

1 function [stres,stresb,stres01,
2 dsengdeta1,cmatx] =
3 stress_strain_v1(cmatx_bcc,
4 cmatx_fcc, ...
5 eldis,nstre,ndime,nevab2, ...
6 bmatx,eigen1,etalgp)
7 format long;
8
9 %-- Initialize:
10
11 stres(istre) = 0.0;
12 stresb(istre) = 0.0;
13 stres01(istre) = 0.0;
14 stres02(istre) = 0.0;
15
16 stran(istre) = 0.0;
17 end
18
19 %-- form composite elasticity matrix:
20
21 for istre=1:nstre
22 for jstre=1:nstre
23
24 cmatx(istre,jstre) = cmatx_fcc
25 (istre,jstre) + etalgp*(cmatx_bcc
26 (istre,jstre)- ...
27 cmatx_fcc(istre,jstre));
28 end
29 end
30
31 for istre =1:nstre
32 for jstre =1:nstre
33 cmatx_diff(istre,jstre) = cmatx_bcc
34 (istre,jstre)-cmatx_fcc(istre,
35 jstre);
36
37 %-- calculate strains:
38
39 for istre =1:nstre
40 for ievab =1:nevab2
41
42 stran(istre) = stran(istre) + bmatx
43 (istre,ievab)*eldis(ievab);
44 end
45 end
46
47 %-- calculate stress:
48
49 for istre=1:nstre
50 for jstre=1:nstre
51
52 stres(istre) = stres(istre) + cmatx
53 (istre,jstre)*(stran(jstre) - ...
54 etalgp*eigen1(jstre));
55 end
56 end
57
58 %-- calculate stresb:
59
60 for istre=1:nstre
61 for jstre=1:nstre
62
63 stresb(istre) = stresb(istre) +
64 cmatx_diff(istre,jstre)*(stran
65 (jstre) - ...
66 etalgp*eigen1(jstre));
67 end
68 end
69
70 for istre=1:nstre
71 for jstre=1:nstre
72 stres01(istre) = stres01(istre) +
73 cmatx(istre,jstre)*eigen1(jstre);
74 end
75
76 %-- derivative of strain-energy
77 for eta1 & eta2:
78 dsengdeta1 =0.0;
79

```

```

80 for istre=1:nstre
81 dsengdeta1 = dsengdeta1 + 0.5*stran
82 (istre)*stresb(istre) - ...
83 eigen1(istre)*stres(istre);
84 end
85
86
87 end %endfunction

```

cmatx_bcc (nstre,nstre):	Elasticity matrix for martensite phase.
cmatx_fcc(nstre, nstre):	Elasticity matrix for austenite phase.
bmatx2(nelem, nstre,nevab2):	Strain matrix.

Listing:

```

1 function [stres,stresb,stres01,
2 dsengdeta1,cmatx] =
3 stress_strain_v2(nelem,
4 cmatx_bcc,cmatx_fcc, ...
5 eldis,nstre,ndime,nevab2, ...
6 bmatx2,eigen1,eta1gp)
7
8 format long;
9
10 %%-- Initialize:
11
12 cmatx=zeros(nelem,3,3);
13
14 stres=zeros(nelem,3);
15 stresb=zeros(nelem,3);
16 stres01=zeros(nelem,3);
17 stres02=zeros(nelem,3);
18
19 stran=zeros(nelem,3);
20
21 %%-- form composite elasticity tensor:
22
23 for istre=1:nstre
24 for jstre=1:nstre
25
26 cmatx( :,istre,jstre) = cmatx_fcc
27 (istre,jstre) + eta1gp*(cmatx_bcc
28 (istre,jstre)- ...
29 cmatx_fcc(istre,jstre));
30 end
31 end
32
33 cmatx_diff=cmatx_bcc-cmatx_fcc;
34

```

Line numbers:

9–17:	Initialize stress components.
21–27:	Form composite elasticity matrix, Eq. 6.97.
29–35:	Form differential elasticity matrix, Eq. 6.113.
39–45:	Calculate strain values from nodal displacements, Eq. 6.108.
47–56:	Calculate overall stress, Eq. 6.101.
60–64:	Calculate stress, $\tilde{\sigma}$, Eq. 6.113.
68–74:	Calculate stress, σ_1^0 , Eq. 6.113.
76–84:	Calculate the derivative of elastic strain energy with respect to first martensite invariant

Function

stress_strain_v2.m

This function calculates stress components, composite elasticity matrix, and derivative of elastic strain energy at the integration points. It is optimized for Matlab/Octave.

Variable and array list:

nelem:	Number of elements in the solution.
nstre:	Number of stress components.
ndime:	Number of Cartesian coordinate dimension.
nevab2:	Total number of variables for displacements per element.
eigen1(nstre):	Components of eigenstrain.
eta1gp(nelem):	Order parameter value at the current integration point.
dsendeta1 (nelem):	Derivative of elastic strain energy.
stres(nelem, nstre):	Nominal stress components.
stresb(nelem, nstre):	Differential stress components
stres01(nelem, nstre):	Stress components from eigenstrain.
eldis(nelem, ntoty2):	Elemental nodal values of displacements.

```

35 %-- calculate strains:
36
37 for istre=1:nstre
38 for ievab=1:nevab2
39
40 stran( :,istre) = stran( :,istre) +
bmatx2( :,istre,ievab).*eldis( :,ievab);
41
42 end
43 end
44
45 %-- calculate stress:
46
47 for istre=1:nstre
48 for jstre=1:nstre
49
50 stres( :,istre) = stres( :,istre)
+cmatx( :,istre,jstre).*(stran( :,jstre) - ...
51 eta1gp*eigen1(jstre));
52
53 end
54 end
55
56 %-- calculate stresb:
57
58 for istre=1:nstre
59 for jstre=1:nstre
60
61 stresb( :,istre) = stresb( :,istre)
+cmatx_diff(istre,jstre)*(stran( :,jstre) - ...
62 eta1gp*eigen1(jstre));
63 end
64 end
65
66 %-- calculate stres01
67
68 for istre=1:nstre
69 for jstre=1:nstre
70 stres01( :,istre) = stres01( :,istre)
+cmatx( :,istre,jstre)*eigen1
(jstre);
71 end
72 end
73
74 %-- derivative of strain-energy
for eta1 & eta2:
75

```

```

76 for istre=1:nstre
77 dsengdetal = dsengdetal + 0.5*stran
( :,istre).*stresb( :,istre) - ...
78 eigen1(istre)*stres( :,istre);
79
80 end
81
82 end %endfunction

```

Line numbers:

9:	Initialize composite elasticity matrix for all elements.
11–14:	Initialize stress components for all elements.
19:	Initialize derivative of elastic strain energy for all elements.
21–29:	Form composite elasticity matrix for all elements, Eq. 6.97.
33:	Form differential elasticity matrix for all elements, Eq. 6.113.
37–43:	Calculate strain values from nodal displacements.
47–54:	Calculate overall stress for all elements, Eq. 6.101.
58–64:	Calculate stress, $\tilde{\sigma}$, for all elements, Eq. 6.113.
68–72:	Calculate stress, σ_1^0 , for all elements, Eq. 6.113.
76–80:	Calculate the derivative of elastic strain energy with respect to first martensite invariant for all elements.

References

- Verhoeven JD (2007) Steel Metallurgy for the non-metallurgist. American Society for Metals
- Ashby MF, Jones DRH (1992) Engineering materials. Pergamon Press, Oxford
- Patoor E, Lagoudas DC, Entchev PB, Brinson LC, Gao X (2006) Shape memory alloys, Part I: general properties and modeling of single crystals. Mech Mater 38:391
- Olson GB, Hartman H (1982) Martensite and life: displacive transformations as biological processes. J Phys Colloques 43:855
- Rubini S, Ballone P (1993) Quasiharmonic and molecular-dynamics study of the martensitic transformation in Ni-Al alloys. Phys Rev B 48:99
- Entel P, Mayer R, Kadau K (2000) Molecular dynamics simulations of martensitic transformations. Philos Mag B 80:183
- Sondoval L, Urbassek HM (2009) Transformation pathways in the solid-solid transitions of iron nanowires. Appl Phys Lett 95:191909

8. Engin C, Urbassek HM (2008) Molecular dynamics investigation of the fcc \geq bcc phase transformations in Fe. *Comput Mater Sci* 41:297
9. Wang Y and Khachaturyan AG (1997) Three-dimensional field model and computer modeling of martensitic transformations. *Acta Mater* 45:759
10. Artemev A, Wang Y, Khachaturyan AG (2000) Three-dimensional phase field model and simulation of martensitic transformation in multilayer systems under applied stress. *Acta Mater* 48:2503
11. Jin YM, Aremev A, Khachaturyan AG (2001) Three-dimensional phase-field model of low-symmetry martensitic transformation in polycrystal: Simulation of ζ martensite in AuCd alloys. *Acta Mater* 49:2309
12. Zhang W, Jin YM, Khachaturyan AG (2007) Phase-field microelasticity modeling of heterogeneous nucleation and growth in martensitic alloys. *Acta Mater* 55:565
13. Hildebrand FE, Miehe C (2012) Comparison of two bulk energy approaches for the phase-field modeling of two-variant martensitic laminate microstructures. *Tech Mech* 32:3
14. Schmitt R, Muller R, Khun C (2013) A phase-field approach for multivariant martensitic transformations of stable and metastable phases. *Arch Appl Mech* 83:849
15. Schmitt R, Wang B, Urbassek HM, Muller R (2013) Modeling of martensitic transformations in pure iron by a phase-field approach using information from atomistic simulation. *Tech Mech* 33:119
16. Mamivand M, Zaeem MA, Kadiri HE (2013) A review on phase-field modeling of martensitic transformation. *Comput Mater Sci* 77:304
17. Schrade D, Muller R, Xu BX, Gross D (2007) Domain evolution in ferroelectric materials: a continuum phase-field model and finite element implementation. *Comput Methods Appl Mech Eng* 196:2007
18. Yamanaka A, Takaki T, Tomita Y (2008) Elastoplastic phase-field model simulation of self-and plastic accommodation in cubic-> tetragonal martensitic transformation. *Mater Sci Eng A* 491:378

6.10 Case Study-XV

Phase-field modeling of brittle fracture

Objective:

This case study is another example of fully coupled solution of elasticity and phase-field equations with FEM algorithm which also incorporates the incremental external loading.

6.10.1 Background

The prediction of the unstable extension of preexisting cracks, fracture, and their growth path is primary importance of safe operations of engineering structures and components. Similarly, the micromechanistic understanding of fracture is crucial for the development of materials that are resistant to premature failure in the presence of crack-like defects. The theoretical foundation of fracture is based on the early works of Griffith [1] and Irwin [2]. Based on the thermodynamic framework, they postulated that the unstable propagation of existing crack begins when the elastic strain-energy at the crack tip becomes equal or greater than the energy required for creation of two new surfaces.

Over the years, variety of computational algorithms, based on the conventional FEM and Boundary Element Method, BEM, have been developed for modeling and simulation of fracture in some degree of detail. The main difficulty, in these classical sharp-interface crack approaches, arises in the determination of the crack path, their kinking and splitting during their propagation. These events are commonly handled either imposing unrealistic underlying physics into the constitutive models or computationally very cumbersome and demanding algorithms.

There is a growing interest in modeling and simulation of fracture with the framework of phase-field modeling, owing to its superiority and the natural way of handling evolving interfaces. A few examples of the application of phase-field modeling to brittle fracture in linear elasticity can be found in [3–11]. Its extensions to materials behaving elastic-plastic are given in [12–14]. It is recently also applied to analyze the propagation behavior of fluid-filled cracks in porous media [15].

6.10.2 Phase-Field Model

In this case study, phase-field fracture model proposed in [8, 9, 11] is utilized. In the model,

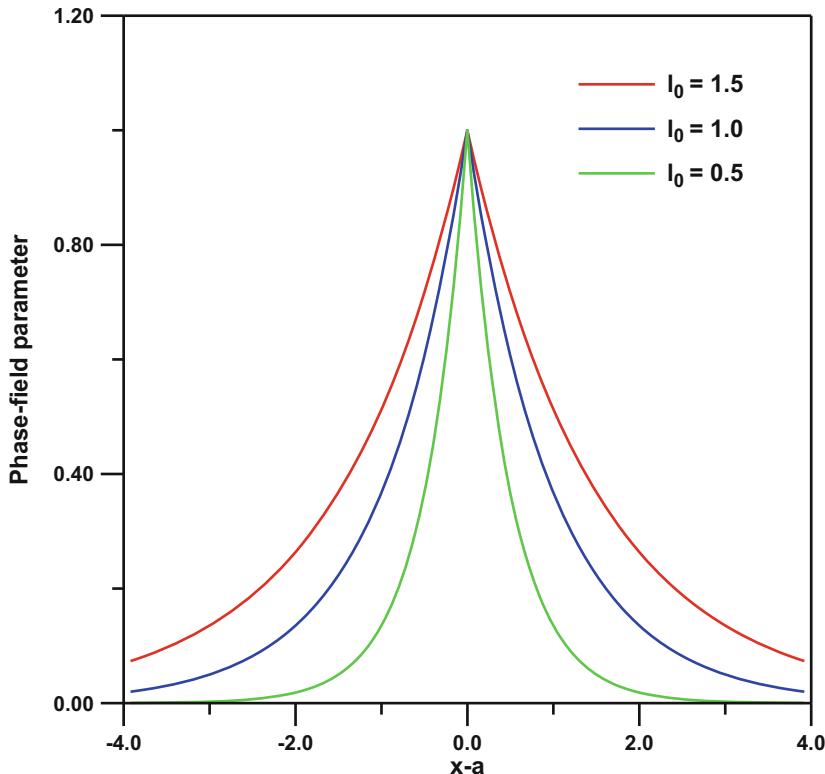


Fig. 6.15 Variation of crack phase-field parameter ϕ with different values of l_0

the non-conserved phase-field parameter ϕ varies between the values of zero (no damaged/uncracked) and one (fully damaged/cracked). This concept has been borrowed from earlier diffuse fracture models [16] and is associated with the loss of area that can withstand to applied tractions. For one-dimensional case, it is expressed as:

$$\phi = \exp\left(\frac{-|x-a|}{l_0}\right) \quad (6.123)$$

where x is the distance, a is the initial crack length, and l_0 is length parameter which controls the spread of damage. The sharp solution limit is approached as $l_0 \rightarrow 0$. The interrelationship between the l_0 and ϕ in Eq. 6.123 is schematically shown in Fig. 6.15 where the crack front is in a direction perpendicular to plane of the figure.

In the presence of crack/damage the total potential energy functional takes the form:

$$\begin{aligned} \Psi(\phi, u) = & \int_V \left[(1 - \phi)^2 + k \right] \psi(\epsilon) dV \\ & + \int_V \frac{G_c}{2} \left[l_0 \nabla \phi \cdot \nabla \phi + \frac{1}{l_0} \phi^2 \right] dV \end{aligned} \quad (6.124)$$

In the above equation, the first integrand represents the change of the strain energy due to change in the crack/damage. k is a parameter chosen for numerical convenience. Its value is taken as small as possible while keeping the system of equations well conditioned. The second integrand represents the energy dissipation due to creation of new crack surfaces with changes in crack/damage. The parameter G_c is the critical energy release for unstable crack/damage extension which is a material property.

The term $\psi(\epsilon)$ is the elastic strain energy and expressed as:

$$\psi(\epsilon) = \frac{1}{2} \epsilon C \epsilon \quad (6.125)$$

in which C is the elasticity matrix and strains ϵ are related to displacements usual way for small strain formulation:

$$\epsilon_{ij} = \frac{1}{2} \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) \quad (6.126)$$

6.10.3 FEM Formulation

For Eq. 6.124, starting with variation of the external work increment, δW_{ext} , the governing equations in the weak form are:

$$\delta W_{ext} = \int_V b_j \delta u_j dV + \int_{\partial V} h_j \delta u_j \partial V \quad (6.127)$$

in which b_j and h_j are the components of body forces and boundary tractions on the surfaces ∂V , respectively.

The change in the internal energy is given by:

$$\delta W_{int} = \delta \Psi = \frac{\partial \Psi}{\partial \epsilon_{ij}} \delta \epsilon_{ij} + \frac{\partial \Psi}{\partial \phi} \delta \phi \quad (6.128)$$

and applying Eqs. 6.128 to 6.124 yields:

$$\begin{aligned} \delta \Psi = & \int_V \left[(1 - \phi)^2 + k \right] \sigma_{ij} \delta \epsilon_{ij} dV + \int_V -2(1 - \phi) \delta \phi \psi(\epsilon) dV \\ & + \int_V G_c \left(l_o \frac{\partial \phi}{\partial x_i} \frac{\partial \phi}{\partial x_i} + \frac{1}{l_0} \phi \delta \phi \right) dV \end{aligned} \quad (6.129)$$

in which δ is the test functions.

As before, by taking nodal variables as displacements, u , and the order parameter, ϕ ,

and utilizing the element shape functions, u , ϕ , $\nabla \phi$, and ϵ are discretized at element level as:

$$u = \sum_i^n N_i u_i, \quad \epsilon = \sum_i^n B_i^u u_i \quad (6.130)$$

and

$$\phi = \sum_i^n N_i \phi_i, \quad \nabla \phi = \sum_i^n B_i^\phi \phi_i \quad (6.131)$$

where n is the number of nodes of the element and N_i is the element shape functions corresponding to the nodes. The B_i^u is the usual strain matrix and B_i^ϕ is the Cartesian derivative matrices, respectively, which are expressed as:

$$B_i^u = \begin{bmatrix} \frac{\partial N_i}{\partial x} & 0 \\ 0 & \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial y} & \frac{\partial N_i}{\partial x} \end{bmatrix} \quad \text{and} \quad B_i^\phi = \begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \end{bmatrix} \quad (6.132)$$

Requiring that $\delta W_{int} - \delta W_{ext} = 0$, the residual vector for stress equilibrium at the element level takes the form:

$$\begin{aligned} R_e^u = & \int_V \left[(1 - \phi)^2 + k \right] (B_i^u)^T \sigma dV - \int_V N^T b dV \\ & - \int_{\partial V} N^T h \partial V \end{aligned} \quad (6.133)$$

in which T represents the transpose.

Similarly, the resulting residual term for Eq. 6.129 is:

$$R_e^\phi = \int_V \left[G_c l_0 (B_i^\phi)^T \nabla \phi + \left(\frac{G_c}{l_0} + 2\psi(\epsilon) \right) N \phi - 2N\psi(\epsilon) \right] dV \quad (6.134)$$

The corresponding element stiffness at the element level are:

$$K_e = \begin{bmatrix} K_{ij}^{uu} & K_{ij}^{u\phi} \\ K_{ij}^{\phi u} & K_{ij}^{\phi\phi} \end{bmatrix} \quad (6.135)$$

and they are calculated as:

$$\begin{aligned} K_{ij}^{uu} &= \frac{\partial R_e^u}{\partial u} \\ &= \int_V [(1 - \phi)^2 + k] (B_i^u)^T C B_j^u dV \end{aligned} \quad (6.136)$$

$$K_{ij}^{u\phi} = \frac{\partial R_e^u}{\partial \phi} = \int_V -2(1 - \phi) (B_i^u)^T \sigma N_j dV \quad (6.137)$$

$$K_{ij}^{\phi u} = \frac{\partial R_e^\phi}{\partial u} = \int_V -2(1 - \phi) N_i \sigma B_j^u dV \quad (6.138)$$

$$\begin{aligned} K_{ij}^{\phi\phi} &= \frac{\partial R_e^\phi}{\partial \phi} \\ &= \int_V \left[G_c l_0 (B_i^\phi)^T B_j^\phi + \left(\frac{G_c}{l_0} + 2\psi(\epsilon) \right) N_i N_j \right] dV \end{aligned} \quad (6.139)$$

6.10.4 Numerical Implementation

Unfortunately, the system of equations given above does not guarantee the irreversible evolution of non-conserved phase-field parameter (i.e., $\phi_{t+\Delta t} < \phi_t$) which corresponds to healing of crack/damage. This drawback can be approximately circumvented by introducing an energy barrier or a penalty term as described in [9, 11].

Defining a function, in which

$$\langle x \rangle_- = \begin{cases} -x, & x < 0 \\ 0, & x \geq 0 \end{cases} \quad (6.140)$$

Then, a suitable penalty term can be constructed such that:

$$P(\dot{\phi}) = \frac{\eta}{n} \langle \dot{\phi} \rangle_- \quad (6.141)$$

in which n is positive integer which is taken as to be equal to two and $\dot{\phi} = \phi_{t+\Delta t} - \phi_t$. The value of η controls the magnitude of the penalty term and should be set a value that is large enough to enforce to prevent the irreversibility but not too large to cause ill-conditioned system of equations. Now, with the introduction of the penalty term, Eqs. 6.134 and 6.139 take the form:

$$R_e^\phi = \int_V \left[G_c l_0 (B_i^\phi)^T \nabla \phi + \left(\frac{G_c}{l_0} + 2\psi(\epsilon) \right) N \phi - 2N \left(\psi(\epsilon) - \frac{\eta}{n \Delta t} \dot{\phi}_- \right) \right] dV \quad (6.142)$$

and the corresponding stiffness term becomes:

$$K_{ij}^{\phi\phi} = \frac{\partial R_e^\phi}{\partial \phi} = \int_V \left[G_c l_0 (B_i^\phi)^T B_j^\phi + \left(\frac{G_c}{l_0} + 2\psi(\epsilon) \right) N_i N_j + \frac{\eta}{\Delta t} \dot{\phi}_- N_i N_j \right] dV \quad (6.143)$$

After forming element stiffness and residuals, they are assembled into the global stiffness and right-hand side vectors, and the resulting set of nonlinear equation.

$$[K^G] \{\delta\} = \{R^G\} \quad (6.144)$$

Note that the assembly process is carried out in such a away that the unknown vector δ in

Eq. 6.144 takes the form below. This is just a numerical convince for bookkeeping purposes.

$$\delta = (u_1^1, u_2^1, u_1^2, u_2^2, \dots, u_1^N, u_2^N, \phi^1, \phi^2, \dots, \phi^N)^T \quad (6.145)$$

where N is the total number of nodes in the FEM mesh, u_1 and u_2 are the components of nodal displacement, and ϕ is the value of the order parameter for the nodes.

As in previous case studies, the set of nonlinear equation are again solved with the modified Newton–Raphson scheme. Thus, the steps in the algorithm are:

Step-1

For each time increment:

Increment the values of prescribed displacements to increase the applied loading.

Form the global stiffness matrix, $[K^G]$, in Eq. 6.144

Step-2

Newton–Raphson iterations:

Form the right-hand side residual vector, $\{R^G\}$, in Eq. 6.144.

Solve Eq. 6.144.

Update nodal variables.

Check convergence, if it satisfied go to step-1, else repeat step-2.

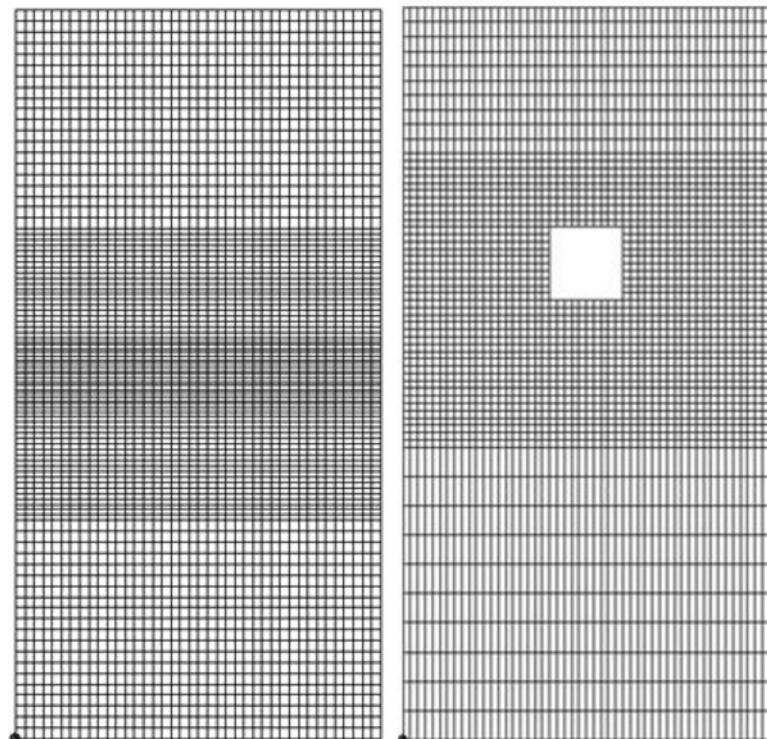
6.10.5 Results and Discussion

Two simulations were carried out by using the phase-field model developed in this section. The parameters used in the simulations that appear in the formulation are summarized in Table 6.2. The FEM meshes used in these simulations are shown in Fig. 6.16 and were generated with the mesh-generator program given in Appendix-C. The dimensions of the plates were 10 mm × 20 mm. The meshes were composed of four-node isoparametric elements. As it will be apparent, the initial cracks were introduced into the system by assigning order parameter value of one to certain element nodes in function *initilize.m*. There are 4141 nodes and 4000 elements in the mesh shown on the left; and 3030 nodes and 2900 elements on the right. As can be seen, the meshes are refined in the region in which the crack extension is expected to take place. This example also

Table 6.2 The values of the parameters used in the simulations

$G_c(\text{Nmm}^2/\text{mm})$	$E(\text{N/mm}^2)$	ν	L_0	k	N	η
0.001	20.0×10^5	0.3	0.125	1.0×10^{-6}	2	$2 \times E$

Fig. 6.16 FEM meshes used in the simulations



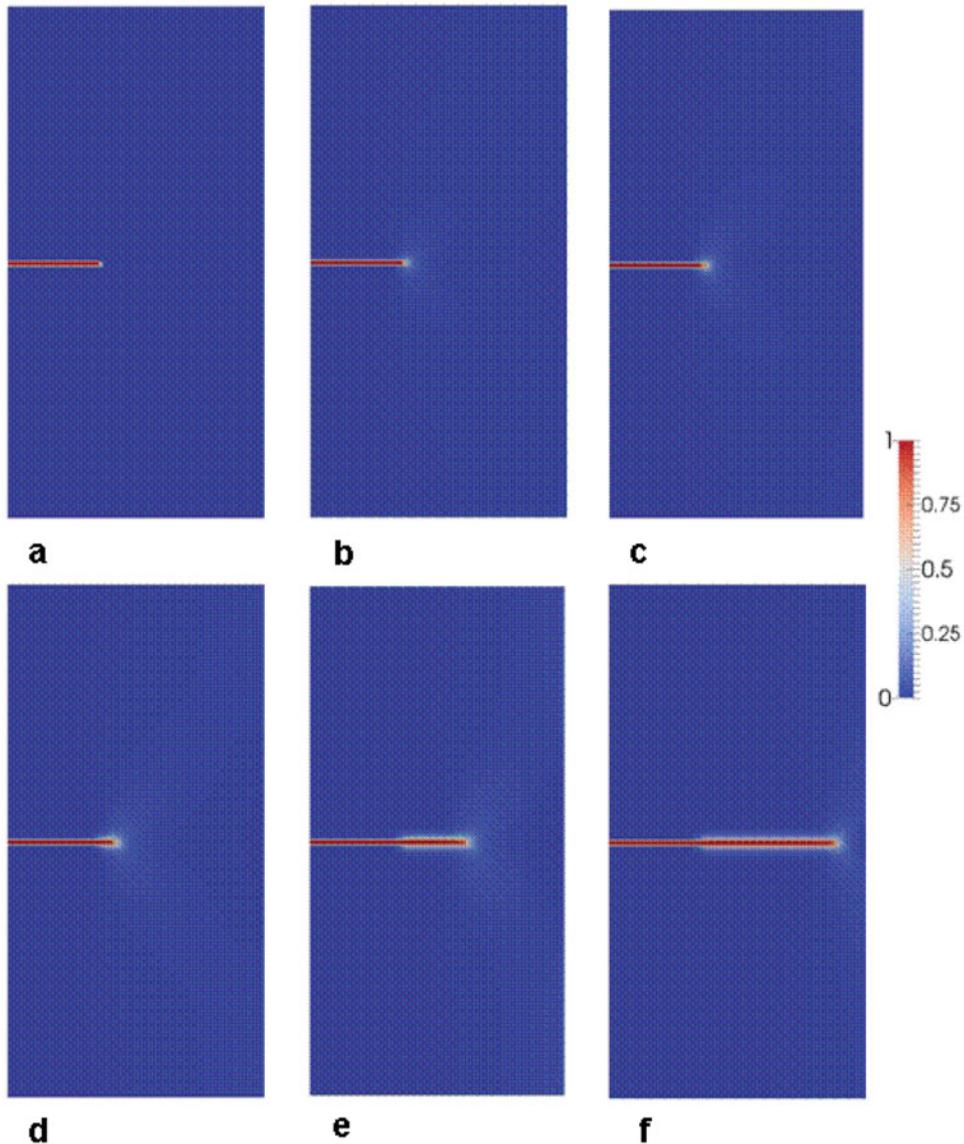


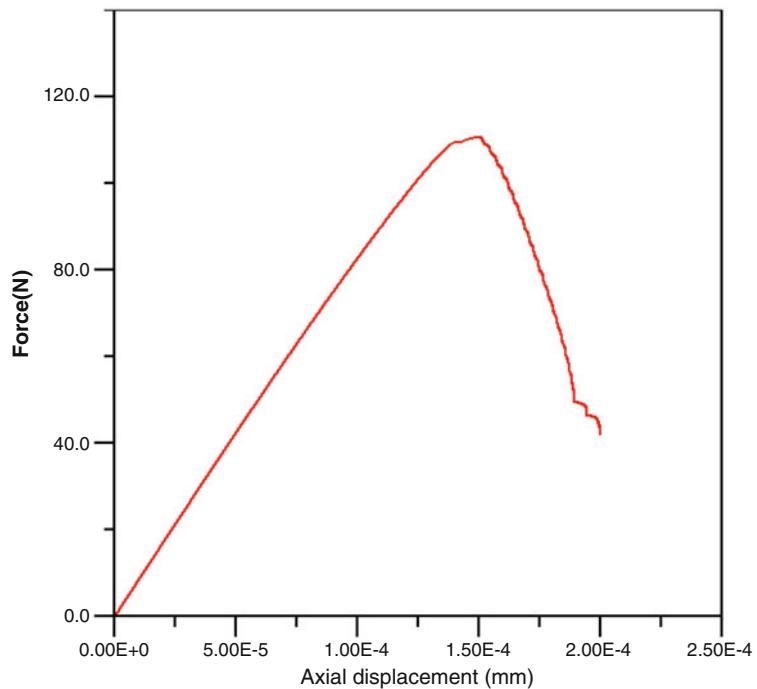
Fig. 6.17 Fracture process of a rectangular plate containing an edge-crack under uniaxial loading. The axial displacements, in mm, in the figure are (a) 1.25×10^{-5} , (b) 1.25×10^{-3} , (c) 1.375×10^{-3} , (d) 1.5×10^{-3} , (e) 1.6875×10^{-3} , and (f) 2.0×10^{-3}

shows one of the advantages of FEM in terms of its flexibility in spatial discretization. The nodes on the top and bottom edges have the prescribed displacements. While the axial displacements of the bottom nodes were constrained to be zero, the axial displacements of the nodes on the top edge were incremented 0.0005 mm per time step providing the applied loading to the system. The simulations were carried out under plane-strain

condition. Periodically, the resulting reaction forces at the bottom edge were integrated. They were output together with the current axial displacement value of the top edge, to generate overall force–displacement curves.

The time evolution of fracture process obtained from the mesh on the left is shown in Fig. 6.17 and the resulting force–displacement curve is given in Fig. 6.18. As can be seen from

Fig. 6.18 The resulting force–displacement curve during the fracture process of the plate with an initial edge crack under uniaxial loading



the figures, the initial response of system is linear. As damage starts to accumulate (i.e., increase in the order parameter values ahead of the crack tip), the force–displacement curve starts to deviate from the linearity. After reaching the critical energy release rate G_c the propagation of the initial crack takes place quite rapidly (Fig. 6.17). Even though there is still increase in the applied displacements, the system starts losing its load-bearing capacity and force–displacement curve exhibits a sharp drop.

In the second simulation, the applied loading rate and the simulation parameters were identical to that in the previous simulation. For this case, the evolution of fracture is summarized in Fig. 6.19 and the resulting force–displacement curve is given in Fig. 6.20. This example also demonstrates the flexibility of the FEM, in simulating complex geometries, over the two previous algorithms presented in Chaps. 4 and 5. Again, the system behavior is linear initially. With increasing load, the damage stars accumulating at the crack tip. This is apparent in contour maps as the color changes ahead of

the crack tip. With the satisfaction of the fracture criterion, the initial crack starts to propagate. However, soon after it interacts with the stress field of the hole and it becomes arrested temporarily. This can be observed very clearly in the movie files provided. It eventually resumes its propagation again, and kinks towards to hole and joins to hole, rather than moving straight ahead as in the first simulation. As can be seen from force–displacement curve in Fig. 6.20, although there is drop in the force displacement curve, however, this is not as severe as the first case. The system is still able to carry out the load and final axial displacement is almost 50 % more than the first case. Owing to the stress concentration effect at the corners of the hole, soon after, another crack nucleates around this region.

Even though very coarse mesh is used in these simulations, the results are in excellent agreement with the previous studies and experiments. The input files and the movie file resulting from these simulations are given in subdirectory *case_study_15* in downloadable file.

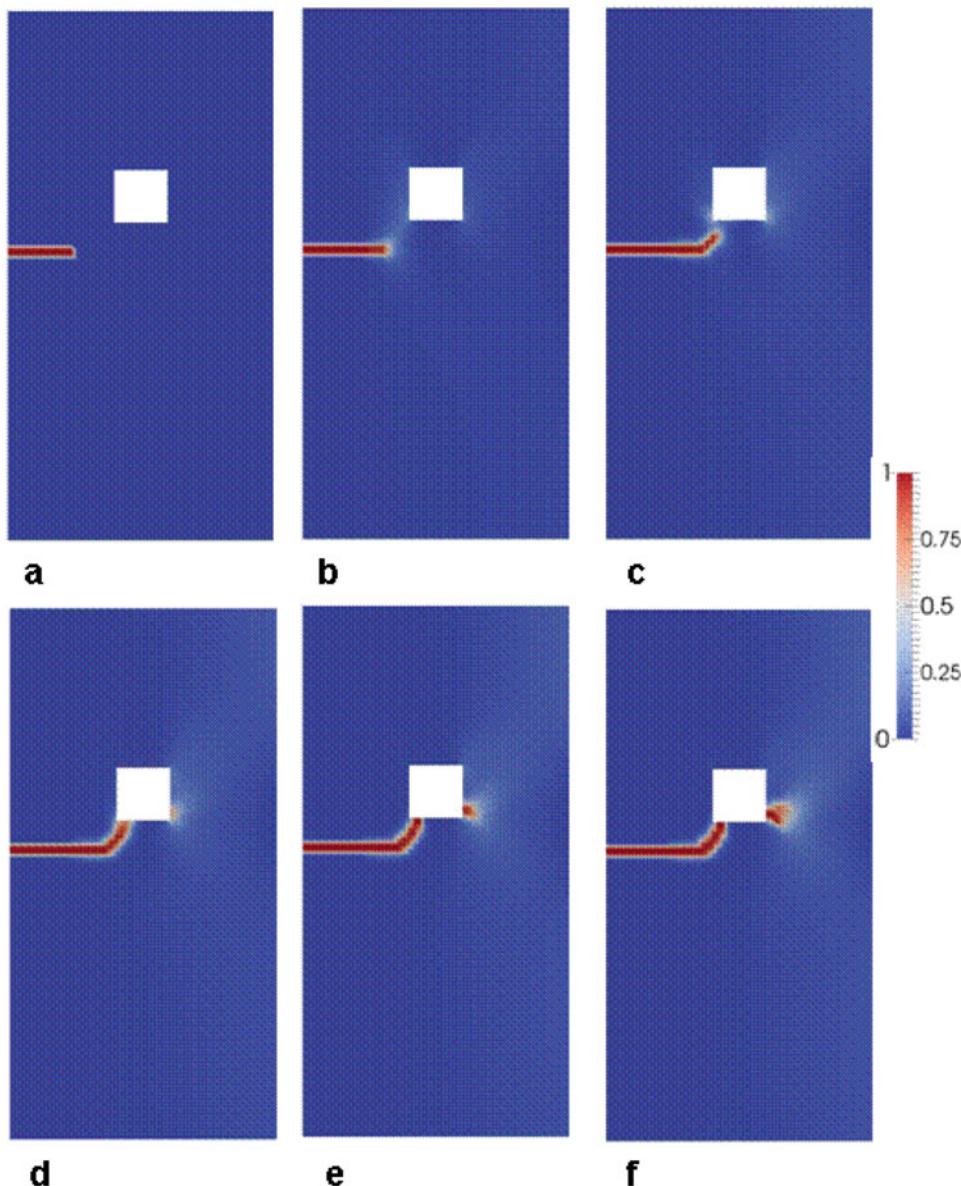


Fig. 6.19 Fracture process of a plate rectangular containing an edge-crack and a hole under uniaxial loading. The axial displacements, in mm, in the figure are
 (a) 1.25×10^{-5} , (b) 1.75×10^{-3} , (c) 2.25×10^{-3} ,
 (d) 2.5×10^{-3} , (e) 2.75×10^{-3} , and (f) 3.0×10^{-3}

6.10.6 Source Codes

Program

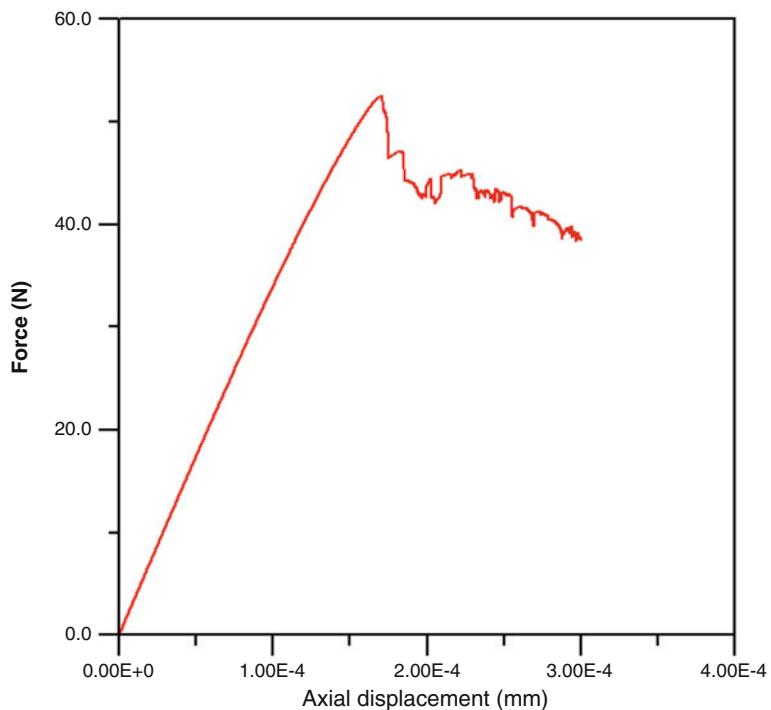
`fem_frac_v1_2.m`

This is the main program to solve Allen–Cahn equation with elastic inhomogeneities with FEM

algorithm. The algorithm solves the evolution of the non-conserved order parameter and the evolution of stress-strain fields simultaneously in fully coupled mode by taking into account the incremental external loading.

Depending on the selection of the `isolve` parameter in the code: For `isolve = 1`, the code executes in longhand format and un-optimized

Fig. 6.20 The resulting force–displacement curve during the fracture process of the plate with an initial edge crack and a hole under uniaxial loading



mode. For $\text{isolve} = 2$, execution is for Matlab/Octave optimized mode. Therefore, for the desired execution mode, this parameter in line 39 should be modified in the program.

The program makes calls to the following functions:

- **input_fem_elast.m**
- **initialize.m**
- **gauss.m**
- **cart_deriv.m**
- **fract_stiff_v1.m**
- **fract_stiff_v2.m**
- **boundary_cond2_v1.m**
- **boundary_cond2_v2.m**
- **stress_fract_v1.m**
- **stress_fract_v2.m**
- **residual_v1.m**
- **residual_v2.m**
- **write_vtk_fem.m**

Listing:

```

1 %%%%%%%%%%%%%%%%
2 %
3 % FEM PHASE-FIELD CODE FOR FRACTURE %
4 %
5 %%%%%%%%%%%%%%%%
6
7 % get initial wall time:
8 time0=clock();
9
10 icase=1;
11
12 global in;
13 if(icase==1)
14 in=fopen('fract_1ca.inp','r');
15 end
16
17 if(icase==2)
18 in=fopen('fract_1hca.inp','r');
19 end

```

```

20                               65
21 global out;                  66 if(isolve == 2)
22 out = fopen('result_1.out','w'); 67
23                                     68 [dgdx,dvolum] = cart_deriv(npoin,
24 global out2;                nelem,nnode,nstre,ndime,ndofn, ...
25 out2 = fopen('force-disp','w'); 69      ngaus,ntype,lnods,coord,
26                                     posgp,weigp);
27 %--- input time integration    70 end
28 parameters:                   71 %--- time integration
29 nstep= 4000;                 72 for istep = 1:nstep
30 nprnt= 25;                   73   tfacto = tfacto + dfacto;
31 dtme=1.0;                    74   tdisp_old = tdisp;
32 miter= 5;                     75   gforce = zeros(ntotv,1);
33 toler= 5.0e-3;               76   if(isolve == 1)
34 tfacto = 0.0;                 77     [gstif]= fract_stiff_v1(npoin,
35 dfacto = 0.0005;              nelem,nnode,nstre,ndime,ndofn, ...
36                                     ngaus,ntype,lnods,coord,
37                                     props,posgp, ...
38                                     weigp,dtme,constk,cenerg,
39 isolve =2;                      constl,constn, ...
40                                     coneta,stres,stran,tdisp,
41 % Material specific parameters: 78   tdisp_old);
42                                     tdisp;
43 constk = 1.0e-6;               79 end
44 cenerg = 0.001;               80
45 constl = 0.125;               81
46 constn = 2;                   82
47 coneta = 20.0e5*2;            83 if(isolve == 2)
48 %-----                         84   [gstif]= fract_stiff_v2(npoin,
49 %input data:                   nelem,nnode,nstre,ndime,ndofn, ...
50 %-----                         85   ngaus,ntype,lnods,coord,
51 %-----                         86   props,posgp, ...
52 [npoin,nelem,nvfix,ntype,nnode, 87   weigp,dgdx,dvolum,dtme,
53 ndofn,ndime,ngaus,...          constk,cenerg, ...
54 nstre,nmats,nprop,lnods,matno, 88   constl,constn,coneta,stres,
55 coord,props,nofix,...          stran,tdisp, ...
56 iffix,fixed]=input_fem_elast(); 89   tdisp_old);
57 ntotv = npoin*ndofn;           90 end
58 ndofn2 = 2;                   91
59 ntotv2 = npoin*ndofn2;         92
60                                     93
61 [tdisp,stres,stran] = initialize 94
62 (nelem,npoin,nnode,ngaus,ndofn,... 95
63      nstre,nvfix,isolve,icase); 96
64 [posgp,weigp] = gauss(ngaus,nnode); 97
                                     98
                                     99 %--- newton iteration:
100                                     100 for iter = 1:miter
101                                     101
102                                     102
103                                     103

```

```

104 %--- boundary conditions:
105
106 if(isolve == 1)
107
108 [gstif,gforce,treac]
109 =boundary_cond2_v1(npoin,nvfix,
110 nofix,ifix,fixed,ndofn, ...
111 tfacto,gstif,gforce,tdisp);
112 end
113 if(isolve == 2)
114 [gstif,gforce,treac]
115 =boundary_cond2_v2(npoin,nvfix,
116 nofix,ifix,fixed,ndofn, ...
117 tfacto,gstif,gforce,
118 tdisp);
119 end
120 %--- solve:
121 asdis = gstif\gforce;
122 %--- update:
123 tdisp = tdisp + asdis;
124 %--- adjust small deviations:
125 if(isolve == 1)
126 for ipoin=1:npoin
127 itotv = ntotv2 +ipoin;
128 if(tdisp(itotv) > 0.999)
129 tdisp(itotv) = 0.999;
130 end
131 if(tdisp(itotv) < 0.0)
132 tdisp(itotv) = 0.0;
133 end
134 end
135 end
136 end
137 end
138 end
139 end
140 %
141 if(isolve == 2)
142 dummy =tdisp(ntotv2+1:ntotv);
143 inrange = (dummy > 0.999);
144 dummy(inrange) =1.0;
145 inrange = ( dummy < 0.0);
146 dummy(inrange) = 0.0;
147
148 tdisp(ntotv2+1:ntotv) = dummy;
149 end
150
151 %---calculate stress & strain
152 increments:
153 if (isolve == 1)
154 [stran,stres] = stress_fract_v1
155 (asdis,nelem,npoin,nnodes,ngaus,
156 nstre,props, ...
157 ntype,ndofn,ndime,lnods,
158 matno,coord,posgp,weigp, ...
159 tdisp);
160 end
161 if(isolve == 2)
162 [stran,stres] = stress_fract_v2
163 (asdis,nelem,npoin,nnodes,ngaus,
164 nstre,props, ...
165 ntype,ndofn,ndime,lnods,
166 matno,coord,posgp,weigp, ...
167 dgdx,dvolum,tdisp);
168 end
169
170 %--- check norm for convergence:
171 if(normF <= toler)
172 break;
173 end
174 %--- calculate residual force
175 vector:
176 if(isolve == 1)
177 [gforce]=residual_v1(npoin,nelem,
178 nnodes,nstre,ndime,ndofn, ...
179 ngaus,ntype,lnods,coord,
180 props,posgp, ...
181 weigp,dtime,constk,
182 cenerg,constl,constn,
183 coneta,stres,stran,
184 tdisp,tdisp_old);
```

```

185
186 [gforce]=residual_v2(npoin,nelem,
187 nnodes,nstre,ndime,ndofn, ...
188 ngaus,ntype,lnods,coord,
189 props,posgp, ...
190 weigp,dgdx,dvolum,dtime,
191 constk,cenerg, ...
192 constl,constn,coneta,
193 stres,stran,tdisp, ...
194 tdisp_old);
195 end
196
197 end %end of Newton
198
199 %--- print data for force-disp
200 curves:
201 lnode=nofix(nvfix);
202 nvfix2=nvfix/2;
203 sumr=0.0;
204 for ivfix=1:nvfix2
205 sumr=sumr+treac(ivfix,2);
206 fprintf(out2,'%14.6e %14.6e\n',
207 tdisp((lnode-1)*2+2),sumr);
208 %--- print results:
209 if(mod(istep,nprnt) == 0 )
210 fprintf('Done step: %5d\n',istep);
211
212 %fname=sprintf('time_%d.out',
213 istep);
214 %out=fopen(fname,'w');
215 %ntotv2=npoin*ndofn2;
216 %for ipoin=1:npoin
217 %itotv=ntotv2+ipoin;
218 %fprintf(out,'%14.6e %14.6e
219 %14.6e\n',coord(ipoin,1),
220 coord(ipoin,2),tdisp(itotv));
221 %end
222 %fclose(out);
223 %--- write to vtk file with updated
224 mesh
225 for ipoin=1:npoin
226 for idofn=1:ndofn2
227 itotv=(ipoin-1)*ndofn2+idofn;
228 cord2(ipoin,idofn)=coord(ipoin,
229 idofn)+10.0*tdisp(itotv);
230 jtotv=ntotv2+ipoin;
231 cont1(ipoin)=tdisp(jtotv);
232 end
233 end
234 write_vtk_fem(npoin,nelem,nnodes,
235 lnods,cord2,istep,cont1);
236 end %if
237 end %istep
238
239 compute_time=etime(clock(),
240 time0)
241
242 fprintf(out,'compute time: %7d\n',
243 compute_time);

```

Line numbers:

8:	Get wall clock time beginning of the execution.
10:	icase = 1 single crack simulation, icase = 2 crack-hole simulation (see results and discussion).
12–25:	Depending on the value of icase, assign unit names for input and output files.
27–38:	Time integration parameters.
29:	Number of time steps.
30:	Print frequency to output the results to file.
31:	Time increment for numerical integration.
33:	Maximum number of Newton–Raphson iteration.
34:	Tolerance value for iterative solution.
36:	Total increment factor for displacement increments.
37:	Displacement increment per time steps.
39:	Solution flag; isolve = 1 for un-optimized solution, isolve = 2 for optimized solution.
41–47:	Material-specific parameters.
43:	Parameter to avoid overflow for cracked elements.
44:	Critical strain energy for fracture (Fracture toughness values).
45:	The value of l_0 , Eq. 6.124
46–47:	The value of penalty parameters, n and η , Eqs. 6.141, 6.142 and 6.143.

(continued)

53–55:	Input FEM mesh and control parameters.	199–203:	Sum the reaction force vector for bottom boundary nodes.
57:	Total number of variables in the solution.	206:	Print the results to force-disp output file.
58:	The number of DOFs for displacements per node.	210–236:	If print frequency is reached, print the results to file.
59:	Total number of variables for displacements.	214–222:	Open an output file and print nodal coordinates and nodal values of order parameter to file. These lines are commented out, but they can be changed, including the output format.
61–62:	Initialize crack geometry in the FEM mesh, depending on the value of icase.	225–232:	Add current displacement values to the nodal coordinates and collect the order parameter values from solution vector into cont1 array.
64:	Parameters for numerical integration.	234:	Write the results in vtk file format for contour plots to be viewed by using Paraview.
66–70:	If solution is in optimized mode (isolve = 2), precalculate the Cartesian derivatives of shape functions for all elements in the solution.	240–242:	Calculate the total execution time and print it.
72–238:	Time evolution.		
76:	Increase the total displacement factor.		
78:	Transfer solution vector values from previous time step to tdisp_old vector.		
79:	Initialize global rhs, load, vector.		
81–88:	If solution is in un-optimized mode (isolve = 1), form global stiffness matrix.		
90–98:	If solution is in optimized mode (isolve = 2) form global stiffness matrix.		
100–193:	Newton–Raphson iterative solution.		
106–110:	If solution is in un-optimized mode (isolve = 1), modify global stiffness matrix and global rhs, load, vector for boundary conditions.		
112–116:	If solution is in optimized mode (isolve = 2), modify global stiffness matrix and global rhs, load, vector for boundary conditions.		
120:	Solve system equations.		
124:	Update nodal solution vector.		
126–149:	If there are small deviations from max and min values, reset the limits for order parameter.		
128–137:	For un-optimized solution mode.		
141–149:	For optimized solution mode.		
153–158:	If solution is in un-optimized mode, calculate stress and strain values.		
160–164:	If solution is in optimized mode (isolve = 2, calculate stress and strain values.		
166–172:	Check convergence, if convergence is reached exit from Newton–Raphson iteration loop.		
176–182:	If solution is in un-optimized mode (isolve = 1), calculate global rhs, load, vector.		
184–191:	If solution is in optimized mode (isolve = 2), calculate global rhs, load, vector.		
195–207:	Calculate total reaction forces for the bottom boundary for force–displacement curve.		
197:	Last node number in the boundary nodes array nofix.		

Function

fract_stiff_v1.m

This function calculates the global stiffness matrix. It is in longhand format and is not optimized.

The function makes calls to the following functions:

- **sfr2.m**
- **jacob2.m**
- **modps.m**
- **bmats.m**
- **dbe.m**

Variable and array list:

npoint:	Number of nodes.
nelem:	Number of elements.
nnode:	Number of nodes per element.
nstre:	Number of stress components.
ndime:	Number of Cartesian coordinate dimension.
ndofn:	Number of DOF per node.
ngaus:	Order of numerical integration.
ntype:	Solution type, ntype = 1 for plane-stress, ntype = 2 for plane-strain.
dtime:	Time increment for numerical integration.
constk:	Small value to avoid overflow for cracked elements.
cenerg:	Critical strain energy for fracture.

(continued)

constl:	Interface control parameter.
constn:	Power term for the penalty parameter.
coneta:	Magnitude of the penalty term.
tdisp(ntotv):	Solution vector, ntotv = npoin × ndofn.
tdisp_old (ntotv):	Nodal values from previous time step.
posgp(ngaus):	Position of integration points.
weigp(ngaus):	Weights for numerical integration.
props(nmat, nprop):	Material properties, nmat number of different materials in the FEM mesh and nprop, number of properties.
lnods(nelem, nnodes):	Element nodal connectivity list.
coord(npoin, ndime:	Cartesian coordinates of the nodes.
gstif(ntotv, ntotv):	Globals stiffness matrix.
stres(nelem, mgaus,nstre):	Stress values at the integration points of the elements.
stran(nelem, mgaus,nstre):	Strain values at the integration point of the elements

Listing:

```

1 function [gstif]=fract_stiff_v1
2     (npoin,nelem,nnodes,nstre,ndime,
3      ndofn, ...
4      ngaus,ntype,lnods,coord,
5      props,posgp, ...
6      weigp,dtime,constk,cenerg,
7      constl,constn, ...
8      coneta,stres,stran,tdisp,
9      tdisp_old)
10
11 format long;
12
13 %--- order of integration
14
15 ngaus2=ngaus;
16
17 if(nnodes == 3)
18     ngaus2=1;
19 end
20
21
22
23
24
25
26
27 %---global stiffness:
28
29 gstif = sparse(ntotv,ntotv);
30
31 for ielem=1:nelem
32
33 %--initialize element stiffnesses &
34 %& rhs:
35
36 for ievab=1:nevab
37     eload(ievab) = 0.0;
38     for jevab=1:nevab
39         estif(ievab,jevab) = 0.0;
40     end
41
42 for ievab=1:nevab2
43     eload1(ievab) = 0.0;
44     for jevab=1:nevab2
45         estif1(ievab,jevab) = 0.0;
46     end
47 end
48
49 for ievab=1:nevab2
50     for jevab=1:nevab3
51         estif2(ievab,jevab) = 0.0;
52         estif3(jevab,ievab) = 0.0;
53     end
54 end
55
56 for ievab=1:nevab3
57     eload2(ievab) = 0.0;
58     for jevab=1:nevab3
59         estif4(ievab,jevab) = 0.0;;
60     end
61 end
62
63 %---
64 %--- element nodal values
65 %--
66
67 for inode=1:nnodes
68     lnode = lnods(ielem,inode);
69     itotv = ntotv2 +lnode;

```

```

70 ephi(inode) = tdisp(itotv);
71 ephir(inode) = tdisp(itotv)
-tdisp_old(itotv);
72 end
73
74 %--- Coordinates of element nodes:
75
76 for inode=1:nnode
77 lnode=lnods(ielem,inode);
78 for idime=1:ndime
79 elcod(idime,inode)=coord(lnode,
idime);
80 end
81 end
82
83 %--- integrate element stiffness
84
85 kgasp=0;
86 for igaus=1:ngaus
87 exisp=posgp(igaus);
88 for jgaus=1:ngaus2
89 etasp =posgp(jgaus);
90 if(nnode ==3)
91 etasp=posgp(ngaus+igaus);
92 end
93
94 kgasp=kgasp+1;
95 [shape,deriv]=sfr2(exisp,etasp,
nnode);
96
97 [cartd,djacb,gpcod]=jacob2(ielem,
elcod,kgasp,shape,deriv,nnode,
ndime);
98
99 dvolu=djacb*weigp(igaus)
*weigp(jgaus);
100 if(nnode == 3)
101 dvolu=djacb*weigp(igaus);
102 end
103
104 %-- values at the integration points:
105 phigp=0.0;
106 phirgp = 0.0;
107
108 for inode=1:nnode
109 phigp = phigp + ephi(inode)*shape
(inode);
110
111 phigp = phigp + ephir(inode)*shape
(inode);
112 phirgp = phirgp + ephir(inode) * shape
(inode);
113 end
114
115
116 mtype = 1;
117 [dmatx] = modps(mtype,ntype,nstre,
props);
118
119 %---
120
121 [bmatx]=bmats(cartd,shape,inode);
122
123 [dbmat] =dbe(nevab2,nstre,bmatx,
dmatx);
124
125 %element stiffness:
126
127 %--- estif1:
128
129 for ievab=1:nevab2
130 for jevab=1:nevab2
131 for istre=1:nstre
132
133 estif1(ievab,jevab)=estif1(ievab,
jevab)+((1.0-phigp)^2+constk)* ...
134 bmatx(istre,ievab)*dbmat(istre,
jevab)*dvolu;
135
136 end
137 end
138 end
139
140 %---estif2:
141
142 for ievab=1:nevab2
143 dummy(ievab) =0.0;
144 end
145
146 for istre =1:nstre
147 for inode =1:nnode
148 dummy(istre,inode) =stres(ielem,
kgasp,istre)*shape(inode);
149 end
150 end
151
152 for ievab=1:nevab2
153 for jevab=1:nevab3
154 for istre=1:nstre

```

```

155
156 estif2(ievab,jevab) = estif2(ievab,
157   jevab)-2.0*(1.0-phirgp)*bmatx(istre,
158   ievab)*...
159   dummy(istre,jevab)*dvolu;
160 end
161 end
162
163 %--- estif4
164
165 for ievab=1:nevab3
166 for jevab=1:nevab3
167 for istre=1:ndime
168   estif4(ievab,jevab) = estif4(ievab,
169     jevab) + cenerg*constl*cartd(istre,
170     ievab)*...
171   cartd(istre,jevab)*dvolu;
172 end
173
174 %--- strain energ
175 senerg = 0.0;
176 for istre=1:nstre
177   senerg = senerg + 0.5*stres(ielem,
178     kgasp,istre) * stran(ielem,kgasp,
179     istre);
180 end
181
182 for inode=1:nnode
183 for jnode=1:nnode
184   estif4(inode,jnode) = estif4(inode,
185     jnode)+((cenerg/constl) +
186     2.0*senerg)*...
187   shape(inode)*shape(jnode)*dvolu;
188 end
189
190 %--- penalty term:
191
192 constx = 0.0;
193 if(phirgp < 0.0 )
194   constx = -phirgp;
195 end
196
197 for inode=1:nnode
198 for jnode=1:nnode
199   estif4(inode,jnode)=estif4(inode,
200     jnode)+(coneta/dtime)*constx^
201     (constn-1)*...
202   shape(inode)*shape(jnode)*dvolu;
203 end
204 end%jgaus
205 end%igaus
206
207 %--- global stiffness matrix:
208
209 %--- assemble estif1:
210
211 for inode =1:nnode
212 lnode = lnods(ielem,inode);
213 for idofn =1:ndofn2
214 ievab=(inode-1)*ndofn2+idofn;
215 itotv=(lnode-1)*ndofn2+idofn;
216 %
217 for jnode =1:nnode
218 knode = lnods(ielem,jnode);
219 for jdofn =1:ndofn2
220 jevab=(jnode-1)*ndofn2+jdofn;
221 jtovt=(knode-1)*ndofn2+jdofn;
222
223 gstif=gstif +sparse(itotv,jtovt,
224   estif1(ievab,jevab),ntotv,ntotv);
225 end
226 end
227 end
228 end
229
230 % assemble estif2 :
231
232 for inode =1:nnode
233 lnode = lnods(ielem,inode);
234 for idofn =1:ndofn2
235 ievab=(inode-1)*ndofn2+idofn;
236 itotv=(lnode-1)*ndofn2+idofn;
237 %
238 for jnode =1:nnode
239 knode = lnods(ielem,jnode);
240 for jdofn =1:ndofn3
241 jevab=(jnode-1)*ndofn3+jdofn;
242 jtovt=(knode-1)*ndofn3+jdofn
243 +ntotv2;

```

```

243
244 g stif = g stif + sparse(itotv,jtotv,
245   estif2(ievab,jevab),ntotv,ntotv);
246 end
247 end
248 end
249 end
250
251 % assemble estif 3 as transpose
252   of estif2:
253 for inode=1:nnode
254 lnode = lnods(ielem,inode);
255 for idofn=1:ndofn3
256 ievab=(inode-1)*ndofn3+idofn;
257 itotv=(lnode-1)*ndofn3+idofn+
258 ntotv2;
259 %
260 for jnode=1:nnode
261 knode = lnods(ielem,jnode);
262 jdofn=1:ndofn2
263 jevab=(jnode-1)*ndofn2+jdofn;
264 jtotv=(knode-1)*ndofn2+jdofn;
265 g stif = g stif + sparse(itotv,jtotv,
266   estif2(jevab,ievab),ntotv,ntotv);
267 end
268 end
269 end
270 end
271
272 % assemble estif4:
273
274 for inode=1:nnode
275 lnode = lnods(ielem,inode);
276 for idofn=1:ndofn3
277 ievab=(inode-1)*ndofn3+idofn;
278 itotv=(lnode-1)*ndofn3+idofn+
279 ntotv2;
280 %
281 for jnode=1:nnode
282 knode = lnods(ielem,jnode);
283 jdofn=1:ndofn3
284 jevab=(jnode-1)*ndofn3+jdofn;
285 jtotv=(knode-1)*ndofn3+jdofn+
286 ntotv2;
287
288 estif4(ievab,jevab),ntotv,ntotv);
289 end
290 end
291 end
292
293 end %ielem
294
295 end %endfunction

```

Line numbers:

10–13:	Order of numerical integration, depending on the element type.
15:	Total number of integration points per element.
19:	Total number of variables.
20:	Total number of variables per element.
21:	Number of DOFs for displacements per node.
22:	Number of DOFs for order parameters per node.
23:	Total number of variables for displacements.
24:	Total number of displacements per element.
25:	Total number of variables for order parameters per element.
29:	Initialize global stiffness matrix.
31–293:	Loop over elements.
33–62:	Initialize element stiffness matrix, submatrices and rhs, load, vectors.
64–73:	Get elemental nodal order parameter values.
74–82:	Get elemental nodal Cartesian coordinates.
83–205:	Integrate element stiffness matrix.
87,	Coordinates of the integration points in local coordinate system.
89, 91:	
94:	Increment integration point counter.
95:	Calculate shape functions, their derivative values.
97:	Calculate Cartesian derivatives of shape functions.
99–103:	Calculate element area/volume.
105–114:	Calculate the value of the order parameters at the current integration point, Eq. 6.131.
117:	Calculate the elasticity matrix.
121:	Calculate the strain matrix.
123:	Multiply strain matrix with elasticity matrix.
129–138:	Form submatrix, K_{ij}^{uu} , Eq. 6.136.
142–161:	Form submatrix, $K_{ij}^{u\phi}$, Eq. 6.137.
163–202:	Form submatrix, $K_{ij}^{\phi\phi}$, Eq. 6.143.
207–292:	Assemble element stiffness submatrices directly into global stiffness matrix.

Function**fract_stiff_v2.m**

This function calculates the global stiffness matrix. It is optimized for Matlab/Octave.

The function makes calls to the following functions:

- **sfr2.m**
- **modps.m**
- **bmats1.m**
- **bmats2.m**
- **dbe2.m**

Variable and array list:

npoin:	Number of nodes
nelem:	Number of elements.
nnode:	Number of nodes per element.
nstre:	Number of stress components.
ndime:	Number of Cartesian component dimension.
ndofn:	Number of DOF per node.
ngaus:	Order of numerical integration.
ntype:	Solution type, ntype = 1 for plane-stress, ntype = 2 for plane-strain.
dtime:	Time increment for numerical integration.
constk:	Small value to avoid overflow for cracked elements.
cenerg:	Critical strain energy for fracture.
constl:	Interface control parameter.
constn:	Power term for the penalty parameter.
coneta:	Magnitude of the penalty term.
tdisp(ntotv):	Solution vector, ntotv = npoin × ndofn.
tdisp_old(ntotv):	Nodal values from previous time step.
posgp(ngaus):	Position of integration points.
weigp(ngaus):	Weights for numerical integration.
props(nmat, nprop):	Material properties, nmat number of different materials in the FEM mesh and nprop, number of properties.
lnods(nelem, nnode):	Element nodal connectivity list.
coord(npoin, ndime):	Cartesian coordinates of the nodes.
gstif(ntotv,ntotv):	Global stiffness matrix.

dvolum(nelem, mgaus):	Area/volume of elements.
stres(nelem,mgaus, nstre):	Stress values at the integration points of the elements.
stran(nelem, mgaus,nstre):	Strain values at the integration point of the elements
dgdx(nelem, mgaus,nstre, nnode):	Precalculated Cartesian derivatives of shape functions for elements.

Listing:

```

1  function [gstif]=fract_stiff_v2
2    (npoin,nelem,nnode,nstre,ndime,
3     ndofn, ...
4      ngaus,ntype,lnods,coord,
5      props,posgp, ...
6      weigp,dgdx,dvolum,dtime,
7      constk,cenerg, ...
8      constl,constn,coneta,
9      stres,stran,tdisp, ...
10     tdisp_old)
11
12  format long;
13
14  %--- order of integration
15
16  ngaus2=ngaus;
17
18  if(nnode == 3)
19    ngaus2=1;
20
21  end
22
23  ngaus = ngaus*ngaus2;
24
25  %--initialize global and local
26  %stiffness:
27
28  ntotv = npoin*ndofn;
29
30  nevab = nnode*ndofn;
31
32  ndofn2 = ndofn-1;
33
34  ndofn3 = ndofn-2;
35
36  ntotv2 = npoin*ndofn2;
37
38  nevab2 = nnode*ndofn2;
39
40  nevab3 = nnode*ndofn3;
41
42  %---global stiffness:
43
44  gstif = sparse(ntotv,ntotv);
45
46  %--- element stiffnesses

```

```

33
34 estif = zeros(nelem,nevab,nevab);
35 estif1 = zeros(nelem,nevab2,
36 nevab2);
36 estif2 = zeros(nelem,nevab2,
37 nevab3);
37 estif3 = zeros(nelem,nevab3,
38 nevab2);
38 estif4 = zeros(nelem,nevab3,
39 nevab3);

40 eload = zeros(nelem,nevab);
41 eload1 = zeros(nelem,nevab2);
42 eload2 = zeros(nelem,nevab3);
43
44 %---
45 %--- element nodal values
46 %--
47
48 ephi = zeros(nelem,nnode);
49 ephir = zeros(nelem,nnode);
50
51 for inode =1:nnode
52 lnode = lnods( :,inode);
53 itotv = ntotv2 +lnode;
54 ephi( :,inode) = tdisp(itotv);
55 ephir( :,inode) = tdisp(itotv)-
56 tdisp_old(itotv);
56 end
57
58 %--- integrate element stiffness
59
60 kgasp=0;
61 for igaus=1:ngaus
62 exisp=posgp(igaus);
63 for jgaus=1:ngaus2
64 etasp =posgp(jgaus);
65 if(nnode ==3)
66 etasp=posgp(ngaus+igaus);
67 end
68
69 kgasp=kgasp+1;
70 [shape,deriv]=sfr2(exisp,etasp,
71 nnode);
72 %-- values at the integration points:
73
74 phigp=zeros(nelem,1);
75 phirgp =zeros(nelem,1);
76

77 for inode=1:nnode
78 phigp =phigp + ephi( :,inode)
79 *shape(inode);
79 phirgp =phirgp + ephir( :, inode) *
80 shape(inode);
80 end
81
82 mtype =1;
83 [dmatx] =modps(mtype,ntype,nstre,
84 props);
84
85 %--- cartesien derivative matrices;
86
87 [bmatx1]=bmats1(dgdx,nelem,nnode,
88 nstre,nevab2,kgasp);
88 [bmatx2]=bmats2(dgdx,nelem,nnode,
89 nstre,nevab2,kgasp);
89
90 [dbmat] =dbe2(nelem,nevab2,nstre,
91 bmatx2,dmatx);
91
92 %element stiffness:
93
94 %--- estif1:
95
96 for ievab=1:nevab2
97 for jevab=1:nevab2
98 for istre=1:nstre
99
100 estif1( :,ievab,jevab)=estif1
101 ( :,ievab,jevab)+((1.0-phigp).^2
102 +constk).*...
103 bmatx2( :,istre,ievab).
104 *dbmat( :,istre,jevab).
105 *dvolum(:,kgasp);
106
107 %---estif2:
108
109 dummy = zeros(nelem,nevab2);
110
111 for istre=1:nstre
112 for inode=1:nnode
113 dummy( :,istre,inode) =stres( :,
114 kgasp,istre)*shape(inode);
114 end
115 end

```

```

116                                         154
117 for ievab=1:nevab2                  155 %--- penalty term:
118 for jevab=1:nevab3
119 for istre=1:nstre
120
121 estif2( :,ievab,jevab) = estif2
122   ( :,ievab,jevab)-2.0*(1.0-phirgp).
123   *bmatx2( :,istre,ievab).*...
124   dummy( :,istre,jevab).*dvolum
125   ( :,kgasp);
126
127
128 %--- estif4
129
130 for ievab=1:nevab3
131 for jevab=1:nevab3
132 for istre=1:ndime
133 estif4( :,ievab,jevab) = estif4( :,
134   ievab,jevab) +
135   cenerg*constl*bmatx1( :,istre,
136   ievab).*...
137   bmatx1( :,istre,jevab).*dvolum
138   ( :,kgasp);
139
140 %--- strain energ
141 senerg=zeros(nelem,1);
142 for istre=1:nstre
143 senerg = senerg + 0.5*stres
144   ( :,kgasp,istre). * stran
145   ( :,kgasp,istre);
146 for inode=1:nnode
147 for jnode=1:nnode
148 estif4( :,inode,jnode) = estif4
149   ( :,inode,jnode)+((cenerg/constl)
150   + 2.0*senerg).*...
151   shape(inode)*shape(jnode).
152   *dvolum( :,kgasp);
153
154
155 %--- penalty term:
156
157 constx = zeros(nelem,1);
158 inrange = (phirgp < 0 );
159 constx(inrange) = -phirgp
160   (inrange);
161
162 for inode=1:nnode
163 for jnode=1:nnode
164 estif4( :,inode,jnode) = estif4
165   ( :,inode,jnode)+(coneta/dtime)
166   *constx.^((constn-1)*...
167   shape(inode)*shape(jnode).
168   *dvolum( :,kgasp));
169
170 end %jgaus
171
172 %--- global stiffness matrix
173
174 %--- assemble estif1:
175
176 for inode =1:nnode
177 lnode = lnods( :,inode);
178 for idofn =1:ndofn2
179 ievab =(inode-1)*ndofn2+idofn;
180 itotv =(lnode-1)*ndofn2+idofn;
181 %
182 for jnode =1:nnode
183 knode = lnods( :,jnode);
184 for jdofn =1:ndofn2
185 jevab =(jnode-1)*ndofn2+jdofn;
186 jtovt =(knode-1)*ndofn2+jdofn;
187
188 gstif = gstif +sparse(itotv,jtotv,
189   estif1( :,ievab,jevab),ntotv,
190   ntotv);
191
192 end
193
194
195 % assemble estif2 :
196
197 for inode =1:nnode

```

```

198 lnode = lnods( :, inode );
199 for idofn =1:ndofn2
200 ievab =(inode-1)*ndofn2+idofn;
201 itotv =(lnode-1)*ndofn2+idofn;
202 %
203 for jnode =1:nnode
204 knode = lnods( :, jnode );
205 for jdofn =1:ndofn3
206 jevab =(jnode-1)*ndofn3+jdofn;
207 jtotv =(knode-1)*ndofn3+jdofn
+ntotv2;
208
209 gstif =gstif +sparse(itotv,jtotv,
estif2( :, ievab,jevab ),ntotv,
ntotv );
210
211 end
212 end
213 end
214 end
215
216 % assemble estif 3 as transpose
of estif2:
217
218 for inode =1:nnode
219 lnode = lnods( :, inode );
220 for idofn =1:ndofn3
221 ievab =(inode-1)*ndofn3+idofn;
222 itotv =(lnode-1)*ndofn3+idofn
+ntotv2;
223 %
224 for jnode =1:nnode
225 knode = lnods( :, jnode );
226 for jdofn =1:ndofn2
227 jevab =(jnode-1)*ndofn2+jdofn;
228 jtotv =(knode-1)*ndofn2+jdofn;
229
230 gstif =gstif +sparse(itotv,jtotv,
estif2( :, jevab,ievab ),ntotv,
ntotv );
231
232 end
233 end
234 end
235 end
236
237 % assemble estif4:
238
239 for inode =1:nnode
240 lnode = lnods( :, inode );
241 for idofn =1:ndofn3
242 ievab =(inode-1)*ndofn3+idofn;
243 itotv =(lnode-1)*ndofn3+idofn
+ntotv2;
244 %
245 for jnode =1:nnode
246 knode = lnods( :, jnode );
247 for jdofn =1:ndofn3
248 jevab =(jnode-1)*ndofn3+jdofn;
249 jtotv =(knode-1)*ndofn3+jdofn
+ntotv2;
250
251 gstif =gstif +sparse(itotv,jtotv,
estif4( :, ievab,jevab ),ntotv,
ntotv );
252
253 end
254 end
255 end
256 end
257
258 end %endfunction

```

Line numbers:

11–14:	Order of numerical integration.
16:	Total number of integration points per element.
20:	Total number of variables.
21:	Total number of variables per element.
22:	Number of DOFs for displacements per node.
23:	Number of DOFs for order parameters per node.
24:	Total number of variables for displacements.
25:	Total number of displacements per element.
26:	Total number of variables for order parameters per element.
30:	Initialize global stiffness matrix.
32–42:	Initialize stiffness matrix submatrices and rhs, load, vectors for all elements.
48–49:	Initialize the arrays holding elemental nodal order parameters values.
51–56:	Get the elemental nodal values of order parameters for all elements.
58–170:	Numerical integration of elements.
62, 64, 66:	Coordinates of integration points in local coordinate system.
69:	Increment integration point counter.
70:	Calculate shape functions and their derivatives values.
72–80:	Values of order parameters at the current integration point, Eq. 6.131, for all elements.

(continued)

83:	Calculate the elasticity matrix.
87:	Calculate the Cartesian derivatives of shape functions of all elements.
88:	Form strain matrix for all elements.
90:	Multiply elasticity matrix with strain matrix for all elements.
96–105:	Form submatrix, K_{ij}^{uu} , Eq. 6.136, for all elements.
107–127:	Form submatrix, $K_{ij}^{u\phi}$, Eq. 6.137, for all elements.
128–168:	Form submatrix, $K_{ij}^{\phi\phi}$, Eq. 6.143, for all elements.
172–257:	Assemble element stiffness submatrices directly into global stiffness matrix, for all elements.

Function

initialize.m

Depending on the value of `icase`, this function introduces the initial crack configurations into FEM mesh. Also initializes solution vector and arrays holding elemental stress and strain values.

Variable and array list:

nelem:	Number of elements.
npoin:	Number of nodes.
ngaus:	Order of numerical integration.
ndofn:	Number of DOFs per node.
nstre:	Number of stress components.
nvfix:	Number of nodes having prescribed displacements.
isolve:	Solution flag, <code>isolve = 1</code> for un-optimized solution mode, <code>isolve = 2</code> for optimized mode.
icase:	<code>icase = 1</code> , for crack simulation. <code>icase = 2</code> , for crack-hole simulation (see results and discussion).
tdisp(ntotv):	Solution vector.
stres(nelem, mgaus,nstre):	Stress component values at the integration points of all elements.
stran(nelem, mgaus,nstre):	Strain components values at the integration points of all elements.

Listing:

```

1   function[tdisp,stres,stran]
2       =initialize(nelem,npoin,nnode,
3           ngaus,ndofn,...
4           nstre,nvfix,isolve,icase)

```

```

4   format long;
5
6   ndofn2 = ndofn-1;
7   ntotv2 = npoin*ndofn2;
8   ntotv = npoin*ndofn;
9
10  ngaus2 = ngaus;
11  if(nnode == 3)
12      ngaus2=1;
13  end
14  mgaus = ngaus*ngaus2;
15
16  %-- initialize stress and strain:
17
18  if(isolve == 1)
19
20  for ielem=1:nelem
21  for igaus=1:mgaus
22  for istre=1:nstre
23
24  stres(ielem,igaus,istre) = 0.0;
25  stran(ielem,igaus,istre) = 0.0;
26  end
27  end
28  end
29
30  for itotv=1:ntotv
31  tdisp(itotv,1)=0.0;
32  end
33
34  end %if
35
36  if(isolve == 2)
37
38  stres = zeros(nelem,mgaus,nstre);
39  stran = zeros(nelem,mgaus,nstre);
40
41  tdisp =zeros(ntotv,1);
42
43  end %if
44
45  if(icase == 1)
46
47  ncrack =30;
48  crack(1) = 2010;
49  crack(2) = 2011;
50  crack(3) = 2012;
51  crack(4) = 2013;
52  crack(5) = 2014;
53  crack(6) = 2015;

```

```

54 crack(7) = 2016;
55 crack(8) = 2017;
56 crack(9) = 2018;
57 crack(10) = 2019;
58 crack(11) = 2020;
59 crack(12) = 2021;
60 crack(13) = 2022;
61 crack(14) = 2023;
62 crack(15) = 2024;
63 %
64 crack(16) = 2051;
65 crack(17) = 2052;
66 crack(18) = 2053;
67 crack(19) = 2054;
68 crack(20) = 2055;
69 crack(21) = 2056;
70 crack(22) = 2057;
71 crack(23) = 2058;
72 crack(24) = 2059;
73 crack(25) = 2060;
74 crack(26) = 2061;
75 crack(27) = 2062;
76 crack(28) = 2063;
77 crack(29) = 2064;
78 crack(30) = 2065;
79
80 for icrack=1:ncrack
81 lnode = crack(icrack);
82 itotv=ntotv2+lnode;
83 tdisp(itotv) = 0.99;
84 end
85
86 end
87
88 if(icase == 2)
89
90 ncrack =26;
91 crack(1) = 291;
92 crack(2) = 292;
93 crack(3) = 296;
94 crack(4) = 297;
95 crack(5) = 1553;
96 crack(6) = 1554;
97 crack(7) = 1555;
98 crack(8) = 1556;
99 crack(9) = 1557;
100 crack(10) = 1558;
101 crack(11) = 1559;
102 crack(12) = 1560;
103 crack(13) = 1561;
104 crack(14) = 1562;
105 crack(15) = 1563;
106 crack(16) = 1568;
107 crack(17) = 1569;
108 crack(18) = 1570;
109 crack(19) = 1571;
110 crack(20) = 1572;
111 crack(21) = 1573;
112 crack(22) = 1574;
113 crack(23) = 1575;
114 crack(24) = 1576;
115 crack(25) = 1577;
116 crack(26) = 1578;
117
118 for icrack=1:ncrack
119 lnode = crack(icrack);
120 itotv=ntotv2+lnode;
121 tdisp(itotv) = 0.99;
122 end
123
124 end
125
126
127 end %endfunction

```

Line numbers:

6:	The number of DOFs for displacements per node.
7:	Total number of variables for displacements.
8:	Total number of variables.
10–13:	Order of numerical integration, depending on the element type.
14:	Total number of integration points per element.
18–34:	If the solution is in un-optimized mode (isolve = 1), initialize stress, strain arrays and solution vector.
36–43:	If solution is in optimized mode (isolve = 2), initialize stress, strain arrays and solution vector.
45–86:	If simulation is for crack (icase = 1), set initial crack configuration in given FEM mesh. Given node numbers have their order parameters values equal to 0.99.
88–124:	If simulation is for crack-hole (icase = 2), set initial crack configuration in given FEM mesh. Given node numbers have their order parameters values equal to 0.99.

Function

residual_v1.m

This function calculates the global rhs vector. It is in longhand format and is not optimized.

The function makes call to the following functions:

- **sfr2.m**
- **jacob2.m**
- **modps.m**
- **bmats.m**

Variable and array list:

npoin:	Number of nodes
nelem:	Number of elements.
nnode:	Number of nodes per element.
nstre:	Number of stress components.
ndime:	Number of Cartesian component dimension.
ndofn:	Number of DOF per node.
ngaus:	Order of numerical integration.
ntype:	Solution type, ntype = 1 for plane-stress, ntype = 2 for plane-strain.
dtime:	Time increment for numerical integration.
constk:	Small value to avoid overflow for cracked elements.
cenerg:	Critical strain energy for fracture.
constl:	Interface control parameter.
constn:	Power term for the penalty parameter.
coneta:	Magnitude of the penalty term.
tdisp(ntotv):	Solution vector, ntotv = npoin × ndofn.
rload(ntotv):	Global rhs, load, vector.
tdisp_old (ntotv):	Nodal values from previous time step.
posgp(ngaus):	Position of integration points.
weigp(ngaus):	Weights for numerical integration.
props(nmat, nprop):	Material mechanical properties, nmat number of different materials in the FEM mesh and nprop, number of properties.
lnods(nelem, nnode):	Element nodal connectivity list.
coord(npoin, ndime):	Cartesian coordinates of the nodes.
stres(nelem, ngaus,nstre):	Stress values at the integration points of the elements.
stran(nelem, ngaus,nstre):	Strain values at the integration point of the elements.

Listing:

```

1   function [rload] = residual_v1
2     (npoin,nelem,nnode,nstre,ndime,
3     ndofn, ...
4      ngaus,ntype,lnods,coord,
5       props,posgp, ...
6       weigp,dtime,constk,
7       cenerg,constl,constn, ...
8       coneta,stres,stran,tdisp,
9       tdisp_old)
10    format long;
11    %--- order of integration
12    ngaus2=ngaus;
13    if (nnode == 3)
14      ngaus2=1;
15    end
16    mgaus = ngaus*ngaus2;
17    %%--initialize global and local
18    %rhs vectors:
19    ntotv = npoin*ndofn;
20    nevab = nnode*ndofn;
21    ndofn2 = ndofn-1;
22    ndofn3 = ndofn-2;
23    ntotv2 = npoin*ndofn2;
24    nevab2 = nnode*ndofn2;
25    nevab3 = nnode*ndofn3;
26    %%--global residuals:
27    rload = zeros(ntotv,1);
28    for ielem=1:nelem
29      %%-- initialize element loads:
30      for ievab=1:nevab
31        eload(ievab) = 0.0;
32      end
33      for ievab=1:nevab2
34        eload1(ievab) = 0.0;
35      end
36      for ievab=1:nevab3
37        eload2(ievab) = 0.0;
38      end
39    end
40  end
41  for ievab=1:nevab3
42    eload3(ievab) = 0.0;
43  end

```

```

44 ;eload2(ievab) =0.0;
45 end
46
47 %-- elemental values
48 %--%
49
50 for inode=1:nnode
51 lnode=lnods(ielem,inode);
52 itotv=ntotv2+lnode;
53 ephi(inode)=tdisp(itotv);
54 ephir(inode)=tdisp(itotv)
55 -tdisp_old(itotv);
56 end
57
58 %--- Coordinates of element nodes:
59 for inode=1:nnode
60 lnode=lnods(ielem,inode);
61 for idime=1:ndime
62 elcod(idime,inode)=coord(lnode,
63 idime);
64 end
65 end
66
67 %--- integrate elemental loads:
68 kgasp=0;
69 for igaus=1:ngaus
70 exisp=posgp(igaus);
71 for jgaus=1:ngaus2
72 etasp=posgp(jgaus);
73 if(nnode ==3)
74 etasp=posgp(ngaus+igaus);
75 end
76
77 kgasp=kgasp+1;
78
79 [shape,deriv]=sfr2(exisp,etasp,
80 nnode);
81 [cartd,djacb,gpcod]=jacob2(ielem,
82 elcod,kgasp,shape,deriv,nnode,
83 ndime);
84 dvolu=djacb*weigp(igaus)*weigp
85 (jgaus);
86 if(nnode == 3)
87 dvolu=djacb*weigp(igaus);
88 end
89
90 %--- values at the integration
91 points:
92 phigp=0.0;
93 phirgp=0.0;
94 for inode=1:nnode
95 phigp=phigp + ephi(inode)*shape
96 (inode);
97 phirgp=phirgp + ephir(inode)*
98 shape(inode);
99 end
100 mtype=1;
101 [dmatx]=modps(mtype,ntype,nstre,
102 props);
103 %---
104
105 [bmatx]=bmats(cartd,shape,inode);
106
107 % strain energy:
108 senerg=0.0;
109 for istre=1:nstre
110
111 senerg=senerg + 0.5*stres(ielem,
112 kgasp,istre) * stran(ielem,kgasp,
113 istre);
114 end
115
116 %--- penalty term:
117 constx=0.0;
118 if(phirgp < 0.0 )
119 constx=-phirgp;
120 end
121
122 %--- residuals (rhs)
123
124 %--- eload1:
125
126 for ievab=1:nevab2
127 for istre=1:nstre
128
129

```

```

130 eload1(ievab) = eload1(ievab) +
  ((1.0-phigp)^2 + constk)*bmatx
  (istre,ievab)*...
131      stres(ielem,kgas,istre)
      *dvolu;
132
133 end
134 end
135
136 %--- eload2
137
138 for istre=1:ndime
139 dummy(istre) = 0.0;
140 end
141
142 for istre = 1:ndime
143 for ievab = 1:nevab3
144 dummy(istre) = dummy(istre) + cartd
  (istre,ievab)*phigp;
145 end
146 end
147
148 for ievab=1:nevab3
149 for istre=1:ndime
150 eload2(ievab) = eload2(ievab) +
  cenerg*constl*cartd(istre,ievab)*
  ...
151      dummy(istre)*dvolu;
152 end
153 end
154
155 for inode=1:nnode
156 eload2(inode) = eload2(inode) +
  ((cenerg/constl)+2.0*senerg)*...
157      phigp*shape(inode)*dvolu;
158 end
159
160 for inode=1:nnode
161 eload2(inode) = eload2(inode) -
  2.0*shape(inode)*...
162      (senerg-0.5*(coneta/dtime)
      *constx^constn)*dvolu;
163
164 end
165
166 end %jgaus
167 end %igaus
168
169 %--- assemble global residuals:
170
171 for inode=1:nnode
172 lnode = lnods(ielem,inode);
173 for idofn =1:ndofn2
174 ievab=(inode-1)*ndofn2+idofn;
175 itotv=(lnode-1)*ndofn2+idofn;
176 rload(itotv) = rload(itotv)
  +eload1( :,ievab);
177 end
178 end
179
180 for inode=1:nnode
181 lnode = lnods(ielem,inode);
182 for idofn =1:ndofn3
183 ievab=(inode-1)*ndofn3+idofn;
184      itotv=(lnode-1)*ndofn3+idofn
      +ntotv2;
185 rload(itotv) = rload(itotv) + eload2
  ( :,ievab);
186 end
187 end
188
189 end %ielem
190
191 end %endfunction

```

Line numbers:

10–13:	Order of numerical integration, depending on the element type.
15:	Total number of integration points per element.
19:	Total number of variables.
20:	Total number of variables per element.
21:	Number of DOF for displacements per node.
22:	Number of DOF for order parameter per node.
23:	Total number of variables for displacements.
24:	Number of variables for displacements per element.
25:	Number of variables for order parameter per element.
29:	Initialize global rhs, load, vector.
31–189:	Loop over elements.
33–46:	Initialize element rhs, load, vectors.
48–56:	Calculate the elemental nodal order parameter values.
58–65:	Calculate the elemental nodal Cartesian coordinates.
67–167:	Numerical integration of element.
71,	Coordinates of the integration points in local coordinate system.
73, 75:	
78:	Increment integration point counter.

(continued)

80:	Calculate shape function and their derivative values.
82:	Calculate Cartesian derivatives of shape functions.
84–88:	Calculate element area/volume.
90–99:	Calculate the values of order parameters at the current integration point, Eq. 6.131.
101:	Calculate the elasticity matrix.
105:	Calculate the strain matrix.
109–114:	Calculate the elastic strain energy.
116–121:	Form penalty term, Eq. 6.140.
123–167:	Form rhs, load, vectors.
127–134:	Form the first row of rhs, load, vector, Eq. 6.133.
136–146:	Form the first term of second row of rhs, load, vector, Eq. 6.142.
155–158:	Form the second term of second row of rhs, load, vector, Eq. 6.142.
160–164:	Form the last term of second row of rhs, load, vector, Eq. 6.142.
169–188:	Assemble element rhs, load, vectors into the global rhs, load vector.

Function

residual_v2.m

This function calculates the global rhs vector. It is optimized for Matlab/Octave.

The function makes call to the following functions:

- **sfr2.m**
- **modps.m**
- **bmats1.m**
- **bmats2.m**

Variable and array list:

npoin:	Number of nodes.
nelem:	Number of elements.
nnode:	Number of nodes per element.
nstre:	Number of stress components.
ndime:	Number of Cartesian component dimension.
ndofn:	Number of DOF per node.
ngaus:	Order of numerical integration.
ntype:	Solution type, ntype = 1 for plane-stress, ntype = 2 for plane-strain.
dtime:	Time increment for numerical integration.

constk:	Small value to avoid overflow for cracked elements.
cenerg:	Critical strain energy for fracture.
constl:	Interface control parameter.
constn:	Power term for the penalty parameter.
coneta:	Magnitude of the penalty term.
tdisp(ntotv):	Solution vector, ntotv = npoin × ndofn.
rload(ntotv):	Global rhs, load, vector.
tdisp_old(ntotv):	Nodal values from previous time step.
posgp(ngaus):	Position of integration points.
weigp(ngaus):	Weights for numerical integration.
props(nmat, nprop):	Material mechanical properties, nmat number of different materials in the FEM mesh and nprop, number of properties.
lnods(nelem, nnode):	Element nodal connectivity list.
coord(npoin, ndime):	Cartesian coordinates of the nodes.
dvolum(nelem, mgaus):	Precalculated element's area/volumes.
stres(nelem, mgaus, nstre):	Stress values at the integration points of the elements.
stran(nelem, mgaus, nstre):	Strain values at the integration point of the elements.
dgdx(nelem, mgaus, nstre, nnode):	Precalculated Cartesian derivatives of shape functions for all elements

Listing:

```

1  function [rload ]=residual_v2
2    (npoin,nelem,nnode,nstre,ndime,
3     ndofn, ...
4      ngaus,ntype,lnods,coord,
5      props,posgp, ...
6      weigp,dgdx,dvolum,dtime,
7      constk,cenerg, ...
8      constl,constn,coneta,stres,
9      stran,tdisp, ...
10     tdisp_old)
11
12   format long;
13
14   %--- order of integration
15
16   ngaus2=ngaus;
17
18   if(nnode == 3)
19     ngaus2=1;

```

```

14 end
15
16 mgaus = ngaus*ngaus2;
17
18 %--initialize global and local rhs
19 % vectors:
20 ntotv = npoin*ndofn;
21 nevab = nnodes*ndofn;
22 ndofn2 = ndofn-1;
23 ndofn3 = ndofn-2;
24 ntotv2 = npoin*ndofn2;
25 nevab2 = nnodes*ndofn2;
26 nevab3 = nnodes*ndofn3;
27
28 %--global residuals:
29
30 rload = zeros(ntotv,1);
31
32 %-- element loads:
33
34 eload = zeros(nelem,nevab);
35 eload1 = zeros(nelem,nevab2);
36 eload2 = zeros(nelem,nevab3);
37
38 %--
39 %-- elemental values
40 %--
41
42 ephi = zeros(nelem,nnodes);
43 ephir = zeros(nelem,nnodes);
44
45 for inode=1:nnodes
46 lnode = lnods( :,inode );
47 itotv = ntotv2 +lnode;
48 ephi( :,inode ) = tdisp(itotv);
49 ephir( :,inode ) = tdisp(itotv) -
50 tdisp_old(itotv);
51 end
52
53 %-- integrate elemental loads:
54 kgasp=0;
55 for igaus=1:ngaus
56 exisp=posgp(igaus);
57 for jgaus=1:ngaus2
58 etasp = posgp(jgaus);
59 if(nnodes ==3)
60 etasp=posgp(ngaus+igaus);
61 end
62
63 kgasp=kgasp+1;
64 [shape,deriv]=sfr2(exisp,etasp,
65 nnodes);
66 %-- values at the integration
67 % points:
68 phigp=zeros(nelem,1);
69 phirgp=zeros(nelem,1);
70
71 for inode=1:nnodes
72 phigp = phigp + ephi( :,inode )
73 *shape(inode);
74 phirgp = phirgp + ephir( :, inode )
75 *shape(inode);
76 end
77
78 mtype = 1;
79 [dmatx] = modps(mtype,ntype,nstre,
80 props);
81
82 %-- cartesien derivative matrices;
83
84
85 % strain energy:
86
87 senerg = zeros(nelem,1);
88 for istre=1:nstre
89
90 senerg = senerg + 0.5*stres
91 ( :,kgasp,istre) .* stran
92 ( :,kgasp,istre);
93
94 %-- penalty term:
95
96 constx = zeros(nelem,1);
97 inrange = (phirgp < 0 );
98 constx(inrange) = -phirgp
99 (inrange);
100
101 %-- residuals (rhs)

```

```

102 %--- eload1:
103
104 for ievab=1:nevab2
105 for istre=1:nstre
106
107 eload1( :,ievab) = eload1( :,ievab)
108 + ((1.0-phigp).^2 + constk) .
* bmatx2( :,istre,ievab).* ...
109     stres( :,kgasp,istre).*dvolum
110     ( :,kgasp);
111 end
112
113 %--- eload2
114
115 dummy = zeros(nelem,ndime);
116
117 for istre = 1:ndime
118 for ievab = 1:nevab3
119 dummy( :,istre) = dummy( :,istre) +
bmatx1( :,istre,ievab).*phigp;
120 end
121 end
122
123 for ievab=1:nevab3
124 for istre=1:ndime
125 eload2( :,ievab) = eload2( :,ievab)
+ cenerg*constl*bmatx1( :,istre,
ievab).* ...
126     dummy( :,istre).*dvolum
( :,kgasp);
127 end
128 end
129
130 for inode=1:nnode
131 eload2( :,inode) = eload2( :,inode)
+ ((cenerg/constl)+2.0*senerg) .
* ...
132     phigp*shape(inode).*dvolum
( :,kgasp);
133 end
134
135 for inode=1:nnode
136 eload2( :,inode) = eload2( :,inode)
- 2.0*shape(inode)* ...
137     (senerg-0.5*(coneta/dtime)
*constx.^constrn).*dvolum
( :,kgasp);
138
139 end
140
141 end %jgaus
142 end %igaus
143
144 %--- assemble global residuals:
145
146 for inode=1:nnode
147 lnode = lnods( :,inode);
148 for idofn =1:ndofn2
149 ievab=(inode-1)*ndofn2+idofn;
150 itotv=(lnode-1)*ndofn2+idofn;
151 rload(itotv) = rload(itotv)
+eload1( :,ievab);
152 end
153 end
154
155 for inode=1:nnode
156 lnode = lnods( :,inode);
157 for idofn =1:ndofn3
158 ievab=(inode-1)*ndofn3+idofn;
159 itotv=(lnode-1)*ndofn3+idofn
+nntotv2;
160 rload(itotv) = rload(itotv)
+ eload2( :,ievab);
161 end
162 end
163
164 end %endfunction

```

Line numbers:

11–14:	Order of numerical integration, depending on the element type.
16:	Total number of integration points per element.
20:	Total number of variables.
21:	Total number of variables per element.
22:	Number of DOFs for displacements per node.
23:	Number of DOF for order parameter per node.
24:	Total number of variables for displacements.
25:	Number of variables for displacements per element.
26:	Number of variables for order parameter per element.
30:	Initialize global rhs, load, vector.
32–36:	Initialize rhs, load, vectors for all elements in the solution.
42–43:	Initialize arrays that hold the value of order parameters.
45–50:	Calculate elemental nodal values of order parameters for all elements.
52–142:	Numerical integration of elements.

(continued)

56, 58, 60:	Coordinates of the integration points in local coordinate system.
63:	Increment the integration point counter.
64:	Calculate the shape function and their derivative values.
66–75:	Values of the order parameters at the current integration point, Eq. 6.131, for all elements.
77:	Calculate the elasticity matrix.
81:	Calculate the Cartesian derivative of shape functions for all elements.
82:	Calculate the strain matrix for all elements.
87–92:	Calculate the elastic strain energy for all elements.
94–99:	Form penalty term, Eq. 6.140, for all elements.
100–139:	Form rhs, load, vectors of all elements.
104–111:	Form the first row of rhs, load, vector, Eq. 6.133.
113–128:	Form the first term of second row of rhs, load, vector, Eq. 6.142.
130–133:	Form the second term of second row of rhs, load, vector, Eq. 6.142.
135–139:	Form the last term of second row of rhs, load, vector Eq. 6.142.
144–162:	Assemble element rhs, load, vectors into the global rhs, load vector.

nype:	Solution type, ntype = 1 for plane-stress, ntype = 2 for plane-strain.
matno(nelem):	Material number of elements.
tdisp(ntotv):	Solution vector, ntotv = npoin × ndofn.
asdis(ntotv):	Current solution vector.
posgp(ngaus):	Position of integration points.
weigp(ngaus):	Weights for numerical integration.
props(nmat, nprop):	Material properties, nmat number of different materials in the FEM mesh and nprop, number of properties.
lnods(nelem, nnode):	Element nodal connectivity list.
coord(npoin, ndime):	Cartesian coordinates of the nodes.
stres(nelem, mgaus,nstre):	Stress values at the integration points of the elements.
stran(nelem, mgaus,nstre):	Strain values at the integration point of the elements.

Listing:

```

1  function [stran,stres] = stress_
fract_v1(asdis,nelem,npoin,nnode,
ngaus,nstre,props, ...
2      ntype,ndofn,ndime,lnods,
matno,coord,posgp,weigp, ...
3      tdisp)
4  format long;
5
6  %--- order of integration
7  ngaus2=ngaus;
8  if(nnod == 3)
9    ngaus2=1;
10 end
11
12 mgaus = ngaus*ngaus2;
13
14 %--initialize:
15
16 ndofn2 =2;
17 nevab2 =nnode*ndofn;
18 ntotv2 =npoin*ndofn2;
19 nevab = nnod*ndofn2;
20
21 for ielem =1:nelem
22
23 for igaus =1:mgaus
24 for istre =1:nstre
25 stres(ielem,igaus,istre) = 0.0;
26 stran(ielem,igaus,istre) = 0.0;

```

Function

stress_fract_v1.m

This function calculates the stress and strain values at the integration points of all elements. It is in longhand format and is not optimized.

The function makes calls to the following functions:

- **modps.m**
- **sfr2.m**
- **jacob2.m**
- **bmats.m**

Variable and array list:

npoin:	Number of nodes.
nelem:	Number of elements.
nnode:	Number of nodes per element.
nstre:	Number of stress components.
ndime:	Number of Cartesian component dimension.
ndofn:	Number of DOF per node.
ngaus:	Order of numerical integration.

```

27 end
28 end
29
30 %--Nodal displacements:
31
32 for inode =1:nnode
33 lnode =lnods(ielem,inode);
34 for idofn=1:ndofn2
35 ievab=(inode-1)*ndofn2+idofn;
36 itotv=(lnode-1)*ndofn2+idofn;
37 eldis(ievab)=tdisp(itotv);
38 end
39 end
40
41 %--- elasticity matrix:
42
43 mtype =1;
44 [dmatx] =modps(mtype,ntype,nstre,
props);
45
46 %--- Coordinates of element nodes:
47
48 for inode=1:nnode
49 lnode=lnods(ielem,inode);
50 for idime=1:ndime
51 elcod(idime,inode)=coord(lnode,
idime);
52 end
53 end
54
55 %--- calculate strains and stresses
at integration points:
56
57 kgasp=0;
58
59 for igaus=1:ngaus
60 exisp=posgp(igaus);
61 for jgaus=1:ngaus2
62 etasp =posgp(jgaus);
63 if(nnode ==3)
64 etasp=posgp(ngaus+igaus);
65 end
66
67 kgasp=kgasp+1;
68
69 [shape,deriv]=sfr2(exisp,etasp,
nnode);
70
71 [cartd,djacb,gpcod]=jacob2(ielem,
elcod,kgasp,shape,deriv,nnode,
ndime);
72
73 %--- strain matrix:
74
75 [bmatx]=bmats(cartd,shape,inode);
76
77 %--- calculate the strains:
78
79 for istre=1:nstre
80 for ievab=1:nevab
81
82 stran(ielem,kgasp,istre) =stran
(ielem,kgasp,istre)+bmatx(istre,
ievab)*eldis(ievab);
83
84 end
85 end
86
87 %calculate stresses:
88
89 for istre=1:nstre
90 for jstre=1:nstre
91
92 stres(ielem,kgasp,istre) =stres
(ielem,kgasp,istre)+dmatx(istre,
jstre)*stran(ielem,kgasp,jstre);
93
94 end
95 end
96
97 end%igaus
98 end%jgaus
99
100 end%ielem
101 end%endfunction

```

Line numbers:

6–10:	Order of numerical integration.
12:	Total number of integration points per element.
16:	Total DOF for displacements per node.
17:	Total number of variables per element.
18:	Total number of variables.
19:	Total number of variables for displacements per element.
21–100:	Loop over elements.
23–28:	Initialize arrays holding elemental stress and strain values.
32–39:	Get elemental nodal displacement values.
44:	Form the elasticity matrix.
48–53:	Elemental nodal Cartesian coordinates.
55–98:	Calculate the stress and strain values at the integration points of the elements.

(continued)

59–98:	Loop over integration points.
60–64:	Position of integration points in the local coordinate system.
67:	Increment integration point counter.
69:	Calculate the shape functions and their derivative values.
71:	Calculate the Cartesian derivatives of shape functions.
75:	Form strain matrix.
79–85:	Calculate strain values from nodal displacements.
89–95:	Calculate stress values from strain values.

coord(npoin, ndime):	Cartesian coordinates of the nodes.
dvolum(nelem, mgaus):	Precalculated element's area/volumes.
stres(nelem,mgaus, nstre):	Stress values at the integration points of the elements.
stran(nelem, mgaus,nstre):	Strain values at the integration point of the elements.
dgdx(nelem, mgaus,nstre, nnode):	Precalculated Cartesian derivatives of shape functions for all elements.

Listing:

```

1  function [stran,stres] = stress_fr
act_v2(asdis,nelem,npoin,nnode,
ngaus,nstre,props, ...
2      ntype,ndofn,ndime,lnods,
matno,coord,posgp,weigp, ...
3      dgdx,dvolum,tdisp)
4  format long;
5
6  %--- order of integration
7  ngaus2=ngaus;
8  if(nnode == 3)
9    ngaus2=1;
10 end
11
12 ngaus = ngaus*ngaus2;
13
14 %--initialize:
15
16 ndofn2 =2;
17 nevab2 =nnode*ndofn;
18 ntotv2 =npoin*ndofn2;
19 nevab = nnodes*ndofn2;
20
21 stres = zeros(nelem,mgaus,nstre);
22 stran = zeros(nelem,mgaus,nstre);
23 eldis = zeros(nelem,nevab);
24
25 %--Nodal displacements:
26
27 for inode =1:nnode
28 lnode = lnods( :,inode);
29 for idofn=1:ndofn2
30 ievab =(inode-1)*ndofn2+idofn;
31 itotv =(lnode-1)*ndofn2+idofn;
32 eldis( :,ievab)=tdisp(itotv);
33 end
34 end

```

Function

stress_fract_v2.m

This function calculates the stress and strain values at the integration points of all elements. It is optimized for Matlab/Octave.

The function makes calls to the following functions:

- **modps.m**
- **bmats2.m**
- **sfr2.m**

Variable and array list:

npoin:	Number of nodes.
nelem:	Number of elements.
nnode:	Number of nodes per element.
nstre:	Number of stress components.
ndime:	Number of Cartesian component dimension.
ndofn:	Number of DOF per node.
ngaus:	Order of numerical integration.
ntype:	Solution type, ntype = 1 for plane-stress, ntype = 2 for plane-strain.
matno(nelem):	Material number of elements.
tdisp(ntotv):	Solution vector, ntotv = npoin × ndofn.
asdis(ntotv):	Current solution vector.
posgp(ngaus):	Position of integration points.
weigp(ngaus):	Weights for numerical integration.
props(nmat, nprop):	Material properties, nmat number of different materials in the FEM mesh and nprop, number of properties.
lnods(nelem, nnode):	Element nodal connectivity list.

```

35                                         77
36 %--- elasticity matrix:                78 end
37                                         79 end
38 mtype =1;                            80
39 [dmatx] = modps(mtype, ntype, nstre,
      props);                         81 end %igaus
40                                         82 end %jgaus
41 %--- calculate strains and stresses   83
42                                         at integration points:
43 kgasp=0;
44
45 for igaus=1:ngaus
46 exisp=posgp(igaus);
47 for jgaus=1:ngaus2
48 nbsp;etasp =posgp(jgaus);
49 if(nnode ==3)
50 etasp=posgp(ngaus+igaus);
51 end
52
53 kgasp=kgasp+1;
54
55 [shape,deriv]=sfr2(exisp,etasp,
      nnode);
56
57 %--- strain matrix:
58
59 [bmatx]=bmats2(dgdx,nelem,nnode,
      nstre,nevab,kgasp);
60
61 %--- calculate the strains:
62
63 for istre =1:nstre
64 for ievab =1:nevab
65
66 stran( :,kgasp,istre) = stran
      ( :,kgasp,istre) +bmatx( :,istre,
      ievab).*eldis( :,ievab);
67
68 end
69 end
70
71 %calculate stresses:
72
73 for istre =1:nstre
74 for jstre =1:nstre
75
76 stres( :,kgasp,istre) = stres
      ( :,kgasp,istre) + dmatx(istre,
      jstre).*stran( :,kgasp,jstre);

```

Line numbers:

6–10:	Order of numerical integration, depending on the element type.
12:	Total number of integration points per element.
16:	Total DOFs for displacements per node.
17:	Total number of variables per element.
18:	Total number of variables.
19:	Total number of variables for displacements per element.
21–22:	Initialize stress and strain array for all elements.
25–34:	Calculate elemental nodal displacements for all elements.
45–82:	Calculate stress and strain values at the integration points of all elements.
46–51:	Position of integration points at the local coordinates.
53:	Increment integration point counter.
55:	Calculate the shape functions and their derivative values.
59:	Calculate strain matrix for all elements.
63–69:	Calculate the strain values from displacements for all elements.
73–79:	Calculate the stress values from strains for all elements.

References

1. Griffith AA (1921) The phenomena of rupture and flow in solids. Philos Trans R Soc London, Ser A 221:163–198
2. Irwin GR (1958) Elasticity and plasticity: fracture. In: Flügge S (ed) Encyclopedia of physics, vol 6. Springer, Berlin
3. Aranson JS, Kalatsy VA, Vinokur VM (2000) Continuum field description of crack propagation, Phys Rev Lett 85:118
4. Karma A, Kessler DA, Levine H (2001) Phase-field model of mode-III dynamic fracture. Phys Rev Lett 87:045501
5. Hakim V, Karma A (2009) Laws of crack motion and phase-field model of fracture. J Mech Phys Solids 57:342

6. Biner SB, Hu SY (2009) Simulation of damage evolution in composites: A phase-field model. *Acta Mater* 57:2088
7. Kuhn C, Muller R (2010) A continuum phase-field model of fracture. *Eng Fract Mech* 77:3625
8. Miehe C, Hofacker M, Welschinger F (2010) A phase-field model for rate-independent crack propagation: Robust algorithmic implementation based on operator splits. *Comput Methods Appl Mech Eng* 199:2765
9. Miehe C, Welschinger F, Hofacker M (2010) Thermodynamically consistent phase-field models of fracture: variational principles and multifield FE implementation. *Int J Numer Methods Eng* 83:1273
10. Borden MJ, Verhoosel CV, Scott MA, Hughes TJR, Landis CM (2012) A phase-field description of dynamic brittle fracture. *Comput Methods Appl Mech Eng* 217–220:77
11. Msekh MA, Sargado JM, Jamshidian M, Areras PM (2015) Abaqus implementation of phase-field model for brittle fracture. *Comput Mater Sci* 96:272
12. Ulmer H, Hofacker M, Miehe C (2013) Phase-field modeling of brittle and ductile fracture. *Proc Appl Math Mech* 13:533
13. Biner SB, Hu SY (2009) Simulation of damage evolution in discontinuously reinforced metal matrix composites: a phase-field model. *Int J Fract* 158:99
14. Duda FP, Ciarbonetti A, Sanches PJ, Uespe AE (2015) A phase-field/gradient damage model for brittle fracture in elastic-plastic solids. *Int J Plast* 65:269
15. Mikelic A, Wheeler MF, Wick T (2015) A phase-field method for propagating fluid-filled fracture coupled to surrounding porous medium. *Multiscale Model Simul* 13:367
16. Chaboche JL (1987) Continuum damage mechanics: present state and future trends. *Nucl Eng Des* 105:19

7.1 Introduction

The phase-field crystal, PFC, method introduced by Elder and coworkers [1–3], can be viewed as multiscale simulation algorithm that bridges the classical molecular dynamics, MD, simulations and the phase-field methods covered in the previous chapters. PFC method introduces an order parameter defined as the local-time-averaged atomic number density which is able to produce periodicity of crystal lattices. In the model, any perturbation or lattice defects result in an increase in the free energy, thus enabling to obtain the information which has been only possible by the atomistic simulations previously. In addition, PFC method produces various atomistic events in much larger spatial and temporal dimensions that are not easily accessible with current MD simulation techniques. Therefore, PFC method has emerged as an attractive simulation approach.

The original PFC [1–3] model has been developed in the spirit of Ginzburg–Landau theory as a phenomenological approach. In the model, as in the classical phase-field models, the evolution of locally conserved order parameter $\phi(r)$ is given by

$$\frac{\partial \phi}{\partial t} = \nabla \cdot M \nabla \frac{\delta F}{\delta \phi} \quad (7.1)$$

where M is the mobility, F is the free energy of the system as function of $\phi(r)$, and r is the position vector. In PFC models, a minimum requirement for the free energy functional is that it should produce a periodic lattice structure at the ground state in some parameter range. The simplest known free energy functional that meets this condition is the form given by Swift and Hohenberg to study the convective instabilities [4]. This free energy functional is expressed as:

$$F(\phi(r)) = \int \left\{ \frac{\phi(r)}{2} \left[\alpha + \lambda(q_0^2 + \nabla^2)^2 \right] \phi(r) - \frac{h}{3} \phi(r)^3 + \frac{g}{4} \phi(r)^4 \right\} dr \quad (7.2)$$

where α , λ , h , g_0 , and g are phenomenological parameters to fit the properties of the material of interest. It is convenient to rewrite the Eq. 7.2 in dimensionless form, by introducing a new set of variables as:

$$\begin{aligned} x &= q_o r; \\ \psi &= \sqrt{\frac{g}{\lambda q_0^4}} \left(\phi - \frac{h}{3g} \right); \\ \epsilon &= \frac{1}{\lambda q_0^4} \left(\frac{h}{3g} - \alpha \right) \end{aligned} \quad (7.3)$$

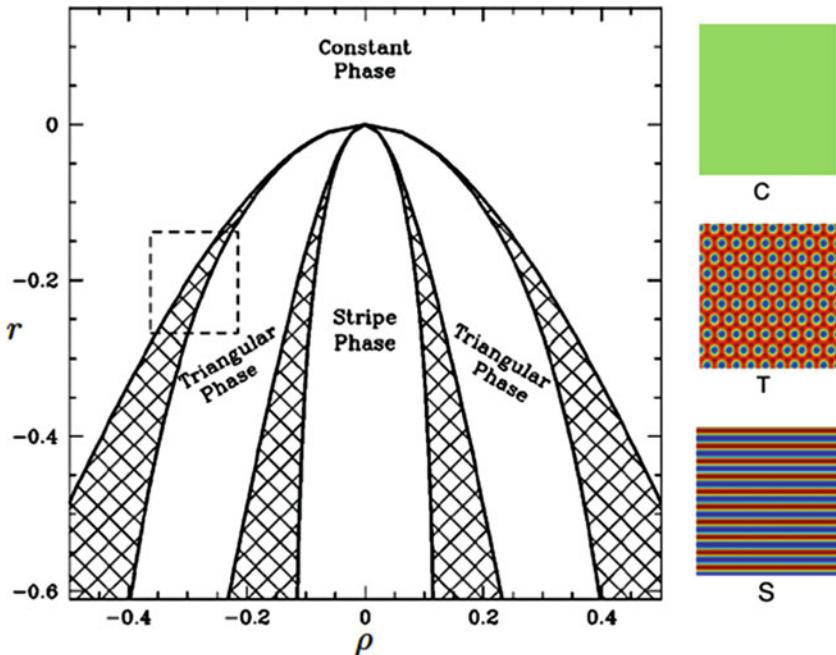


Fig. 7.1 Phase diagram derived from Eq. 7.4, adopted from [2]. The hatched regions are the regions of coexistence

in which all terms that are either constant or linearly dependent to the dimensionless order parameter field ψ can be ignored and the value of h is usually taken as zero. Then, a dimensionless free energy \tilde{F} as $g\lambda^{-2}g_0^{-5}$ times the original free energy, without the constant and linear parts, takes the form,

$$\begin{aligned}\tilde{F}(\psi(x)) \\ = \int \left\{ \epsilon \frac{\psi(x)^2}{2} + \frac{1}{4} \psi(x)^4 + \frac{\psi(x)}{2} (1 + \nabla^2)^2 \psi(x) \right\} dx\end{aligned}\quad (7.4)$$

Here, ϵ is a constant proportional to the deviation of the temperature from the melting temperature; therefore, it takes a negative value. The first two terms in the integrand constitute a double-well potential and guarantees the presence of solid and liquid phases. The last term, containing gradient term, accounts for the deviation from the equilibrium profile owing to increase in elastic strain energy and/or the presence of other energy risers such as the lattice defects.

Even in its simplest form, the PFC model of Eq. 7.4 is a powerful modeling technique. By considering the atomic configurations, the phase diagram given in Fig. 7.1 is derived from Eq. 7.4 in [2].

Figure 7.1 indicates the equilibrium phases. These include constant (liquid) phase, stripe phase, triangular structure in two-dimension and body-centered cubic, *bcc*, structure in three-dimension and the coexistence regions in which liquid and solid phases are in equilibrium.

The work given in [3] also linked the PFC method to the classical density-functional theory, DFT [5]. This connection enabled to development of PFC model for binary alloys and used in simulations of dendritic growth, spinodal decomposition and epitaxial growth. In the work of Jaatinen and Ala-Nissila [6] the PFC phase diagram is extended to produce face-centered cubic, *fcc*, and hexagonal-closed-packed, *hcp*, lattices in three-dimension.

Since its introduction as a multiscale modeling approach, PFC method is applied to

structural transformations [7–9], phase nucleation and growth [10], spinodal decomposition with grain boundaries [11], pre-melting at grain boundaries and dislocations [12–14], to solute drag [15], the detailed analysis of dislocations [16], to grain growth [17], to amorphous materials and glass formation [18]. Its application to real Al–Cu alloy system also can be found in [19]. The cited references are in here just a few examples of the application of PFC model, a detailed overview of the PFC method can be found in [20]. Also, an extensive coverage of the theoretical and practical aspects of PFC method is given in by Pravatas and Elder in their textbook, *Phase-Field Methods in Materials Science and Engineering* [21].

References

1. Elder KR, Katakowski M, Haataja M, Grant M (2002) Modeling elasticity in crystal growth. *Phys Rev Lett* 88:245701
2. Elder KR, Grand M (2004) Modeling elastic and plastic deformation in non-equilibrium processing using phase-field crystals. *Phys Rev E* 70:051505
3. Elder KR, Provatas N, Berry J, Stefanovic P, Grant M (2007) Phase-field crystal modeling and classical density functional theory of freezing. *Phys Rev B* 75:064107
4. Swift J, Hohenberg PC (1977) Hydrodynamic fluctuations at convective instability. *Phys Rev A* 15:319
5. Ramakrishnan TV, Yussouff M (1979) First principles order-parameter theory of freezing. *Phys Rev B* 19:2775
6. Jaatinen A, Ala-Nissila T (2010) Extended phase-diagram of the three-dimensional phase-field crystal model. *J Phys Condens Matter* 22:205402
7. Greenwood M, Rottler J, Provatas N (2011) Phase-field crystal methodology for modeling structural transformations. *Phys Rev E* 83:031601
8. Greenwood M, Ofori-Opoku N, Rottler J, Provatas N (2011) Modeling structural transformations in binary alloys with phase-field crystals. *Phys Rev B* 84:064104
9. Greenwood M, Provatas N, Rottler J (2010) Free energy functionals for efficient phase-field modeling of structural phase transformations. *Phys Rev Lett* 105:045702
10. Toth J, Tegze G, Pustai T, Toth G, Granasy L (2010) Polymorphism, crystal nucleation and growth in the phase-field crystal model in 2D and 3D. *J Phys Condens Matter* 22:364101
11. Yang T, Zhang C, Jing Z, Wei-Ping D, Lin W (2012) Effect of grain boundary on spinodal decomposition using phase-field crystal method. *Chin Phys Lett* 29:078103
12. Mellenthin J, Karma A, Plapp M (2008) Phase-field crystal study of grain boundary pre-melting. *Phys Rev B* 78:184110
13. Berry J, Elder KR, Grant M (2008) Melting at dislocations and grain boundaries: a phase-field crystal study. *Phys Rev B* 77:224114
14. Adland A, Karma A (2013) Phase-field crystal study of grain boundary pre-melting and sheering in bcc iron. *Phys Rev B* 87:024110
15. Greenwood M, Sinclair C, Millitzer M (2012) Phase-field crystal model of solute drag. *Acta Mater* 60:5752
16. Berry J, Provatas N, Rottler J, Sinclair C (2012) Defect stability in phase-field crystal models: stacking faults and partial dislocations. *Phys Rev B* 86:224112
17. Wu K, Voorhees P (2012) Phase-field crystal simulation of nanocrystalline grain growth in two-dimension. *Acta Mater* 60:407
18. Berry J, Grant M (2011) Modeling multiple time scales during glass formation with phase-field crystals. *Phys Rev Lett* 106:175702
19. Fallah V, Korinek A, Ofori-Opoku N, Provatas N, Esmaeili S (2013) Atomistic investigation of clustering phenomenon in the Al–Cu system: three-dimensional phase-field crystal simulation and HRTEM/HRSTEM characterization. *Acta Mater* 61:6372
20. Emmerich H, Lowen H, Wittkowski R, Gruhn T, Toth GT, Tegze G, Granasy L (2012) Phase-field crystal models for condensed matter dynamics on atomic length and diffusive time scales: an overview. *Adv Phys* 61:665
21. Provatas N, Elder KR (2010) Phase-field methods in materials science and engineering. Wiley Publishing. ISBN 978-3-527-40747-7

7.2 Case Study-XVI

Numerical Implementations of Phase-Field Crystal Method

Objectives:

The objective of this case study is to develop a semi-implicit Fourier spectral algorithm to solve phase-field crystal models in two and three dimensions. The algorithm presented in this case study also serves as a template to extent the case studies given in Chap. 5 to three-dimension.

7.2.1 Phase-Field Crystal Model

As discussed in the introduction of this chapter, as in the classical phase-field models, the conserved nondimensional order parameter evolves as:

$$\frac{\partial \psi}{\partial t} = \nabla \cdot M \nabla \frac{\delta F}{\delta \psi} \quad (7.5)$$

in which F is the free energy functional that produces a periodic lattice structure at the ground state in some parameter range. In its simplest form it is given as:

$$F(\psi) = \int \left\{ \epsilon \frac{\psi^2}{2} + \frac{1}{4} \psi^4 + \frac{\psi}{2} (1 + \nabla^2)^2 \psi \right\} dx \quad (7.6)$$

By taking the functional derivative, Eq. 7.5 becomes

$$\frac{\partial \psi}{\partial t} = \nabla^2 M \{ \epsilon \psi + \psi^3 + \psi (1 + 2\nabla^2 + \nabla^4) \} \quad (7.7)$$

$$\frac{\partial \{\psi\}_k}{\partial t} = -k^2 M \{ \epsilon \{\psi\}_k + \{\psi\}_k^3 + \{\psi\}_k (1 - 2k^2 + k^4) \} \quad (7.8)$$

where $\{\cdot\}_k$ is the Fourier transform of the quantity inside the bracket and k is the vector in Fourier space, $k = (k_1, k_2)$, with a magnitude $\sqrt{k_1^2 + k_2^2}$.

$$\frac{\{\psi\}_k^{t+1} - \{\psi\}_k^t}{\Delta t} = -k^2 M \left\{ \epsilon \{\psi\}_k^{t+1} + \left(\{\psi\}_k^3 \right)^t + \{\psi\}_k^{t+1} (1 - 2k^2 + k^4) \right\} \quad (7.9)$$

where Δt is the time increment between time steps $t+1$ and t . By rearranging,

$$\psi^{t+1} = \frac{\{\psi\}_k^t - \Delta t M k^2 \left(\{\psi\}_k^3 \right)^t}{1 + \Delta t M k^2 (\epsilon + 1 - 2k^2 + k^4)} \quad (7.10)$$

and the problem reduces to solving the discretized Eq. 7.10.

The numerical steps to achieve this can be summarized as:

As can be seen, Eq. 7.7 is a parabolic equation, involving first-order time derivative, second-, fourth-, and sixth-order spatial derivatives. The solution of Eq. 7.7 can be provided by any of the three algorithms presented in the previous chapters. Its solution can be found with finite difference techniques, including the adoptive meshing, in [1, 2], with Fourier spectral method in [3] and by utilizing FEM in [4]. In this case study, a semi-implicit Fourier spectral algorithm will be utilized similar to that given in Chap. 5.

7.2.2 Numerical Implementation

For semi-implicit Fourier spectral algorithm, by taking Fourier transform of both sides of Eq. 7.7 the spatial discretization becomes:

By treating the linear terms implicitly and nonlinear term explicitly, the time discretization of Eq. 7.8 becomes:

For number of time steps repeat:

- (1) Fourier transform of current density field ψ and its third power, ψ^3 .
- (2) Calculate the spatial variation of ψ in Fourier space for the time $t+1$ using Eq. 7.10.
- (3) With inverse Fourier transformation bring back the results to real space for the solution at $t+1$.

7.2.3 Results and Discussion

In the simulations given below, the phase diagram shown in Fig. 7.2 is utilized. The simulations were carried in two and three dimensions for both triangular and stripe phases. For all cases the temperature was set to values of -0.25 and the density values for the triangular phase was set to -0.285 and the stripe phase to -0.085 as indicated with the red circles in the phase diagram. The number of grid points was $N_x = N_y = 64$ and grid spacing was taken as $dx = dy = \pi/4$ for the two-dimensional simulations and $N_x = N_y = 32$ and $dx = dy = \pi/4$ for the three-dimensional simulations. For all cases, the initial density field was modulated with a noise term.

The results obtained from two-dimensional simulations are summarized in Fig. 7.3. As can be seen, the microstructure evolves with reduction of the free energy (a and e) for both phases; however, the reduction rates differ significantly hence the time to reach the equilibrium state. As discussed earlier, in the case of the triangular phase, the density fields correspond to the atomic positions in the hexagonal lattice. Owing to the small simulation cell, it appears that the nucleation event is almost spontaneous. For larger simulation cells, it is possible to observe the formation of the polycrystal microstructure and dislocations; in particular, with

higher initial modulated density field and larger undercoolings. On the other hand, the formation of the stripe phase takes place more gradually. The nucleation event starts randomly resulting in the formation of the disconnected stripe phase, as can be seen at the early stages in the figure. Eventually, a fully continuous striped microstructure develops with the absorption and coarsening through the later stages.

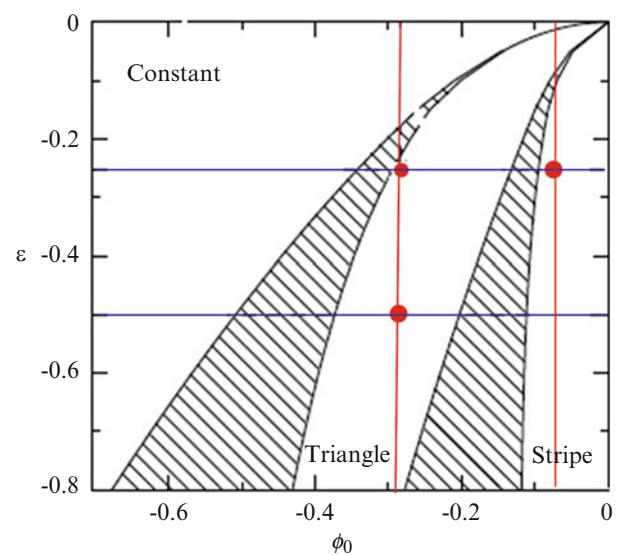
The similar evolution also seen in three-dimension as shown in Fig. 7.4; as expected, for triangular case the resulting lattice structure is body-centered cubic, *bcc*, lattice.

The movie files resulting from these simulations can be found in subdirectory *case_study_16* in downloadable file.

7.2.4 Source Codes

The source code *pfc_2D_v1.m* is in the longhand format and is not optimized, the corresponding code *pfc_2D_v2.m* is the optimized version for the Matlab/Octave. Their counterparts in three-dimension, *pfc_3D_v1.m* is again in longhand format and not optimized and *pfc_3D_v2.m* is the optimized version. These three-dimensional codes also serve as a template for the case studies given in Chap. 5 for their extension to three-dimension.

Fig. 7.2 The PFC model phase diagram used in the simulations. The red circles indicate the density values chosen for triangular and stripe phases



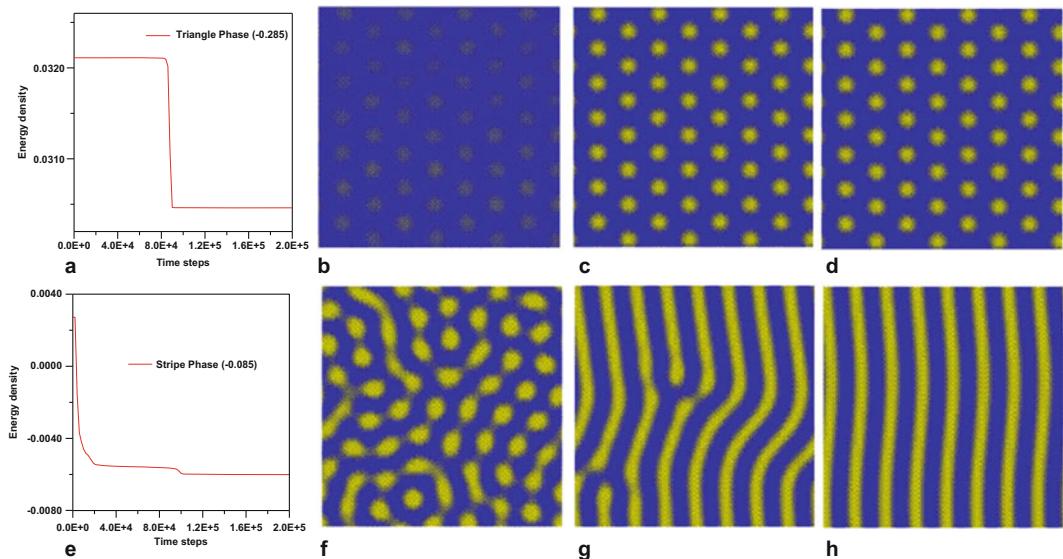
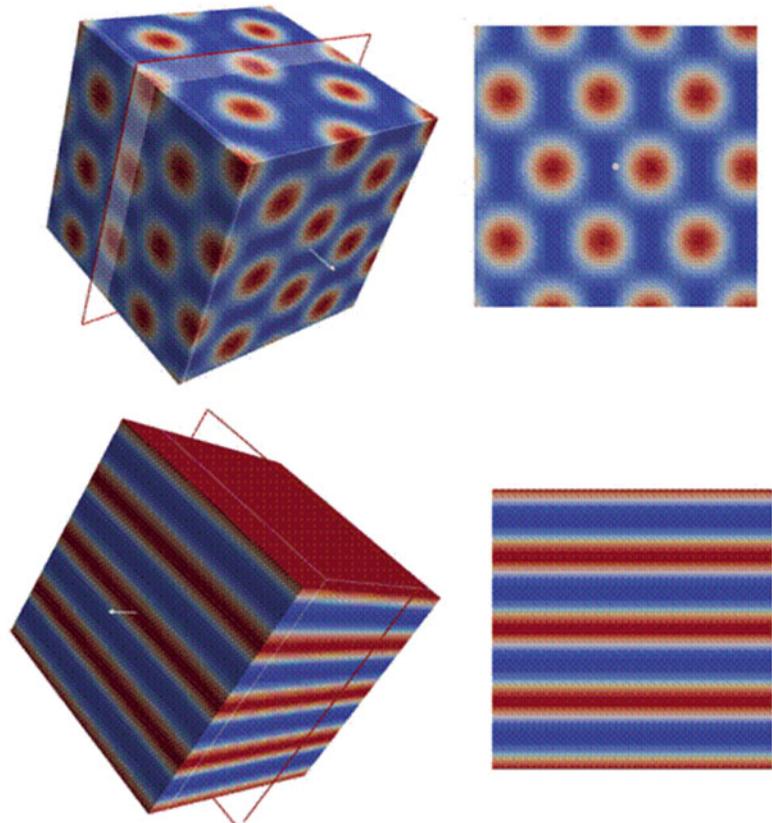


Fig. 7.3 The top row for triangular phase at temperature -0.25 and with average density $\psi_0 = -0.285$. (a) Variation in free energy density. The simulation steps are: (b) 86×10^3 , (c) 90×10^3 , and (d) 200×10^3 . The bottom row for

stripe phase at temperature -0.25 with average density $\psi_0 = -0.085$. (e) Variation in free energy in density. The simulation steps are: (f) 6×10^3 , (g) 60×10^3 , and (h) 200×10^3

Fig. 7.4 The resulting microstructures for triangular and stripe phases from three-dimensional simulations. The cross sections shown on the right are microstructures on the cut-planes shown on left



Program**pfc_2D_v1.m**

This program solves PFC model with semi-implicit Fourier spectral method in two-dimension. It is in longhand format and is not optimized.

The program makes calls to the following functions:

- **prepare_fft.m**
- **write_vtk_grid_values.m**

Listing:

```

1    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2    %                                     %
3    % 2D SEMI-IMPLICIT SPECTRAL %
4    %  PHASE-FIELD CRYSTAL CODE %
5    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6
7    %== get intial wall time:
8    time0=clock();
9    format long;
10
11   out1 = fopen('final_conf.out',
12   'w');
12   out2 = fopen('energy.out','w');
13
14   %-- Simulation cell parameters:
15
16   Nx= 64;
17   Ny= 64;
18
19   pix=4.0*atan(1.0);
20
21   dx= pix/4.0;
22   dy= pix/4.0;
23
24   %
25   %--Time integration parameters:
26   %
27
28   nsteps = 200000;
29   dttime = 0.01;
30   nprint = 2000;
31   nstart = 1;

32   %
33   %-- Material parameters:
34   %
35
36   den0 = -0.285;
37   tempr = -0.25;
38   tempr0 = tempr;
39   noise = den0*(10^-2);

40   %
41   %-----
42   % if infile ==1 read input from
43   file
44   %
45   infile = 0;

46   if (infile == 1)
47
48   %-- open input file
49
50   in1 = fopen('g3_2r.inp','r');

51   %
52   %
53
54   for i=1:Nx
55   for j=1:Ny
56
57   mm1=fscanf(in1,'%5d %5d',
58   [2,1]);
58   mm2=fscanf(in1, '%lf ',[1,1]);
59
60   den(i,j)=mm2(1,1);
61
62   end
63
64   else
65
66   for i = 1:Nx
67   for j = 1:Ny
68
69   den(i,j) = den0 +noise*
70   (0.5-rand);
71
72   end
73
74   end %if
75
76   %
77   %--Prepare fft:
78   %

```

```

79                               125    %
80      [kx,ky,k2,k4] = prepare_fft(Nx, 126    %-- Print results:
81      Ny,dx,dy);                  127    %
82      %--                                128
83      % Evolve                            129    if((mod(istep,nprint)==0) ||
84      %--                                130    (istep ==1))
85      for istep = nstart : nsteps        131    fprintf('done step: %d\n',
86      %tempr = tempr + tempr0/nsteps;     132    istep);
87      f_den =fft2(den);                 133    %energy calculation
88      for i=1:Nx                      134    for i=1:Nx
89      for j=1:Ny                      135    for j=1:Ny
90      for i=1:Nx                      136
91      for j=1:Ny                      137    ss2(i,j) = den(i,j)^2;
92      Linx =-k2(i,j) *(tempr+1.0       138    ss4(i,j) = ss2(i,j)^2;
93      -2.0*k2(i,j)+k4(i,j));          139    end
94      denom(i,j) = 1.0-dtime * Linx;   140    end
95      end                                141
96      end                                142    for i=1:Nx
97      end                                143    for j=1:Ny
98      end                                144
99      f_ff(i,j) = 0.5*f_den(i,j)*      145    f_ff(i,j) = 0.5*f_den(i,j)*
99      (1.0-2.0*k2(i,j)+k4(i,j));      146
100     %--                                147    end
101     for i=1:Nx                      148    end
102     for j=1:Ny                      149    %--
103     end                                150
104     den3(i,j)=den(i,j)^3;           151    ff = real(ifft2(f_ff));
105     end                                152
106     end                                153    %--
107     end                                154
108     end                                155    for i=1:Nx
109     f_den3=fft2(den3);              156    for j=1:Ny
110     %                                157
111     for i=1:Nx                      158    ff(i,j) = ff(i,j)*den(i,j) +
112     for j=1:Ny                      159    0.5*tempr*ss2(i,j)+0.25*ss4
113     Nonx=-k2(i,j) * f_den3(i,j);    160    (i,j);
114     f_den(i,j)=(f_den(i,j) +          161    end
115     dtime*Nonx) /denom(i,j);        162    %
116     end                                163    energ=0.0;
117     end                                164    for i=1:Nx
118     %--                                165    for j=1:Ny
119     end                                166
120     end                                167    energ = energ +ff(i,j);
121     %--                                168
122     den = real(ifft2(f_den));        169    end
123
124

```

```

170    end
171    %
172    energ = energ / (Nx*Ny);
173
174    fprintf(out2, '%d %14.6e\n',
175        istep, energ);
176    write_vtk_grid_values(Nx,Ny,
177        dx,dy,istep,den,ff);
178    end %if
179
180    %-----
181    % Write configuration:
182    %-----
183
184    if(istep == nsteps);
185
186        for i=1:Nx
187            for j=1:Ny
188                fprintf(out1,'%5d %5d %14.
189                    6e\n',i,j,den(i,j));
190            end
191        end
192    end %if
193
194    end % end of time step
195
196    compute_time = etime(clock(), time0);
197    fprintf('Compute Time: %10d\n',
198        compute_time);

```

Line numbers:

8:	Get wall clock time at the beginning of the execution.
11–12:	Assign unit names for output files.
14–23:	Simulation cell parameters.
16:	Number of grid points in the x -direction.
17:	Number of grid points in the y -direction.
19:	The value of π .
21:	The distance between two grid points in the x -direction.
22:	The distance between two grid points in the y -direction
25–32:	Time integration parameters.
28:	Total number of time steps.
29:	Time increment for numerical integration.

30:	Print frequency to output the results to file.
31:	Starting time increment, if a restart is made from a previous execution.
33–40:	Material-specific parameters.
36:	The average density value.
37:	Temperature value.
38:	Initial temperature value.
39:	Noise term for modulation of the initial density field.
48:	If infile=0, a new simulation. If infile=1, restart from a previous execution. In this case, nstart and nsteps values should be set accordingly.
50:	If infile=1, open the input file.
54–62:	Read the density values from the input file.
64–73:	If infile=0, it is a new simulation and modulate the density field with given noise value.
80:	Get the FFT coefficients.
83–194:	Evolve microstructure.
88:	This line is commented out but will be used in the next <i>case studies</i> for reducing the temperature during the annealing process from higher to lower temperature values.
90:	Take current density field from real space to Fourier space (forward FFT transformation).
92–98:	Calculate the value of denominator in Eq. 7.10 for every grid points in the simulation cell.
102–108:	Calculate the nonlinear term, ψ^3 term in Eq. 7.10.
110:	Take the value of ψ^3 from real space to Fourier space (forward FFT transformation).
113–120:	Calculate the value of ψ^{t+1} , Eq. 7.10, at Fourier space.
123:	Bring back the density field values, ψ^{t+1} , from Fourier space to real space (inverse FFT transformation).
129–178:	If print frequency is reached, output the results to file.
132–162:	Calculate the free energy distribution, Eq. 7.6.
163–170:	Integrate the free energy field.
172:	Calculate the average free energy density.
174:	Output the average free energy value to file.
176:	Output the density and energy fields in vtk format to file for contour plots to be viewed by using Paraview.
184–192:	If intermediate configuration files are required for restarts, print the current density field to file.
196–197:	Calculate the total execution time and print it.

Program
pfc_2D_v2.m

This program solves PFC model with semi-implicit Fourier spectral method in two-dimension. It is optimized for Matlab/Octave.

The program makes calls to the following functions:

- **prepare_fft.m**
- **write_vtk_grid_values.m**

Listing:

```

1      %%%%%%%%%%%%%%%%
2      %                         %
3      %  2D SEMI-IMPLICIT SPECTRAL %
4      %  PHASE-FIELD CRYSTAL CODE %
5      % OPTIMIZED FOR MATLAB/OCTAVE %
6      %                         %
7      %%%%%%%%%%%%%%%%
8
9      %== get intial wall time:
10     time0=clock();
11     format long;
12
13     out1 = fopen('final_conf.out',
14                  'w');
14     out2 = fopen('energy.out','w');
15
16     %% Simulation cell parameters:
17
18     Nx= 64;
19     Ny= 64;
20
21     pix=4.0*atan(1.0);
22
23     dx= pix/4.0;
24     dy= pix/4.0;
25
26     %
27     %-Time integration parameters:
28     %
29
30     nsteps = 200000;
31     dtimr = 0.01;
32     nprint = 2000;
33     nstart = 1;

34      %
35      %-- Material parameters:
36      %

37
38      den0 = -0.285;
39      tempr = -0.25;
40      tempr0 = tempr;
41      noise = den0*(10^-2);

42      %-----
43      % if infile ==1 read input from
44      % file
45      %
46      nfile = 0;

47
48      if (infile == 1)
49
50          %--- open input file
51
52          in1 = fopen('g3_2r.inp','r');

53
54      %-----

55
56      for i=1:Nx
57          for j=1:Ny
58
59              mm1=fscanf(in1,'%5d %5d',
60                          [2,1]);
61              mm2=fscanf(in1, '%lf ',[1,1]);
62              den(i,j)=mm2(1,1);
63          end
64      end

65
66      else
67
68          for i = 1:Nx
69              for j = 1:Ny
70
71                  den(i,j) = den0 +noise*
72                      (0.5-rand);
73
74          end
75      end

76      end %if
77
78      %
79      %--Prepare fft:
80      %

```

```

81                                         124
82 [kx,ky,k2,k4] = prepare_fft      125     energ = sum(ff(:))/(Nx*Ny);
83 (Nx,Ny,dx,dy);                  126
84 %--                                127     fprintf(out2, '%d %14.6e\n',
85 % Evolve                           128     istep,energ);
86 %--                                129     write_vtk_grid_values(Nx,Ny,
87                               dx,dy,istep,den,ff);
88 for istep = nstart : nsteps      130
89                                         131     end %if
90 %tempr = tempr + tempr0/nsteps;  132
91                                         133     %-----
92 f_den =fft2(den);               134     % Write configuration:
93                                         135     %-----
94 Linx=-k2.* (tempr+1.0-2.0*      136
95 k2+k4);                         137     if(istep == nsteps);
96 denom = 1.0-dtime.*Linx;        138
97 den3=den.^3;                   139     for i=1:Nx
98 f_den3=fft2(den3);             140     for j=1:Ny
99                                         141     fprintf(out1,'%5d %5d %14.6e
100                                         \n',i,j,den(i,j));
101 Nonx=-k2.*f_den3;              142     end
102                                         143     end
103 f_den=(f_den + dtime*Nonx) .  144
104 /denom;                         145     end %if
105                                         146
106                                         147     end % end of time step
107 %                                         148
108 %-- Print results:              149     compute_time = etime(clock(),
109                                         150     time0);
110 if((mod(istep,nprint) == 0) ||  | Line numbers:
111 (istep == 1) )                  9–10:  Get wall clock time at the beginning of the
112 fprintf('done step: %5d\n',       execution.
113 istep);                         13–14: Assign unit names for output files.
114 %energy calculation            16–25: Simulation cell parameters.
115                                         18: Number of grid points in the  $x$ -direction.
116 ss2 = den.^2;                   19: Number of grid points in the  $y$ -direction.
117 ss4 = ss2.^2;                   21: The value of  $\pi$ .
118                                         23: The distance between two grid points in the
119 f_ff = 0.5*f_den.* (1.0-2.0*    x-direction.
120 k2+k4);                         24: The distance between two grid points in the
121 ff = real(ifft2(f_ff));          y-direction
122                                         27–34: Time integration parameters.
123 ff = ff.*den + 0.5*tempr*ss2+  30: Total number of time steps.
                                         31: Time increment for numerical integration.
                                         32: Print frequency to output the results to file.
0.25*ss4;

```

(continued)

33:	Starting time increment, if a restart is made from a previous execution.
35–42:	Material-specific parameters.
38:	The average density value.
39:	Temperature value.
40:	Initial temperature value.
41:	Noise term for modulation of the initial density field.
48:	If infile=0, a new simulation. If infile=1, restart from a previous execution. In this case, nstart and nsteps values should be set accordingly.
52:	If infile=1, open the input file.
56–64:	Read the density values from the input file.
68–74:	If infile=0, it is a new simulation and modulate the density field with given noise value.
82:	Get the FFT coefficients.
85–147:	Evolve microstructure.
90:	This line is commented out but will be used in the next <i>case studies</i> for reducing the temperature during the annealing process from higher to lower temperature values.
92:	Take current density field from real space to Fourier space (forward FFT transformation).
94–95:	Calculate the value of denominator in Eq. 7.10 for every grid points in the simulation cell.
97:	Calculate the nonlinear term, ψ^3 term in Eq. 7.10.
99:	Take the value of ψ^3 from real space to Fourier space (forward FFT transformation).
101–103:	Calculate the value of ψ^{t+1} , Eq. 7.10, at Fourier space.
105:	Bring back the density field values, ψ^{t+1} , from Fourier space to real space (inverse FFT transformation).
110–145:	If print frequency is reached, output the results to file.
114–123:	Calculate the free energy distribution, Eq. 7.6.
125:	Integrate the free energy field and averaged it.
127:	Output the average free energy value to file.
129:	Output the density and energy fields in vtk format to file for contour plots to be viewed by using Paraview.
137–145:	If intermediate configuration files are required for restarts, print the current density field to file.
149–150:	Calculate the total execution time and print it.

Program

pfc_3D_v1.m

This program solves PFC model with semi-implicit Fourier spectral method in three-dimension. It is in longhand format and is not optimized.

The program makes calls to the following functions:

- **prepare_fft_3d.m**
- **write_vtk_grid_values_3D.m**

Listing:

```

1      %%%%%%%%
2      %
3      % 3D SEMI-IMPLICIT SPECTRAL %
4      % PHASE-FIELD CRYSTAL CODE %
5      %
6      %== get intial wall time:
7      time0=clock();
8      format long;
10
11     out1 = fopen('final_conf.out',
12         'w');
12     out2 =fopen('energy.out', 'w');
13
14     %
15     %%Simulation cell parameters:
16     %
17
18     Nx= 32;
19     Ny= 32;
20     Nz= 32;
21
22     pix=4.0*atan(1.0);
23
24     dx= pix/4.0;
25     dy= pix/4.0;
26     dz= pix/4.0;
27
28     %
29     %%Time integration parameters:
30     %
31

```

```
32      nsteps = 200000;          79      end
33      nprint = 5000;           80      end
34      dtme = 0.05;            81      end
35
36      %                         82
37      %% Material Parameters:   83      for i=1:Nx
38      %                         84      for j=1:Ny
39
40      den0 = -0.085;           85      for k=1:Nz
41      tempr = -0.25;
42      noise = den0*(10^-2);
43
44      %
45      %% Initialize density:    91      end
46      %
47
48      for i = 1:Nx             94      f_den3=fftn(den3);
49      for j = 1:Ny             95
50      for k = 1:Nz             96      %%-
51
52      den(i,j,k) = den0 +noise*(0.5
53      -rand);                 97      for i=1:Nx
54      end                       98      for j=1:Ny
55      end                       99      for k=1:Nz
56      end
57
58      %
59      %%Prepare fft:           101
60      %
61
62      [kx,ky,kz,k2,k4]=prepare_fft_
63      3d(Nx,Ny,Nz,dx,dy,dz);   102      Nonx=-k2(i,j,k)*f_den3(i,j,k);
64      %%                         103      f_den(i,j,k)=(f_den(i,j,k)
65      % Evolve                  104      + dtme*Nonx) /denom(i,j,k);
66      %%                         105      end
67
68      for istep = 1:nsteps     106      end
69
70      f_den =fftn(den);        107      end
71
72      for i=1:Nx               108      %%-
73      for j=1:Ny               109
74      for k=1:Nz               110      den = real(ifftn(f_den));
75
76      Linx=-k2(i,j,k)*(tempr+1.0-2.0
77      *k2(i,j,k)+k4(i,j,k));   111
78      denom(i,j,k)=1.0-dtme.*Linx; 112      %
79      ss2(i,j,k) = den(i,j,k)^2;
```

```

127     ss4(i,j,k) = ss2(i,j,k)^2;           173
128
129     end
130     end
131     end
132
133     for i=1:Nx
134         for j=1:Ny
135             for k=1:Nz
136
137                 f_ff(i,j,k)= 0.5*f_den(i,j,k)*
138                     (1.0-2.0*k2(i,j,k)+k4(i,j,k));
139
140     end
141     end
142     %--
143
144     ff = real(ifftn(f_ff));
145
146     %--
147
148     for i=1:Nx
149         for j=1:Ny
150             for k=1:Ny
151
152                 ff(i,j,k)=ff(i,j,k)*den(i,j,k)
153                     + 0.5*tempr*ss2(i,j,k)
154                     + 0.25*ss4(i,j,k);
155
156     end
157
158     energ = 0.0;
159
160     for i=1:Nx
161         for j=1:Ny
162             for k=1:Nz
163
164                 energ = energ + ff(i,j,k);
165
166     end
167
168     end
169
170     energ =energ / (Nx*Ny*Nz);
171
172     fprintf(out2, '%d %14.6e\n',
173             istep,energ);
174
175
176     end
177
178     %-----
179
180     if(istep == nsteps);
181
182         for i=1:Nx
183             for j=1:Ny
184                 for k=1:Nz
185                     fprintf(out1,'%5d %5d %5d
186                         %14.6e\n',i,j,k,den(i,j,k));
187
188         end
189     end %if
190
191     end % end of time step
192
193     compute_time = etime(clock(),
194                           time0);
195     fprintf('Compute Time: %10d\n',
196            compute_time);

```

Line numbers:

8:	Get wall clock time at the beginning of execution.
11–12:	Assign unit names for the output files.
15–27:	Simulation cell parameters.
18:	Number of grid points in the <i>x</i> -direction.
19:	Number of grid points in the <i>y</i> -direction.
20:	Number of grid points in the <i>z</i> -direction.
22:	The value of π .
24:	The distance between two grid points in the <i>x</i> -direction.
25:	The distance between two grid points in the <i>y</i> -direction.
26:	The distance between two grid points in the <i>z</i> -direction.
29–35:	Time integration parameters.
32:	Total number of time steps.
33:	Print frequency to output the results to file.
34:	Time increment for numerical integration.
37–43:	Material Parameters.
40:	Average density.
41:	Temperature.
42:	Noise term to modulate the initial density field.

(continued)

45–57:	Modulate the density field with given noise term.
62:	Get FFT coefficients.
65–191:	Evolve microstructure.
70:	Take current density field from real space to Fourier space (forward FFT transformation)
72–81:	Calculate the value of denominator in Eq. 7.10 at every grid points.
83–91:	Calculate the nonlinear term, ψ^3 , in Eq. 7.10
94:	Take the values of ψ^3 from real space to Fourier space (forward FFT transformation)
98–107:	Calculate the value of ψ^{t+1} , in Eq. 7.10, at Fourier space.
110:	Bring back the values of ψ^{t+1} from Fourier space to real space (inverse FFT transformation).
116–176:	If print frequency is reached, output the results to file.
120–157:	Calculate the free energy distribution, Eq. 7.6.
158–168:	Integrate the free energy field.
170:	Average free energy density.
172:	Print the average free energy density value to file.
174:	Output the results in vtk file format for contour plots to be viewed by using Paraview.
180–189:	If intermediate configuration files are required, print the density field to file.
193–194:	Calculate the execution time and print it.

Program

pfc 3D v2.m

This program solves PFC model with semi-implicit Fourier spectral method in three dimension. It is optimized for Matlab/Octave.

The program makes calls to the following functions:

- `prepare_fft_3d.m`
 - `write vtk grid values 3D.m`

Listing:

```
1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 %
3 % 3D SEMI-IMPLICIT SPECTRAL %
4 % PHASE-FIELD CRYSTAL CODE %
5 % OPTIMIZED FOR MATLAB/OCTAVE %
6 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```

56    end
57    end
58
59    %
60    %--Prepare fft:
61    %
62
63    [kx,ky,kz,k2,k4] = prepare_
64      fft_3d(Nx,Ny,Nz,dx,dy,dz);
65
66    %-- Evolve
67    %
68
69    for istep = 1:nsteps
70
71      f_den = ifftn(den);
72
73      Linx=-k2.* (tempr+1.0-2.0*k2+k4);
74      denom = 1.0-dtime.*Linx;
75
76      den3=den.^3;
77      f_den3=ifftn(den3);
78      Nonx=-k2.*f_den3;
79
80      f_den=(f_den + dtime*Nonx) .
81      /denom;
82
83      den = real(ifftn(f_den));
84
85    %-- Print results:
86    %
87
88    if(mod(istep,nprint) == 0 )
89
90      fprintf('done step: %5d\n',
91      istep);
92
93    %energy calculation
94
95    ss2 = den.^2;
96    ss4 = ss2.^2;
97
98    f_ff = 0.5*f_den.* (1.0-2.0*
99      k2+k4);
100
101   ff = real(ifftn(f_ff));
102
103   energ1 = sum(ff( :,1));
104   energ2 = sum(ff( :,2));
105   energ3 = sum(ff( :,3));
106
107   energ =(energ1+energ2+energ3)/
108     (Nx*Ny*Nz);
109
110   fprintf(out2, '%d %14.6e\n',
111   istep,energ);
112
113   end
114
115   %-----
116
117   if(istep == nsteps);
118
119     for i=1:Nx
120       for j=1:Ny
121         for k=1:Nz
122           fprintf(out1,'%5d %5d %5d
123             %14.6e\n',i,j,k,den(i,j,k));
124
125     end
126
127   end %if
128
129
130   compute_time = etime(clock(),
131   time0);
132
133   fprintf('Compute Time: %10d
134   \n',compute_time);

```

Line numbers:

8–9:	Get wall clock time at the beginning of execution.
12–13:	Assign unit names for the output files.
16–28:	Simulation cell parameters.
19:	Number of grid points in the <i>x</i> -direction.
20:	Number of grid points in the <i>y</i> -direction.
21:	Number of grid points in the <i>z</i> -direction.
23:	The value of π .
25:	The distance between two grid points in the <i>x</i> -direction.
26:	The distance between two grid points in the <i>y</i> -direction.

(continued)

27:	The distance between two grid points in the z -direction.
30–36:	Time integration parameters.
33:	Total number of time steps.
34:	Print frequency to output the results to file.
35:	Time increment for numerical integration.
38–44:	Material parameters.
41:	Average density.
42:	Temperature.
43:	Noise term to modulate the initial density field.
46–58:	Modulate the density field with given noise term.
63:	Get FFT coefficients.
66–126:	Evolve microstructure.
71:	Take current density field from real space to Fourier space (forward FFT transformation).
73–74:	Calculate the value of denominator in Eq. 7.10 at every grid points.
76:	Calculate the nonlinear term, ψ^3 , in Eq. 7.10.
77:	Take the values of ψ^3 from real space to Fourier space (forward FFT transformation).
78–80:	Calculate the value of ψ^{t+1} , in Eq. 7.10, at Fourier space.
82:	Bring back the values of ψ^{t+1} from Fourier space to real space (inverse FFT transformation).
88–113:	If print frequency is reached, output the results to file.
92–101:	Calculate the free energy distribution, Eq. 7.6.
103–105:	Integrate the free energy field.
107:	Average free energy density.
109:	Print the average free energy density value to file.
111:	Output the results in vtk file format for contour plots to be viewed by using Paraview.
117–126:	If intermediate configuration files are required, print the density field to file.
130–131:	Calculate the execution time and print it.

Function

prepare_fft_3d.m

This function calculates the three-dimensional coefficients of FFT. It is very similar to its two-dimensional counterpart *prepare_fft.m*; therefore, just its listing is given in here.

Listing:

```

1   function [kx,ky,kz,k2,k4] =
2     prepare_fft_3d(Nx,Ny,Nz,dx,
3       dy,dz)
4
5   format long;
6
7   Nx21 = Nx/2 + 1;
8   Ny21 = Ny/2 + 1;
9   Nz21 = Nz/2 + 1;
10
11  Nx2=Nx+2;
12  Ny2=Ny+2;
13  Nz2=Nz+2;
14
15  %--
16  delkx=(2.0*pi)/(Nx*dx);
17  delky=(2.0*pi)/(Ny*dy);
18  delkz=(2.0*pi)/(Nz*dz);
19
20  %--
21  for i=1:Nx21
22    fk1=(i-1)*delkx;
23    kx(i)=fk1;
24    kx(Nx2-i)=-fk1;
25  end
26
27  for j=1:Ny21
28    fk2=(j-1)*delky;
29    ky(j)=fk2;
30    ky(Ny2-j)=-fk2;
31  end
32
33  for k=1:Nz21
34    fk3=(k-1)*delkz;
35    kz(k)=fk3;
36    kz(Nz2-k)=-fk3;
37  end
38
39  %---
40
41  for i=1:Nx
42    for j=1:Ny
43      for k=1:Nz

```

```

45 k2(i,j,k)=kx(i)^2+ky(j)^2
46 +kz(k)^2;
47 end
48 end
49 end
50
51 %%--
52
53 k4 = k2.^2;
54
55 end %endfunction
21
22
23
24 for i = 1:nx
25 for j = 1:ny
26 for k = 1:nz
27
28 x =(i-1)*dx;
29 y =(j-1)*dy;
30 z =(k-1)*dz;

```

Function

write vtk grid values 3D.m

This function writes the output values in vtk format for contour plots to be viewed by using Paraview. It is very similar to its two-dimensional counterpart *write_vtk_grid_values.m*; therefore, just its listing is given in here.

Listing:

```

1 function [ ]= write_vtk_
grid_values_3D(nx,ny,nz,dx,
dy,dz,istep,data1)
2
3 %-- open output file
4
5 fname=sprintf('time_%d.vtk',
istep);
6 out =fopen(fname,'w');
7
8 npoin =nx*ny*nz;
9
10 % start writing ASCII VTK file:
11
12 % header of VTK file
13
14 fprintf(out,'# vtk DataFile
Version 2.0\n');
15 fprintf(out,'time_10.vtk\n');
16 fprintf(out,'ASCII\n');
17 fprintf(out,'DATASET STRUCTURED
_GRID\n');
18
19 %--- coords of grid points:
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42 fprintf(out,' SCALARS DEN float 1
\n');
43
44 fprintf(out,' LOOKUP_TABLE
default\n');
45
46 for i = 1:nx
47 for j = 1:ny
48 for k = 1:nz
49 ii=(i-1)*nx+j;
50
51 fprintf(out,'%14.6e\n',data1
(i,j,k));
52
53 end
54 end
55 end
56
57 %fprintf(out,' SCALARS ENEG
float 1\n');
58
59 %fprintf(out,' LOOKUP_TABLE
default\n');
60
61 %for i = 1:nx

```

```

62      %for j = 1:ny
63      %for k = 1:Nz
64      %ii=(i-1)*nx+j;
65
66      %fprintf(out,'%14.6e\n',data2
67      %    (i,j,k));
68      %end
69      %end
70
71      fclose(out);
72
73  end %endfunction

```

theoretically and experimentally. In the previous chapters, it is established that the grain boundaries moves normal to itself with a velocity that is proportional to its mean curvature without detailing the atomic structure of grain boundaries and the atomic processes taking place during their motion. The PFC modeling can offer an alternative simulation technique to classical molecular dynamics, MD, simulations, to elucidate such atomic processes. Such a PFC modeling of grain growth in nanocrystalline materials is detailed in [1].

References

1. Hu Z, Wise SM, Lowengrub JS (2009) Stable and efficient finite-difference nonlinear-multigrid schemes for the phase-field crystal equation. *J Comp Phys* 228:5323
2. Baskaran A, Hu Z, Lowengrub JS, Wang C, Wise SM, Zhou P (2013) Energy stable and efficient finite-difference nonlinear multigrid schemes for the modified phase-field crystal equation. *J Comp Phys* 250:270
3. Cheng M, Warren JA (2008) An efficient algorithm for solving phase-field crystal model. *J Comp Phys* 227:6241
4. Galenko PK, Gomez H, Kropotin NV, Elder KR (2013) Unconditionally stable method and numerical solution of the hyperbolic phase-field crystal equation. *Phys Rev E* 88:013310

7.3 Case Study-XVII

Phase-Field Crystal Modeling of Grain Growth

Objective:

The objective of this case study is to introduce a semi-implicit Fourier spectral method to study the microstructure evolution in polycrystals with phase-field crystal model.

7.3.1 Background

Owing to their importance in determining the mechanical and physical properties of polycrystalline materials, grain boundaries and their evolution has been extensively investigated both

7.3.2 Phase-Field Crystal Model

In this case study, the grain growth in a bicrystal simulated with the classical PFC model in which the kinetic equation and free energy functional are described by

$$\frac{\partial \psi}{\partial t} = \nabla \cdot M \nabla \frac{\delta F}{\delta \psi} \quad (7.11)$$

and

$$F(\psi) = \int \left\{ \epsilon \frac{\psi^2}{2} + \frac{1}{4} \psi^4 + \frac{\psi}{2} (1 + \nabla^2)^2 \psi \right\} dx \quad (7.12)$$

7.3.3 Numerical Implementation

Although, a direct undercooling from a given average density field in PFC model automatically produces a polycrystalline microstructure; however, the resulting microstructures are usually random and also contain the other lattice defects such as dislocations. To generate a controlled polycrystalline microstructure requires a little computational effort. The program *pfc_poly_v1.m* given in this section serves this purpose. Although the program is set for bicrystal morphology, it can be easily extended to other desired polycrystal configurations.

After forming the bicrystal, a spherical grain embedded in another grain; then, the grain growth simulations were performed, without

any modifications, with the codes *pfc_2D_v1.m* and *pfc_2D_v2.m* which are described in the previous Case Study XVI.

7.3.4 Results and Discussion

The simulations were carried out for the average density of $\psi_0 = -0.285$ at temperatures $\varepsilon = -0.25$ as indicated with the red circle in the phase diagram shown in Fig. 7.2.

The initial bicrystal configuration was prepared with the program *pfc_poly_v1.m*. For a simulation cell containing $N_x = N_y = 512$ grid points, this is shown in Fig. 7.5. The relative misorientation of the two grains (a and b) was 7.5° in the bicrystal configuration.

In the figure, the dislocations residing at the grain boundaries resulting from the mismatch between the two crystals can be easily discerned. The evolution of the microstructure with further simulation steps is summarized in Fig. 7.6. As can be seen, the spherical grain shrinks with time as in the previous case studies given in earlier chapters. The provided movie in subdirectory *case_study_17* in downloadable file clearly shows the rotation of the spherical grain, faceting and the motion of dislocations and their mutual annihilations in this shrinking

process. The detailed analysis of the events taking place during this grain boundary motion, based on the similar PFC simulations, can be found in [1].

7.3.5 Source Codes

The grain growth simulations were performed with the codes *pfc_2D_v1.m* and *pfc_2D_v2.m* which are described in the previous Case Study XVI without any modifications. The code for the preparation of the bicrystal configuration is given below. This program will also be used in the next case study for deformation behavior of another bicrystal morphology by setting the value of parameter *bicrystal* in line 17 in the program. Although the program is for bicrystals, however, it serves as template for generation of other desired polycrystal configurations.

Program

pfc_poly_v1.m

This program prepares the simulation cell for bicrystal configurations. It also serves as template for generation of polycrystal configurations.

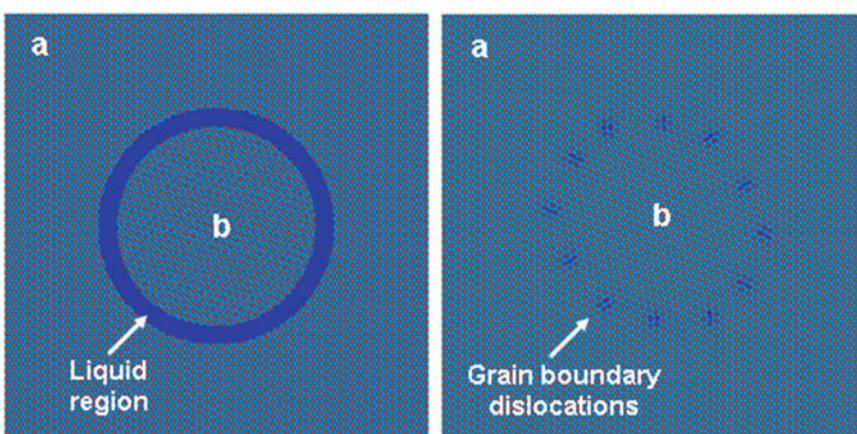


Fig. 7.5 Preparation of spherical bicrystal configuration. Left shows the initial seeding and right shows the formation of the grain boundary between two crystals a and b after 50×10^3 simulation steps

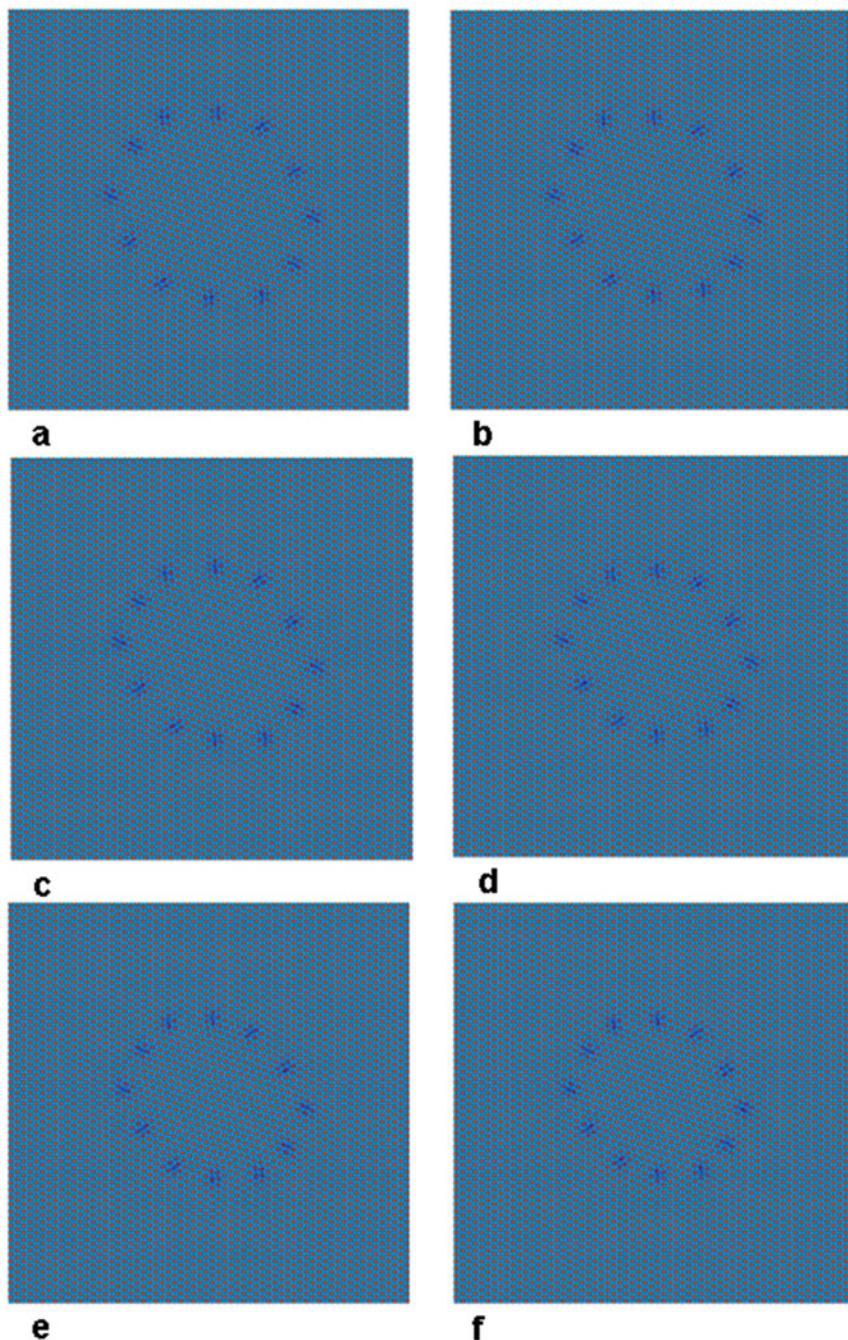


Fig. 7.6 Time evolution of bicrystal morphology. The simulation steps are: (a) 5×10^3 , (b) 50×10^3 , (c) 100×10^3 , (d) 150×10^3 , (e) 200×10^3 , and (f) 250×10^3

Listing:

```

1   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2   % Prepare PFC bi-crystal %
3   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
4
5   in=fopen('final_conf.out','r');
6   = fopen('bi_1r.inp','w');
7
8   Nx = 64;
9   Ny = 64;
10
11  pix =4.0*atan(1.0);
12
13  dx=pix/4.0;
14  dy=pix/4.0;
15  %
16  con=-0.285;
17  bicrystal=1;
18
19  %---
20  %- read input file
21  %---
22
23  for i=1:Nx
24  for j=1:Ny
25
26  mm1=fscanf(in,'%5d %5d',
27  [2,1]);
28  mm2=fscanf(in, '%lf ',[1,1]);
29  den0(i,j)=mm2(1,1);
30  end
31  end
32
33  %-----
34  % replacate 64x64 grid to
35  %----- 512x512 grid
36  for k=1:7
37  for i=1:Nx
38  ii=Nx*k+i;
39  for m=1:7
40  for j=1:Ny
41  jj=m*Ny+j;
42
43  den0(ii,jj)=den0(i,j);
44
45  end
46  end
47  end
48  end
49
50  for i=1:Nx
51  for m=1:7
52  for j=1:Ny
53  jj=m*Ny+j;
54
55  den0(i,jj)=den0(i,j);
56
57  end
58  end
59  end
60
61  for k=1:7
62  for i=1:Nx
63  ii=Nx*k+i;
64  for j=1:Ny
65
66  den0(ii,j)=den0(i,j);
67
68  end
69  end
70  end
71
72  %-----
73  % Generate bicrystal
74  %-----
75
76  for i=1:512
77  for j=1:512
78
79  den(i,j)=con;
80
81  end
82  end
83
84
85  if(bicrystal == 1)
86
87  %-- liquid region:
88
89  radius = 142.0;
90
91  for i=1:512
92  for j=1:512
93  den(i,j)=den0(i,j);
94
95  xlength = sqrt((i-256.0)^2
+ (j-256.0)^2);

```

```

96
97 if(xlength <= radius)
98     den(i,j) = con;
99 end
100
101 end
102 end
103
104 %-- Rotate second crystal:
105 theta =15.0*pix/180.0;
106
107 tmatx(1,1)= cos(theta);
108 tmatx(1,2)= sin(theta);
109 tmatx(2,1)=-sin(theta);
110 tmatx(2,2)= cos(theta);
111
112 for i=1:512
113 for j=1:512
114 den1(i,j)=con;
115
116 vect(1)=i;
117 vect(2)=j;
118
119 for ii=1:2
120 vect2(ii)=0.0;
121 for jj=1:2
122 vect2(ii)=vect2(ii)+tmatx(ii,
123 jj)*vect(jj);
124 end
125 end
126
127 ix=int16(vect2(1));
128 iy=int16(vect2(2));
129
130 if(ix < 1)
131     ix=ix+512;
132 end
133
134 if(ix > 512)
135     ix=ix-512;
136 end
137
138 if(iy < 1)
139     iy=iy+512;
140 end
141
142 if(iy > 512)
143
144
145 iy=iy-512;
146 end
147
148 den1(i,j)=den0(ix,iy);
149
150 end
151 end
152
153 %
154 %-- size of second crystal:
155 radius=120.0;
156
157 for i=1:512
158 for j=1:512
159 xlength=sqrt((i-256.0)^2
160 +(j-256.0)^2);
161
162 if(xlength <= radius)
163 den(i,j)=den1(i,j);
164 end
165
166 end
167 end
168
169 end % if
170
171 %---
172
173 if(bicrystal == 2)
174
175 %liquid region:
176
177 for i=1:512
178 for j=1:512
179
180 den(i,j)=den0(i,j);
181
182 if((i >= 120) && (i <= 392))
183     den(i,j)=con;
184 end
185
186 end
187 end
188
189 % second crystal
190
191 %-- Rotate second crystal:
192
193 theta = 30.0*pix/180.0;

```

```

194                               244
195      tmatx(1,1)= cos(theta);    245      den(i,j)=den1(i,j);
196      tmatx(1,2)= sin(theta);    246
197      tmatx(2,1)=-sin(theta);   247      end
198      tmatx(2,2)= cos(theta);   248      end
199                               249
200      for i=1:512              250      end %if
201      for j=1:512              251
202                                252      %--- print output:
203      den1(i,j)=con;          253
204                                254      for i=1:512
205      vect(1)=i;              255      for j=1:512
206      vect(2)=j;              256
207                                257      fprintf(out,'%5d %5d %14.6e\n',
208      for ii=1:2                i,j,den(i,j));
209      vect2(ii)=0.0;           258
210      for jj=1:2                259      end
211                                260      end
212      vect2(ii)=vect2(ii)+tmatx
213      (ii,jj)*vect(jj);       261      %% Graphic output:
214      end                         262
215      end                         263
216      ix=int16(vect2(1));       264      istep=1;
217      iy=int16(vect2(2));       265      Nx1=512;
218                                266      Ny1=512;
219      if(ix < 1)               267      data2=zeros(Nx1,Ny1);
220          ix=ix+512;           268
221      end                         269      write_vtk_grid_values(Nx1,Ny1,
222                                270      dx,dy,istep,den,data2);
223      if(ix > 512)
224          ix=ix-512;
225      end
226
227      if(iy < 1)
228          iy=iy+512;
229      end
230
231      if(iy > 512)
232          iy=iy-512;
233      end
234
235      den1(i,j)=den0(ix,iy);
236
237      end
238      end
239
240      %% size of the second grain:
241      for i=136:376
242          for j=1:512

```

Line numbers:

5–6:	Assign unit names for input and output files.
8–9:	Number of grid points in the <i>x</i> - and <i>y</i> -directions for the data in the input file.
11:	The value of π .
13:	Spacing between two grid points in the <i>x</i> -direction.
14:	Spacing between two grid points in the <i>y</i> -direction.
16:	Average value of the density.
17:	If bicrystal=1, generate spherical bicrystal configuration. If bicrystal=2, generate layered bicrystal configuration.
19–32:	Read input file for triangular phase generated on $N_x = N_y = 64$ simulation cell.
34–71:	Replicate 64×64 triangular phase to 512×512 simulation cell.
76–82:	Initialize another array with average density.
85–169:	If bicrystal=1, generate spherical bicrystal configuration.
87–103:	Introduce a circular liquid region in the first crystal.

(continued)

104–151:	Rotate second grain with the amount given by the value of theta.
154–168:	Cut a circular seed from the rotated second crystal and insert into the liquid region of the simulation cell
173–250:	If bicrystal=2, generate layered bicrystal configuration.
175–187:	Prepare a rectangular liquid region in the simulation cell.
191–238:	Rotate second crystal with the amount given by the value of theta.
240–248:	Cut a rectangular region from the rotated second crystal and insert into the liquid region of the simulation cell.
252–260:	Output the new simulation configuration to file.
262–269:	Output the results in vtk file format to be viewed by using Paraview.

Reference

- Wu KA, Voorhees PW (2012) Phase-field crystal simulations of nanocrystalline grain growth in two dimensions. *Acta Mater* 60:407.

7.4 Case Study-XVIII

Phase-Field Crystal Modeling of Deformation Behavior of a Bicrystal

Objectives:

The objective of this case study is to develop simple but efficient semi-implicit Fourier spectral algorithm for deformation studies with phase-field crystal modeling.

7.4.1 Background

The plastic deformation of crystalline solids proceeds with the nucleation of dislocations (line defects), their motion and collective and cooperative behavior with each other and with other lattice defects such as grain boundaries and vacancies and solute atoms. At mesoscale, the phase-field models have been developed by considering the long-range stress fields of

dislocations and their short-range interactions (annihilation, junction formation, and climb) [1–7]. Of course, molecular dynamic simulations, MD, provide more detailed information regarding atomic processes taking place resulting from the evolution of dislocations. However, they are limited in spatial and temporal scales, and more often, are carried for only very high stress regime (GPAs) and strain rates (10^7 s).

As seen in the previous case study, PFC model naturally accounts for grain boundaries, dislocations and their motions, glide and climb, in much longer diffusive time scales than the ones currently possible with MD simulations. In order to bring the PFC simulations more closely to the deformation experiments and the underlying physics, the original PFC formulism was modified in [8, 9] and they termed this modified model as MPFC. In the original model the evolution equation was taken as:

$$\frac{\partial \psi}{\partial t} = \nabla \cdot M \nabla \frac{\delta F}{\delta \psi} \quad (7.13)$$

in which F is the free energy functional that produces a periodic lattice structure at the ground state in some parameter range. The kinetic equation above relaxes all disturbances (elastic and plastic) diffusively. However, the elastic strains relax considerably much faster rates compared to phenomena that evolve on diffusive time scales. In the MPFC model, in order to take these effects into consideration, two time scales are introduced into the kinetic evolution equation as:

$$\frac{\partial^2 \psi}{\partial t^2} + \beta \frac{\partial \psi}{\partial t} = \alpha^2 \nabla \cdot M \nabla \frac{\delta F}{\delta \psi} \quad (7.14)$$

By appropriately choosing the effective sound speed α and effective diffusion coefficient β , a finite elastic interaction length and time can be established. The density waves propagate effectively undamped over this elastic interaction time and distance; and beyond the density evolution becomes diffusive. The model is successfully applied to deformation of crystalline of solids in [8–10] and detailed dislocation reactions in [11]. Efficient solution methodologies of the above wave equation can be found in [9, 12].

Another approach for deformation studies by retaining the original formulism of PFC model is introduced in [13, 14] which preserves the constant volume condition. This model is implemented for this case study as given below.

7.4.2 Numerical Implementation

The numerical scheme while preserving the original PFC model in terms of evolution equation Eq. 7.13 can be explained with Fig. 7.7 given below.

To produce the deformation in any given Cartesian coordinate directions, the grid size of the simulation cell is altered with chosen strain rate $\dot{\epsilon}$, while preserving the constant volume, thus

$$V = dx dy = dx' dy' = (dx + \dot{\epsilon} \Delta t) dy' \quad (7.15)$$

Owing to a constant deformation rate applied to all the atomic positions, the deformation state in

this case is affine. The numerical treatment of Eq. 7.15 is very simple, just requiring addition of a few lines to the original codes *pfc_2D_v1.m* and *pfc_2D_v2.m* given in [Case Study XVI](#).

7.4.3 Results and Discussion

Two simulations were carried for the average density of $\psi_0 = -0.285$ at temperatures $\varepsilon = -0.25$ (input file: *bi_2r.inp*) and $\varepsilon = -0.5$ (input file: *bi_3r.inp*), as indicated with the red circles in the phase diagram shown in Fig. 7.2. The initial bicrystal configuration was prepared with the program *pfc_poly_v1.m* given in the previous case study. For this configuration, in line 17 *bicrystal=2* in the program. For a simulation cell containing $N_x = N_y = 512$ grid points, this is shown in Fig. 7.8. The figure also summarizes the steps to prepare the microstructure before the deformation simulations. The relative

Fig. 7.7 Numerical scheme for PFC deformation simulation

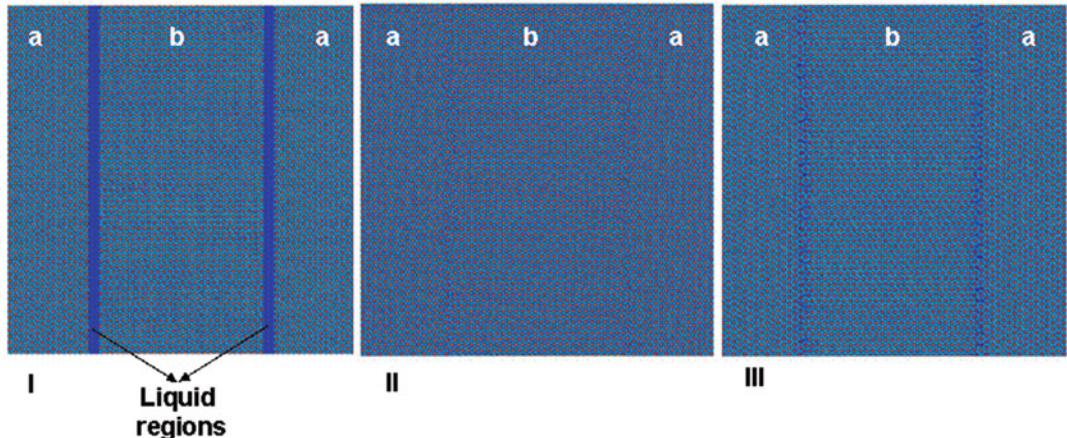
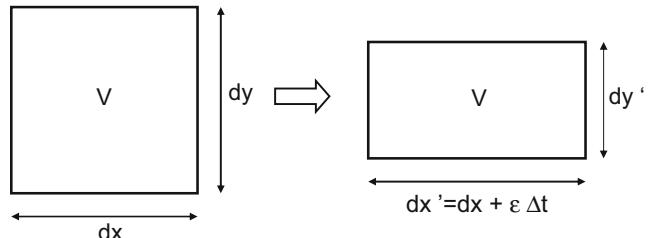


Fig. 7.8 The preparation of bicrystal microstructure. (I) Is the initial seeding, (II) after holding at temperature $\varepsilon = -0.25$ for 50×10^3 simulation steps, and (III) after

reducing temperature to $\varepsilon = -0.5$ with another 50×10^3 simulation steps

misorientation of the two grains (a and b) was 30° for the bicrystal configuration.

As seen in the previous case study, the dislocations at the grain boundaries can be easily identifiable in both microstructures in Fig. 7.8.

The microstructure was strained in the x -direction in all deformation simulations. At every 1000 simulation steps, the current grid spacing in the x -direction was incremented as $dx = dx + 5.0 \times 10^{-4}dx_0$, in which $dx_0 = \pi/4$ is the initial grid spacing. Similarly, the grid spacing in the y -direction was reduced to meet the constant volume requirement as given in Eq. 7.15.

The variations of the free energy density with the applied strains resulting from these simulations are summarized in Fig. 7.9. For both cases, the curves exhibit a linear region in which the deformation is elastic and reversible. A procedure to calculate the elastic constants from such deformation simulations is given in [15]. With increasing strains, both curves exhibit deviation from the linearity and

deformation continues in the irreversible plastic regime.

Figure 7.9 clearly exhibits the dependency of the mechanical properties (i.e., elastic properties and yield strength) to the temperature without requiring any fitting parameters or constitutive rules in the simulations, similar to that in MD studies. The system shows very low elastic stiffness and yield strength at high temperature, owing to the fact that the system was very close to its melting temperature (see Fig. 7.2).

The snap shots of deformation at high and low temperatures are given in Figs. 7.10 and 7.11, respectively. A magnified region containing a dislocation in one of these images is shown in Fig. 7.12 which fits very well to the textbook definition of a dislocation (an extra row of atoms that are not in their equilibrium position in the lattice). In addition, due to presence of dislocation, the nearby lattice distortions are clearly identifiable in the figure. The arrows in Figs. 7.10 and 7.11 indicate the positions of

Fig. 7.9 Variation of free energy density with applied strains during the course of simulations

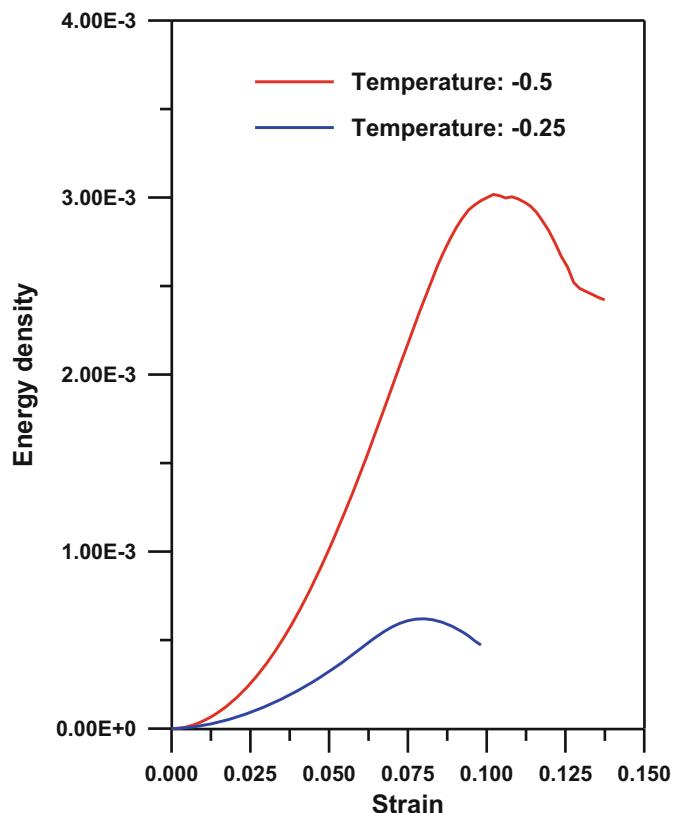
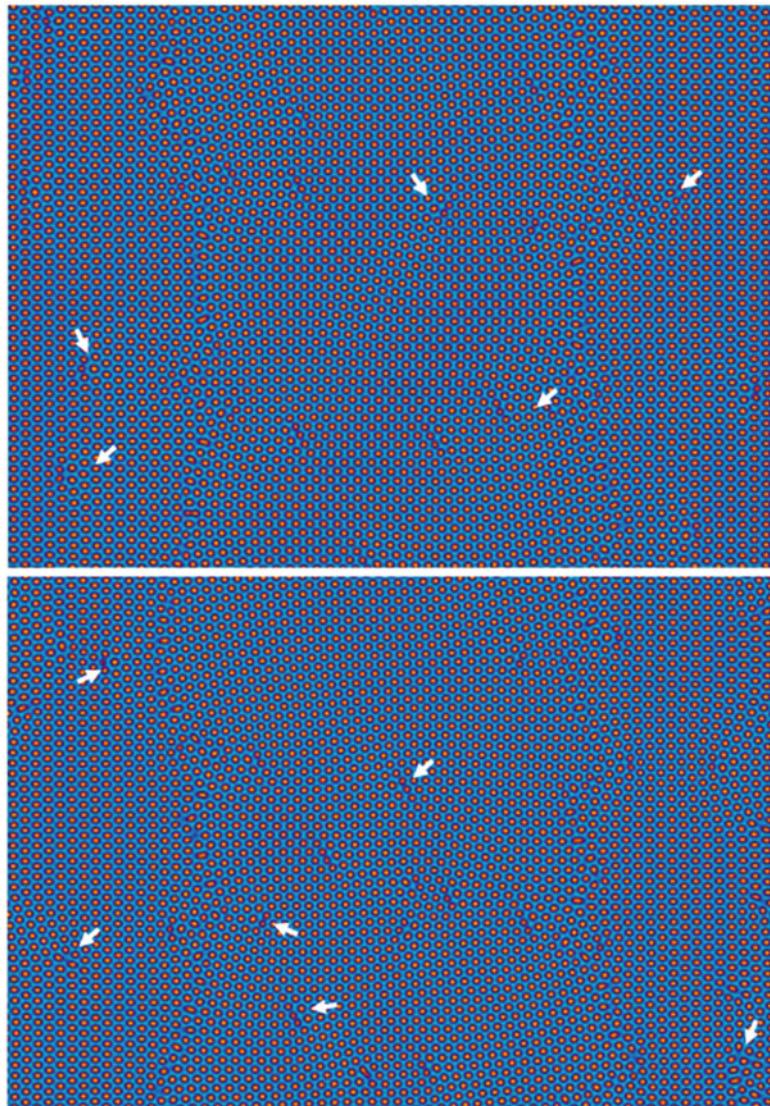


Fig. 7.10 At high temperature deformation simulation, the appearance of the microstructure. The arrows indicate some, but not all, of the present dislocations. Top at strain level 0.0785 and bottom at strain level 0.09187



some, but not all, of the dislocations that are present in the microstructure. The movies that are given in subdirectory *case_study_18* in downloadable file clearly show their emissions from the grain boundaries and their glide on their respective slip planes.

7.4.4 Source Codes

The program *pfc_def_v2.m* is build on the code *pfc_2D_v2.m* given in the [Case Study XVI](#).

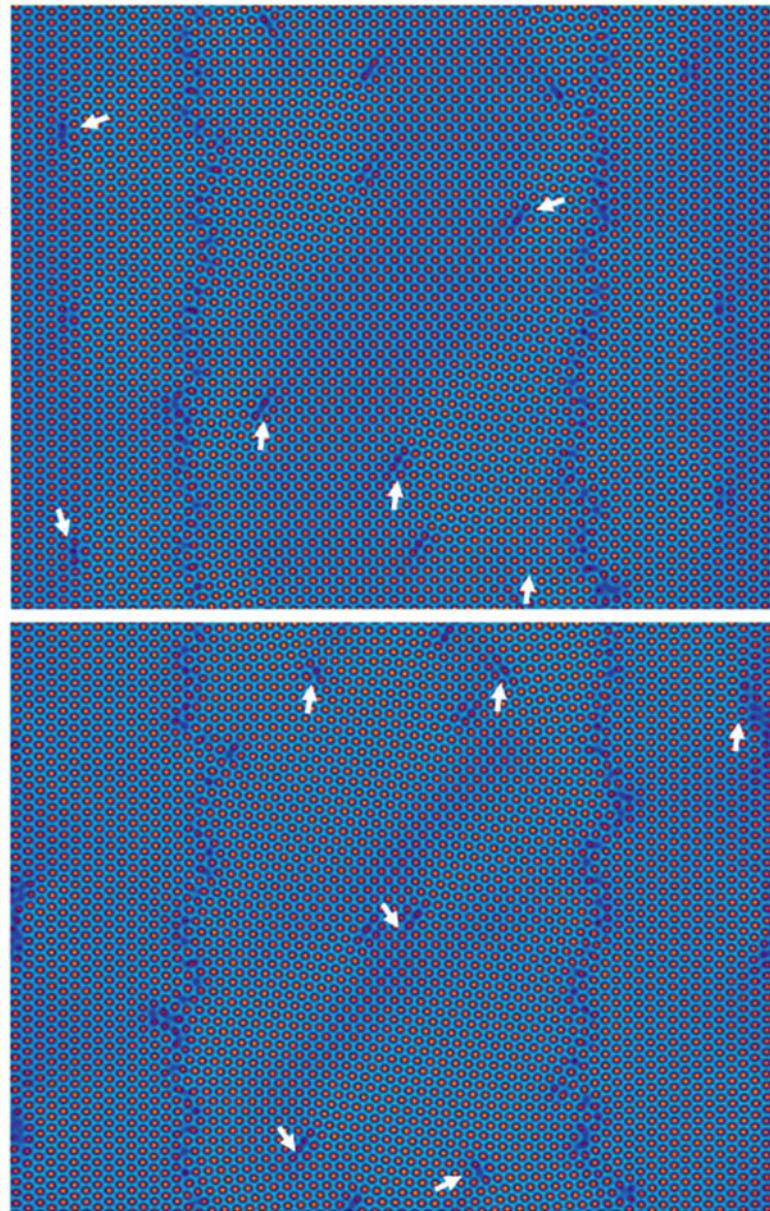
In order to include the deformation, as described earlier, it requires only the inclusion of a few new lines to the original code. In its annotation only these new lines are explained.

Program

pfc_def_v2.m

This program solves PFC model with semi-implicit Fourier spectral method in two

Fig. 7.11 At low temperature deformation simulation, the appearance of the microstructure. The arrows indicate some, but not all, of the present dislocations. Top at strain level 0.0863 and bottom at strain level 0.1375



dimensions and introduces deformation in the chosen Cartesian coordinate direction. It is optimized for Matlab/Octave.

The program makes calls to the following functions:

- **prepare_fft.m**
- **write_vtk_grid_values.m**

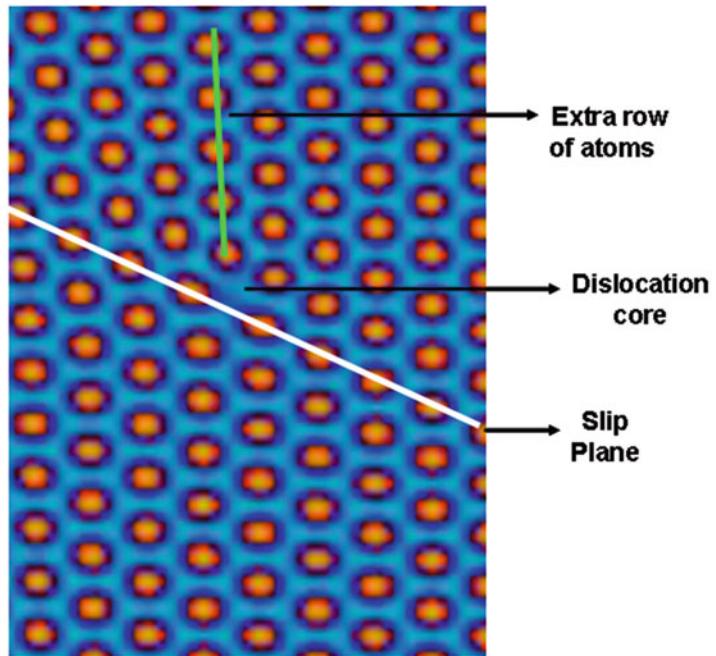
Listing:

```

1      %%%%%%%%
2      %
3      % 2D SEMI-IMPLICIT SPECTRAL %
4      % PHASE-FIELD CRYSTAL CODE %
5      % OPTIMIZED FOR MATLAB/OCTAVE %
6      %
7      %%%%%%%%

```

Fig. 7.12 A close-up view of region containing a dislocation



```

8                               33      dtime =      0.01;
9      %% get intial wall time:   34      nprint =      5000;
10     time0=clock();           35      nstart =        1;
11     format long;            36
12                               37      ndefor = 1000;
13     out1 = fopen('final_conf.out', 38      stran = 0.0;
14         'w');                39
15     out2 = fopen('energy.out', 'w'); 40      %
16     %%-- Simulation cell parameters: 41      %%-- Material parameters:
17                               42      %
18     Nx= 512;                 43
19     Ny= 512;                 44      den0 = -0.285;
20                               45      tempr = -0.5;
21     pix=4.0*atan(1.0);       46      tempr0 = tempr;
22                               47      noise = den0*(10^-2);
23     dx= pix/4.0;             48
24     dy= pix/4.0;             49      %-----
25                               50      % if infile ==1 read input from
26     dx0 = dx;                51      %
27     dy0 = dy;                52      %-----
28     %                           53      infile = 1;
29     %%--Time integration parameters: 54      if (infile == 1)
30     %                           55
31                               56      %%--- open input file
32     nsteps = 350000;          57

```

```

58     in1 = fopen('bi_2r.inp','r');      104
59                                     105     stran = stran + 5.0e-4*dx0;
60 %-----          106
61                                     107     [kx,ky,k2,k4] = prepare_fft
62     for i=1:Nx                      108
63     for j=1:Ny                      109     end %if
64                                     110     %---
65     mm1=fscanf(in1,'%5d %5d',    111
66     [2,1]);                         112     f_den =fft2(den);
67                                     113
68     den(i,j)=mm2(1,1);             114     Linx=-k2.* (tempr+1.0-2.0*
69     end                           115     k2+k4);
70     end                           116     denom = 1.0-dtime.*Linx;
71                                     117
72     else                           118     den3=den.^3;
73                                     119     f_den3=fft2(den3);
74     for i = 1:Nx                   120
75     for j = 1:Ny                   121     Nonx=-k2.*f_den3;
76     den(i,j) = den0 +noise*       122
77     (0.5-rand);                  123     f_den=(f_den + dtime*Nonx) .
78     end                           124     /denom;
79     end                           125     den = real(ifft2(f_den));
80     end                           126
81     end %if                       127     %
82                                     128     %-- Print results:
83                                     129
84     %                               130     if((mod(istep,nprint) == 0) ||
85     %--Prepare fft:               131
86     %                               132     fprintf('done step: %d\n',
87     [kx,ky,k2,k4] = prepare_fft   133
88     (Nx,Ny,dx,dy);              134     istep);
89                                     135
90     %--                           136     %energy calculation
91     % Evolve                      137     ss2 = den.^2;
92     %--                           138     ss4 = ss2.^2;
93                                     139     f_ff = 0.5*f_den.* (1.0-2.0*
94     for istep = nstart : nsteps   140
95     %tempr = tempr + tempr0/nsteps; 141     k2+k4);
96                                     142
97     %--- loading:                 143     ff = real(ifft2(f_ff));
98     if(mod(istep,ndefor) == 0)     144
99                                     145     ff = ff.*den + 0.5*tempr*ss2+
100    dx = dx + 5.0e-4*dx0;        144
101    dy = dy - 5.0e-4*dx0;        145     0.25*ss4;
102                                     146
103                                     147     energ = sum(ff(:)) / (Nx*Ny);

```

```

146
147     if(istep == 1)
148         energ0=energ;
149     end
150
151     energ=energ-energ0;
152
153     fprintf(out2, '%14.6e %14.6e
154     \n', stran,energ);
155
156     write_vtk_grid_values(Nx,Ny,
157     dx,dy,istep,den,ff);
158
159     end %if
160
161     %-----
162     % Write configuration:
163     %-----
164
165     if(istep == nsteps);
166
167         for i=1:Nx
168             for j=1:Ny
169                 fprintf(out1,'%5d %5d %14.6e
170                 \n',i,j,den(i,j));
171             end
172         end
173
174     end %if
175
176     end % end of time step
177
178     compute_time = etime(clock()),
179     time0);
180
181     fprintf('Compute Time: %10d
182     \n',compute_time);

```

Line numbers:

37:	Number of time steps for strain increment.
38:	Total applied strain.
100–109:	Strain increment for loading.
102:	Increase current grid spacing in the x -direction, Eq. 7.15.
103:	Reduce current grid spacing in the y -direction, Eq. 7.15.
105:	Update total applied strain.
107:	Recalculate the FFT coefficients with the new grid parameters.

References

- Wang YU, Jin YM, Cuitino AM, Khachaturyan AG (2001) Nanoscale phase-field microelasticity theory of dislocations: Model and 3D simulations. *Acta Mater* 49:1847
- Shen C, Wang YU (2003) Phase-field model of dislocation networks. *Acta Mater* 51:2595
- Koslowski M, Cuitino AM, Ortiz M (2002) A phase-field theory of dislocation dynamics, strain hardening and hysteresis in ductile single crystals. *J Mech Phys Solid* 50:2597
- Hu SY, Chen LQ (2001) Solute segregation and coherent nucleation and growth near a dislocation: a phase-field model integrating defect and phase-microstructures. *Acta Mater* 49:463
- Wang Y, Li J (2010) Phase-field modeling of defects and deformation: overview no. 150. *Acta Mater* 58:1212
- Geslin PA, Appolarire B, Finel A (2014) A phase-field model for dislocation climb. *App Phys Lett* 104:011903
- Groma I, Vandrus Z, Ispanovity PD (2015) Scale-free phase-field theory of dislocations. *Phys Rev Lett* 114:015503
- Stefanovic P, Haataja M, Provatas N (2006) Phase-field crystals with elastic interactions. *Phys Rev Lett* 96:225504
- Stefanovic P, Haataja M, Provatas N (2009) Phase-field crystal study of deformation and plasticity in nanocrystalline materials. *Phys Rev E* 80:046107
- Chan PY, Tsekenis G, Dantzig J, Dahmen KA, Goldenfield N (2010) Plasticity and dislocation dynamics in phase-field crystal models. *Phys Rev Lett* 105:0.15502
- Berry J, Provatas N, Rottler J, Sinclair C (2012) Defect stability in phase-field crystal models: stacking faults and partial dislocations. *Phys Rev B* 86:224112
- Adland AJ (2013) Phase-field modeling of polycrystalline materials. PhD thesis, Dept of Physics, Northeastern University, Boston Massachusetts.
- Hirouchi T, Takaki T, Tomoita Y (2009) Development of numerical scheme for phase field crystal simulation. *Comp Mater Sci* 44:1192
- Hirouchi T, Takaki T, Tomoita Y (2010) Effects of temperature and grain size on phase-field deformation simulation. *Int J Mech Sci* 52:309
- Arnold NP, Chan VWL, Elder KR, Thornton K (2013) Calculations of isothermal elastic constants in the phase-field crystal model. *Phys Rev B* 87:014103

Any numerical algorithm can be characterized in terms of its properties, which are accuracy, flexibility to handle many different problems, robustness, and computational efficiency in terms of amount of required coding and computational resources memory, storage and processor. Very often, it is difficult to achieve all of these properties in one particular algorithm. The aim of this book was that the reader, after hands on experience, can understand the fine details and strengths and weaknesses of each given solution's methods sufficiently well and can adopt the suitable algorithm needed for their phase-field models. It is hoped that this objective was achieved by facilitating such comparative study with a collection of algorithms and case studies.

The codes presented may also serve as rapid prototyping of new phase-field models either in their present form or with simple modifications. As demonstrated in [Case Study XVI](#), their extension from two-dimension to three-dimension can

be accomplished with little effort. Also, there are still plenty of improvements that can be made to the optimized versions of the codes, so their performance can be increased further and turned into production codes.

For the readers who wishes to implement the codes with the traditional programming languages, Fortran, C, and C++, the longhand version of the codes will be helpful to achieve this objective. In most cases, this may simply require the replacement of the syntax of Matlab/Octave with the syntax of the chosen programming language.

The computer programs and background materials discussed in each section of this book also provide a forum for undergraduate level modeling-simulation courses as part of their curriculum. Even though there are no specific exercises provided, if desired, exercises can be easily devised in variety of ways (for example, by modifying either material-specific properties or simulation-specific properties).

Errata to: Programming Phase-Field Modeling

S. Bulent Biner

Errata to:

**S.B. Biner, *Programming Phase-Field Modeling*,
DOI 10.1007/978-3-319-41196-5**

1) In chapter 4, on page 17, the equation (4.1c) has been updated to read as:

$$(\Delta u)_i^{\pm} = \frac{u_{i+1} - u_{i-1}}{2h}$$

2) In chapter 4, on page 57, in Fig. 4.15 caption, the time steps have been updated to read as:

- (a) At time step 250, (b) at time step 650, (c) at time step 1250, and
- (d) at time step 2500.

3) In chapter 4, on page 58, line number 158 has been updated to read as:

```
mobil = dvol*phi + dvap*(1.0-phi) + dsur*con(i,j)*(1.0-con(i,j)) +
dgrb*sum;
```

4) In chapter 4, on page 62, line number 83 has been updated to read as:

```
mobil = dvol*phi + dvap*(1.0-phi) + dsur*con.*(1.0-con) + dgrb*sum;
```

5) In chapter 4, on page 75, line numbers 104 and 105 have been updated to read as:

```
104    phidx(i,j) = (phi(ip,j)-phi(im,j)) / (2.0*dx);
105    phidy(i,j) = (phi(i,jp)-phi(i,jm)) / (2.0*dy);
```

The updated online versions of these chapters can be found at

<http://dx.doi.org/10.1007/978-3-319-41196-5>

http://dx.doi.org/10.1007/978-3-319-41196-5_3

http://dx.doi.org/10.1007/978-3-319-41196-5_4

http://dx.doi.org/10.1007/978-3-319-41196-5_6

- 6) In chapter 4, on page 75, line numbers 157 and 162 have been updated to read as:

```
157  epsilon(i,jm)*epsilon_deriv(i,jm)*phidx(i,jm)) / (2.0*dx);
162  epsilon(im,j)*epsilon_deriv(im,j)*phidy(im,j)) / (2.0*dy);
```

- 7) In chapter 4, on page 80, line numbers 20 and 21 have been updated to read as:

```
20  matdx = matdx / dx;
21  matdy = matdy / dy;
```

- 8) In chapter 4, on page 87, line numbers 134 and 135 have been updated to read as:

```
134  phidx(i,j) = (phi(ip,j) - phi(im,j)) / (2.0*dx);
135  phidy(i,j) = (phi(i,jp) - phi(i,jm)) / (2.0*dy);
```

- 9) In chapter 6, on page 239, in equation (6.87) the term $\frac{\partial N_j}{\partial j}$ has been replaced with $\frac{\partial N_j}{\partial y}$

- 10) In chapter 6, on page 268, line numbers 150 and 152 have been updated to read as:

```
150  dsendc(ielem,kgasp) = dsendc(ielem,kgasp) + 0.5*(stran(istre) -
           cvgp *
           eigen(istre))*stresb(istre) - eigen(istre)
           *stres(istre);
152  dsen2dc(ielem,kgasp) = dsen2dc(ielem,kgasp) + eigen(istre)*
           (stres0(istre)-2.0*stresb(istre));
```

- 11) In chapter 6, on page 270, line number 144 has been updated to read as:

```
144  dsendc( :,kgasp) = dsendc( :,kgasp) + 0.5*(stran( :,istre) -
           cvgp*eigen(istre)).* stresb( :,istre) - eigen(istre)
           *stress( :,istre);
```

- 12) In chapter 6, on page 317, line number 112 has been updated to read as:

```
112  phirgp = phirgp + ephir(inode) * shape(inode);
```

- 13) In chapter 6, on page 321, line number 79 has been updated to read as:

```
79  phirgp = phirgp + ephir( :, inode) * shape(inode);
```

- 14) In chapter 6, on page 327, line number 97 has been updated to read as:

```
97  phirgp = phirgp + ephir(inode)*shape(inode);
```

- 15) In chapter 6, on page 330, line number 73 has been updated to read as:

```
73  phirgp = phirgp + ephir( :, inode)*shape(inode);
```

- 16) In chapter 6, on page 333, line number 37 has been updated to read as:

```
37  eldis(ievab) = tdisp(itotv);
```

- 17) In chapter 3, on page 15, the last three paragraphs have been replaced with the following text:

Password protected, downloadable zip files, containing the source codes and the movie files resulting from the simulations, are available. Readers who have purchased the eBook or print versions of this book can visit the Springer Link webpage for the book at the following link:

<https://link.springer.com/book/10.1007%2F978-3-319-41196-5>

On this page, there will be downloadable zip files for Chapters 4, 5, 6, and 7, that contain the necessary files to execute the main programs, per the case studies in those individual chapters, with movie files in the AVI format resulting from these simulations. There will also be additional materials related to the Appendices A, B, and C within the downloadable zip files for Chapters 4, 5, and 6.

The names of the downloadable zip files for the individual chapters and the required passwords to unzip them are given in the table below:

Chapters	Filenames	Passwords
Chapter-4	prog_pf_ch4.zip	sbb#4RFV
Chapter-5	prog_pf_ch5.zip	sbb#5TGB
Chapter-6	prog_pf_ch6.zip	sbb#6YHN
Chapter-7	prog_pf_ch7.zip	sbb#7UJM

- 18) In Appendix A, on page 371, the text “The code and the resulting output files are given in subdirectory of Appendix A in the subdirectory Appendices of the downloadable file.” has been replaced with “**The code and the resulting output files for Appendix A are included in the downloadable zip files for Chapters 4, 5, and 6.**”
- 19) In Appendix B, on page 377, the text “These functions can be found in the subdirectory Appendix B of the subdirectory Appendices of the downloadable file.” has been replaced with “**These functions can be found in the downloadable zip file for Chapter 5.**”
- 20) In Appendix C, on page 381, the text “The program and the associated functions together with the input and output files can be found in the subdirectory Appendix C in the subdirectory of Appendices of downloadable file.” has been replaced with “**The program and the associated functions, together with the input and output files, can be found in the downloadable zip file for Chapter 6.**”

Appendix A

The initial polycrystalline microstructure used in grain growth simulations in the *Case Studies II*, *VI*, and *XII* was generated with the program given below. The number of grains in those simulations was 25; however, there is no limit for the number of grains that program can generate. The program is based on the Voronoi tessellation of the randomly distributed points that is equal to the desired number of grains in the phase-field simulation cell with periodicity. For the Voronoi tessellation, the program utilizes the internal function *voronoin* of Matlab/Octave.

Since the points are randomly distributed, each execution produces different grain microstructures with the same number of points. Program produces two graphical output files. The file “*Voroni_vertices.out*” shows the whole Voronoi tessellation results and it can be viewed in gnuplot with the command, *plot “Voroni_vertices.out” w l*, as shown in Fig. A.1. The second graphical output file shows the data in output file *grain_25.inp* which is directly used in the phase-field codes. Again, this graphical output file can be viewed in gnuplot with the command, *load “final_plot.p”*, as shown in Fig. A.2.

The code and the resulting output files for Appendix A are included in the downloadable zip files for Chapters 4, 5, and 6.

Program

voronoi_1.m

This program generates random polycrystalline microstructure by using Voronoi tessellation for the phase-field models.

Listing:

```
1 %%%%%%%%%%%%%%%%
2 % Generation of polycrystal %
3 % grain microstructure %
4 % with Voronoi tessellation %
5 %%%%%%%%%%%%%%%%
6
7 out=fopen('Voroni_vertices.out',
8 'w');
9 out1=fopen('plot_1.out','w');
10 out2=fopen('final_plot.p','w');
11 out3=fopen('original_points.out',
12 'w');
13
14 %---
15
16 npoin=25;
17 xmax=32.0;
18 ymax=32.0;
19 x0 =0.0;
```

The original version of this chapter was revised. An erratum to this chapter can be found at DOI [10.1007/978-3-319-41196-5_9](https://doi.org/10.1007/978-3-319-41196-5_9)

Fig. A.1 A typical Voronoi tessellation results for the points with their all nine replicas. The file name that contains this graphical result is “Voronoi_vertices.out” and the figure is generated in gnuplot with the command `plot "Voronoi_vertices.out" w l`

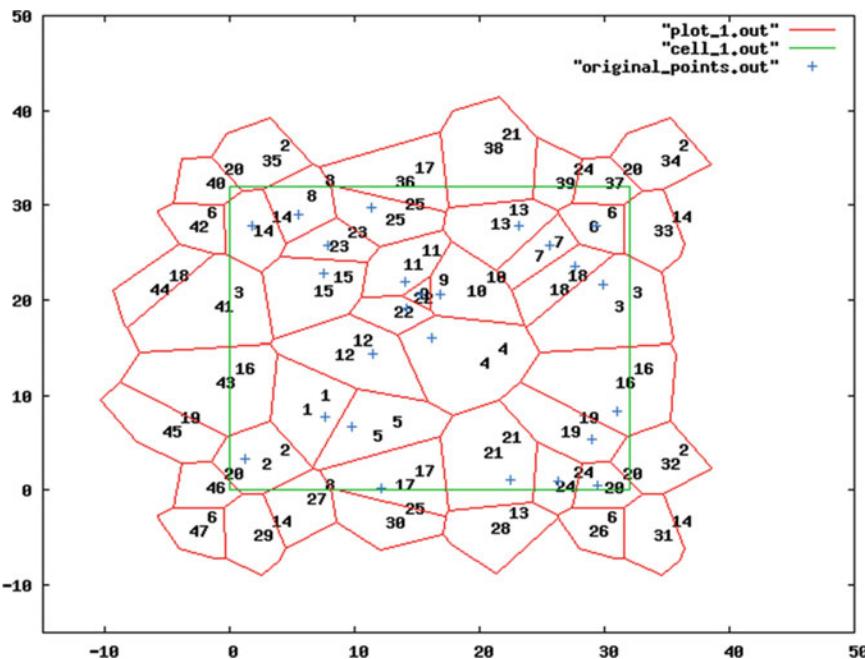
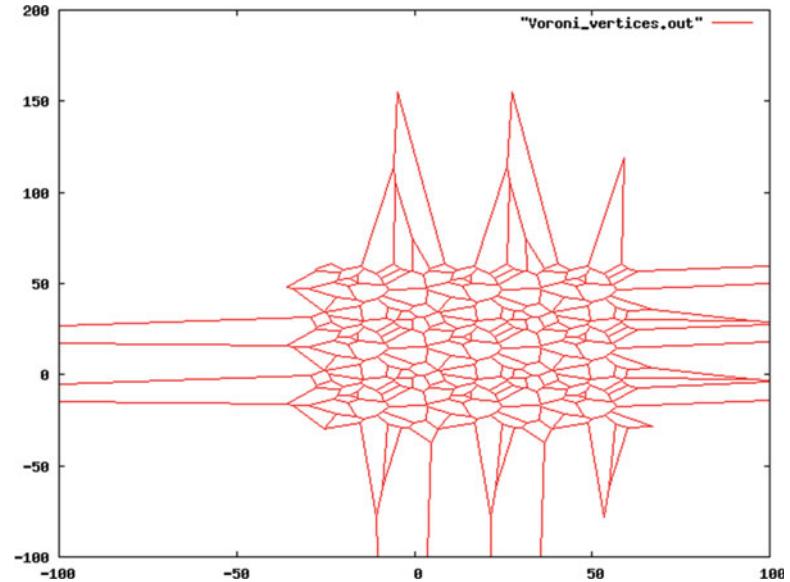


Fig. A.2 Graphical output of the data that will be used in the phase-field simulations. In the figure, green box is the simulation cell in the phase-field model. The upper numbers correspond to the grain numbers and the lower ones are the cell numbers of the Voronoi cells. Blue points correspond to the coordinates of the randomly generated points. The file name that contains this graphical result is `final_plot.p` and the figure is generated in gnuplot by using the command `load "final_plot.p"`

```

20 y0 =0.0;
21 extra = 2.0;
22
23 %-----
24 % generate random points
25 % for voronoi tessellation
26 %-----
27
28 rand('state',7)
29
30 x=xmax*rand(npoin,1);
31 y=ymax*rand(npoin,1);
32
33 %-----
34 % Duplicate the points for symmetry:
35 %-----
36
37 for ipoin=1:npoin
38 jpoind=npoin+ipoin;
39 x(jpoind)=x(ipoin);
40 y(jpoind)=y(ipoin)-ymax;
41 end
42 %---
43 for ipoin=1:npoin
44 jpoind=npoin*2+ipoin;
45 x(jpoind)=x(ipoin)+xmax;
46 y(jpoind)=y(ipoin)-ymax;
47 end
48 %---
49 for ipoin=1:npoin
50 jpoind=npoin*3+ipoin;
51 x(jpoind)=x(ipoin)+xmax;
52 y(jpoind)=y(ipoin);
53 end
54 %---
55 for ipoin=1:npoin
56 jpoind=npoin*4+ipoin;
57 x(jpoind)=x(ipoin)+xmax;
58 y(jpoind)=y(ipoin)+ymax;
59 end
60 %---
61 for ipoin=1:npoin
62 jpoind=npoin*5+ipoin;
63 x(jpoind)=x(ipoin);
64 y(jpoind)=y(ipoin)+ymax;
65 end
66 %---
67 for ipoin=1:npoin
68 jpoind=npoin*6+ipoin;
69 x(jpoind)=x(ipoin)-xmax;
70 y(jpoind)=y(ipoin)+ymax;
71 end
72 %---
73 for ipoin=1:npoin
74 jpoind=npoin*7+ipoin;
75 x(jpoind)=x(ipoin)-xmax;
76 y(jpoind)=y(ipoin);
77 end
78 %---
79 for ipoin=1:npoin
80 jpoind=npoin*8+ipoin;
81 x(jpoind)=x(ipoin)-xmax;
82 y(jpoind)=y(ipoin)-ymax;
83 end
84
85 %%
86 %--- Print-out origional random
87 % points:
88
89 for i=1:npoin
90 fprintf(out3,'%4.6e %14.6e\n',
91 x(i),y(i));
92 end
93 %%
94 %--- Print the simulation cell
95 % coordinates:
96
97 fprintf(out4,'%14.6e %14.6e\n',
98 x0,y0);
99 fprintf(out4,'%14.6e %14.6e\n',
100 xmax,y0);
101 fprintf(out4,'%14.6e %14.6e\n',
102 x0,ymax);
103 =====
104 %--- generate voronoi diagram
105 =====
106
107 [c,f] = voronoin([x,y]);
108
109

```

```

110 %-----
111 %- rearrange voronoin output
112 %- for their connectivity
113 %-----
114
115 nvelem = size(f);
116
117 ncount = 0;
118 for i=1:nvelem
119 flag=1;
120
121 vnodes = f{i,:};
122 nnodes = size(vnodes,2);
123
124 for j=1:nnodes
125 if(vnodes(j) == 1)
126 flag=0;
127 end
128 end
129
130 if(flag == 1)
131 ncount = ncount+1;
132 for j=1:nnodes
133 lnods(ncount,j)=vnodes(j);
134 end
135 end
136
137 end
138
139 %-----
140 % print voronoi results to file
141 % to be viewed later in Voroni_
142 vertices.out
143
144 for i=1:ncount
145 fprintf(out,'# i %d\n',i);
146
147 nnodes = size(lnods,2);
148
149 for j=1:nnodes
150 kk = lnods(i,j);
151 if(kk ~= 0)
152 fprintf(out,'%14.6e %14.6e\n',
153 c(kk,1),c(kk,2));
154 end
155 kk = lnods(i,1);
156 fprintf(out,'%14.6e %14.6e\n',
157 c(kk,1),c(kk,2));
158 fprintf(out,'\n');
159
160 end
161
162 %=====
163 % Clip far outside voronoi elements
164 % from the simulation cell
165 %=====
166
167 nelem=0;
168 for i=1:ncount
169 flag=0;
170 for j=1:nnodes
171 kk = lnods(i,j);
172 if(kk ~= 0)
173 if(c(kk,1) >= -extra && c(kk,1)
174 <= xmax+ extra)
175 if(c(kk,2) >= -extra && c(kk,2)
176 <= ymax+ extra)
177 flag=1;
178 end
179 end %j
180
181 if(flag == 1)
182 nelem=nelem+1;
183 jnode=0;
184 for j=1:nnodes
185 kk=lnods(i,j);
186 if(kk ~= 0 )
187 jnode=jnode+1;
188 lnods2(nelem,jnode) =lnods(i,j);
189 end
190 end %j
191
192 nnodes2(nelem) =jnode;
193 end % iflag
194 end %icount
195
196
197 %=====
198 % Assign grain numbers to
199 % voronoi elements
200 %=====
201 twopi =8.0*atan(1.0);
202 epsilon=1.0e-4;
203
204 for isector=1:9

```

```
205 for ipoin=1:npoint
206 jpoind=(isector-1)*npoint+ipoin;
207
208 for ielem=1:nelem
209 theta=0.0;
210 nnode=nnode2(ielem);
211
212 for inode=1:nnode
213 kk=lnods2(ielem,inode);
214
215 xv1=c(kk,1);
216 yv1=c(kk,2);
217
218 jnode=inode+1;
219 if( inode == nnode)
220 jnode=1;
221 end
222
223 jj=lnods2(ielem,jnode);
224 xv2=c(jj,1);
225 yv2=c(jj,2);
226
227 p2x=(xv1-x(jpoind));
228 p2y=(yv1-y(jpoind));
229
230 p1x=(xv2-x(jpoind));
231 p1y=(yv2-y(jpoind));
232
233 x1=sqrt(p1x*p1x+p1y*p1y);
234 x2=sqrt(p2x*p2x+p2y*p2y);
235
236 if(x1*x2 <= epsilon)
237 theta=twopi;
238 else
239 tx1=((p1x*p2x+p1y*p2y)/(x1*x2));
240
241 if(abs(tx1) >= 1.0)
242 tx1=0.9999999999;
243 end
244
245 theta=theta+acos(tx1);
246
247
248 end
249
250 end %inode
251
252 if(abs(theta-twopi) <= epsilon)
253 igrain(ielem)=ipoin;
254
255
256 end
257
258
259 end % ielem
260
261 end % ipoin
262 end
263
264 %%-----
265 %% print out as input file
266 %%-----
267
268 [nn1,nn2]=size(c);
269 nnode=size(lnods2,2);
270 fprintf(out5,'%5d %5d %5d\n',nn1-1,nnode,nelem);
271
272 for i=2:nn1
273 fprintf(out5,' %5d %14.6e
%14.6e\n', i,c(i,1),c(i,2));
274 end
275
276 for i=1:nelem
277 fprintf(out5,'%5d',i);
278 for j=1:nnode
279 fprintf(out5,'%5d',lnods2(i,j));
280 end
281 fprintf(out5,'%5d',igrain(i));
282 fprintf(out5,'\n');
283 end
284
285 =====
286 % graphic output
287 =====
288
289
290 for i=1:nelem
291 fprintf(out1,'# i %d %d\n',i,
nnode2(i));
292
293 nnode=size(lnods2,2);
294
295 ncount=0;
296 xcod=0.0;
297 ycod=0.0;
298
299 %nnode = nnode2(ielem);
300
301 for j=1:nnode
302
303 kk = lnods2(i,j);
```

```

304 fprintf(out1,'# %5d %5d\n',j, kk);
305
306 if(kk ~= 0)
307 fprintf(out1,'%14.6e %14.6e\n',
c(kk,1),c(kk,2));
308 ncount=ncount+1;
309 xcod =xcod +c(kk,1);
310 ycod =ycod +c(kk,2);
311 end
312 end
313 kk =lnods2(i,1);
314 fprintf(out1,'%14.6e %14.6e\n',
c(kk,1),c(kk,2));
315 fprintf(out1,'\n');
316
317 xcod =xcod /ncount;
318 ycod =ycod /ncount;
319
320 fprintf(out2,'set label ');fprintf
(out2,'"'); fprintf(out2,'%d',i);
...
321 fprintf(out2,'" at'); fprintf
(out2,'%14.6e , %14.6e\n', xcod,
ycod);
322 fprintf(out2,'\n');
323
324 fprintf(out2,'set label ');
fprintf(out2,'"'); fprintf
(out2,'%d',igrain(i)); ...
325 fprintf(out2,'" at'); fprintf
(out2,'%14.6e , %14.6e\n',
xcod+1.5, ycod+1.5);
326 fprintf(out2,'\n');
327
328 end
329 fprintf(out2, 'plot "plot_1.out"
wl, "cell_1.out" wl,
"original_points.out"\n');
330
331 =====
332

```

Line numbers:

7–13:	Assign unit names for the output files.
7:	Output file name for Voronoi tessellation.
8:	Intermediate plot file name.
9:	Final graphical output file name.

10:	File name that contains the coordinates of the randomly generated points.
11:	File name that contains the Cartesian coordinates of the corners of the simulation cell.
12:	Tabulated output file that will be used in phase-field simulations.
16:	Number of grains in the simulation cell.
17:	The length of the simulation cell in the x-direction.
18:	The length of the simulation cell in the y-direction.
19–20:	The origin of the coordinate system.
21:	A parameter will be used for trimming the Voronoi tessellation results.
24–32:	Randomly generate the x- and y-coordinates of the points.
34–84:	Replicate the points for their nine periodic images, for generation of periodic simulation cell in phase-field simulations.
86–92:	Output the coordinates of the points to file.
94–101:	Output the corner coordinates of the simulation cell to file.
104–108:	Generate Voronoi tessellation by using Matlab/Octave function <i>voronoin</i> .
111–137:	Rearrange the <i>voronoin</i> output for the connectivity list of the Voronoi cells.
140–160:	Print out the Voronoi cell data into graphical output file <i>Voronoi-vertices.out</i> file. This file can be viewed in gnuplot with command, <i>plot "Voronoi_vertices.out" wl</i> .
163–195:	Clip the Voronoi cells that are far away from the simulation cell for efficiency. This is achieved with the variable name extra in lines 173 and 174. Too narrow choice for the value of extra may result in non-periodic grain structure in final output. On the other hand, to large value brings an inefficiency. Since the results of the Voronoi tessellation results is not known beforehand, it is set to an optimum value by trial and error as shown in Fig. A.2.
198–263:	Assign grain numbers to the remaining Voronoi cells with the point in the box algorithm.
265–283:	Print out the coordinates and the connectivity list of the Voronoi cells that have the assigned grain numbers to file. This file is directly used by the phase-field codes given in Case Studies II, VI, and XII.
286–330:	Write the graphical output files, “plot_1.out” and “final_plot.p” files.

(continued)

Appendix B

The functions given in this section generate the Green's tensor in three-dimension. In order to avoid the exhaustion of the available memory, in three-dimension, it is generated with two separate functions. In the main program, the function *green_tensor1_3D.m* should be called only once after establishing FFT coefficients. Then, the function *green_tensor2_3D.m* should be called at every grid points, whenever, the Green's tensor is needed.

These functions can be found in the downloadable zip file for Chapter 5.

Function

green_tensor1_3D.m

This function evaluates the coefficients that will be used in the *green_tensor2_3D.m*. It needs to be called only once after establishing the FFT coefficients in the main program.

Variable and array list:

Nx:	Number of grid points in the x-direction.
Ny:	Number of grid points in the y-direction.
Nz:	Number of grid points in the z-direction.
kx(Nx):	Fourier coefficients in the x-direction.
ky(Ny):	Fourier coefficients in the y-direction.
kz(Nz):	Fourier coefficients in the z-direction.
cm11,cm12, cm44:	Elasticity parameters of the matrix phase.

(continued)

cp11,cp12,cp44:	Elasticity parameters of the second phase.
omeg11(Nx,Ny, Nz):	Coefficient needed for the Green's tensor.
omeg22(Nx,Ny, Nz):	Coefficient needed for the Green's tensor.
omeg33(Nx,Ny, Nz):	Coefficient needed for the Green's tensor.
omeg12(Nx,Ny, Nz):	Coefficient needed for the Green's tensor.
omeg23(Nx,Ny, Nz):	Coefficient needed for the Green's tensor.
omeg13(Nx,Ny, Nz):	Coefficient needed for the Green's tensor.

Listing:

```
1 function [omeg11,omeg22,omeg33,  
omeg12,omeg23,omeg13] =  
green_tensor1_3D(Nx,Ny,Nz, ...  
2 kx,ky,kz, ...  
3 cm11,cm12,cm44, ...  
4 cp11,cp12,cp44)  
5 format long;  
6  
7 c11 = 0.5*(cm11+cp11);  
8 c12 = 0.5*(cm12+cp12);  
9 c44 = 0.5*(cm44+cm44);  
10  
11  
12  
13 chi=(c11-c12-2.0*c44)/c44;  
14  
15 for i=1:Nx  
16 for j=1:Ny  
17 for k=1:Nz
```

```

18 rr=kx(i)^2+ky(j)^2+kz(k)^2;
19 d0=c11*rr^3+chi*(c11+c12)*rr*
20 (kx(i)^2*ky(j)^2 ...
21 +ky(j)^2*kz(k)^2+kx(i)^2*
22 kz(k)^2)+chi^2*...
23 (c11+2*c12+c44)*kx(i)^2*ky(j)^2
24 *kz(k)^2;
25 if(rr < 1.0e-8)
26 d0=1.0
27 end
28
29 omeg11(i,j,k)=(c44*rr^2+(c11+c12)
*chi*ky(j)^2*kz(k)^2 ...
30 +(c11-c44)*rr*(ky(j)^2
+kz(k)^2))/(c44*d0);
31
32 omeg22(i,j,k)=(c44*rr^2+(c11+c12)
*chi*kz(k)^2*kx(i)^2 ...
33 +(c11-c44)*rr*(kz(k)^2
+kx(i)^2))/(c44*d0);
34
35 omeg33(i,j,k)=(c44*rr^2+(c11+c12)
*chi*kx(i)^2*ky(j)^2 ...
36 +(c11-c44)*rr*(kx(i)^2
+ky(j)^2))/(c44*d0);
37
38 omeg12(i,j,k)=-(c12+c44)*kx(i)*
ky(j)*(rr+chi*kz(k)^2)/(c44*d0);
39
40 omeg23(i,j,k)=-(c12+c44)*ky(j)*
kz(k)*(rr+chi*kx(i)^2)/(c44*d0);
41
42 omeg13(i,j,k)=-(c12+c44)*kx(i)*
kz(k)*(rr+chi*ky(j)^2)/(c44*d0);
43
44 end
45 end
46 end
47
48 end %endfunction

```

Function

green_tensor2_3D.m

This function evaluates the Green's tensor. It needs to be called at every grid points, whenever

the value of Green's tensor is needed in the main program.

Variable and array list:

Nx:	Number of grid points in the x-direction.
Ny:	Number of grid points in the y-direction.
Nz:	Number of grid points in the z-direction.
i:	Current grid number in the x-direction.
j:	Current grid number in the y-direction.
k:	Current grid number in the z-direction.
kx(Nx):	Fourier coefficients in the x-direction.
ky(Ny):	Fourier coefficients in the y-direction.
kz(Nz):	Fourier coefficients in the z-direction.
omeg11 (Nx,Ny,Nz):	Coefficient needed for the Green's tensor.
omeg22 (Nx,Ny,Nz):	Coefficient needed for the Green's tensor.
omeg33 (Nx,Ny,Nz):	Coefficient needed for the Green's tensor.
omeg12 (Nx,Ny,Nz):	Coefficient needed for the Green's tensor.
omeg23 (Nx,Ny,Nz):	Coefficient needed for the Green's tensor.
omeg13 (Nx,Ny,Nz):	Coefficient needed for the Green's tensor.
tmatx(3,3,3):	Green's tensor.

Listing:

```

1 function [tmatx] = green_tensor2_3D
    (Nx,Ny,Nz,kx,ky,kz, ...
2     omeg11,omeg22,omeg33, ...
3     omeg12,omeg23,omeg13, ...
4     i,j,k)
5 format long;
6
7 gmatx(1,1)=omeg11(i,j,k);
8 gmatx(1,2)=omeg12(i,j,k);
9 gmatx(1,3)=omeg13(i,j,k);
10
11 gmatx(2,1)=omeg12(i,j,k);
12 gmatx(2,2)=omeg22(i,j,k);
13 gmatx(2,3)=omeg23(i,j,k);
14
15 gmatx(3,1)=omeg13(i,j,k);

```

```

16 gmatx(3,2)=omeg23(i,j,k);
17 gmatx(3,3)=omeg33(i,j,k)
18 %-- position vector
19
20 dvect(1)=xk(i);
21 dvect(2)=yk(j);
22 dvect(3)=zk(k);
23
24 %-- Green operator:
25
26 for kk=1:3
27 for ll=1:3
28 for ii=1:3
29 for jj=1:3
30
31
32 tmatx(kk,ll,ii,jj)=0.25*(gmatx
33 (ll,ii)*dvect(jj)*dvect(kk) + ...
34 gmatx(kk,ii)*dvect(jj)*
35 dvect(ll) + ...
36 gmatx(ll,jj)*dvect(ii)*
37 dvect(kk) + ...
38 gmatx(kk,jj)*dvect(ii)*
39 dvect(ll));
40
41
42 end %endfunction

```

Appendix C

Mesh generation programs are commonly used in preparation of the input files for the FEM analysis. There are many freely available mesh-generating programs such as Gmsh and also commercial ones requiring licensing such as Cubit. In this section, a simple, yet very efficient, self-contained FEM mesh-generating program is given. The program is the direct Matlab/Octave implementation of the code given in [1].

The code produces meshes with triangular, four- and eight-node isoparametric elements. The generated input file for the FEM analysis still needs to be supplemented with the control parameters, material parameters, and the boundary conditions following the format in *input_fem_pf.m* and *input_fem_elast.m* functions.

An example input file is annotated, after program and function listings, so other input files to the code can be generated. The code also produces a graphical output file which can be viewed in gnuplot with the command, *plot "filename" w l*, as shown in Fig. C.1.

The program and the associated functions, together with the input and output files, can be found in the downloadable zip file for Chapter 6.

C.1 Source Codes

Program

fem_mesh_gen.m

This program generates mesh with triangular, four- and eight-node isoparametric elements on given geometry.

The program makes calls to the following functions:

- **geom_input.m**
- **gen_mesh4.m**
- **split.m**
- **output.m**

Listing:

```
1 %=====
2 %                                     %
3 %      Fem mesh generator          %
4 %                                     %
5 %=====%
6
7 global in;
8 global out1;
9 global out2;
10
```

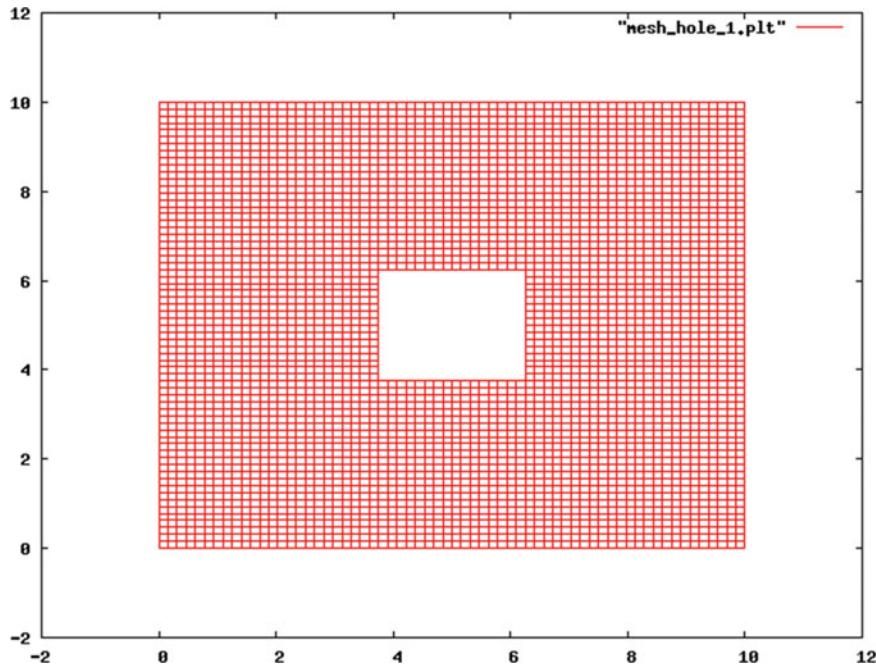


Fig. C.1 FEM mesh generated from the input file, *mesh_hole_1.inp*, with the program *mesh_gem_fem.m*. The figure is plotted in gnuplot with the command, *plot "mesh_hole_1.plt" wl*

```

11
12 global melem;
13 global mpoin;
14
15 melem=5000; %maximum number of
elements to be generated
16 mpoin=5000; %maximum number
of nodes to be generated
17 %-----
18 % open input & output files
19 %-----
20 %-----
21
22 in=fopen('mesh_hole_1.inp','r');
23 out1=fopen('mesh_hole_fem.
inp','w');
24 out2=fopen('mesh_hole_1.plt',
'w');
25 %-----
26 % geometry input
27 %-----
28 %-----
29
30 [nbloc, gpoin, nnodes, ndime, gnods,
gmatn, ...
31      gcord] = geom_input();
32
33 %-----
34 % generate mesh
35 %-----
36 [nelem,npoin,lnods,coord,matno]
= gen_mesh4(nbloc, gpoin, nnodes,
ndime, ...
37      gnods, gmatn, gcord, gmatn);
38
39 %-----
40 % generate triangler elements
41 %-----
42
43
44 if(nnodes == 3)
45 [nelem,lnods,npoin,coord,matno]=
split(nelem,ndime,lnods, ...
46      npoin,coord,nnodes,matno);
47 end
48
49 %-----
50 % output results
51 %-----
```

```

52 out_put(npoin,nelem,nnode,ndime,
53 lnode,coord,matno);
54 disp('Done')
55 disp('Number of elements
56 generated:');disp(nelem);
56 disp('Number of nodes generated: ');
57 disp(npoin);
57

```

Function

geom_input.m

This function reads the topologic information regarding geometry, material information, and the types of finite elements to be generated.

Listing:

```

1 function[nbloc, gpoin, nnode, ndime,
2 gnods, gmatn, ...
3
4
5 global in;
6
7 %-----
8 % read the control cards
9 %-----
10
11 gpoin = fscanf(in,'%d',1);
12 nbloc = fscanf(in,'%d',1);
13 nnode = fscanf(in,'%d',1);
14 ndime = fscanf(in,'%d',1);
15
16 lnode = 8;    %number of node per block
17 %-----
18 % read block nodes
19 %-----
20
21 for ibloc=1:nbloc
22
23 ix = fscanf(in,'%d',[1,1]);
24 idumy = fscanf(in,'%d %d %d %d %d %
25 %d',[8,1]);
25
26 for inode = 1:lnode
27

```

```

28 gnods(ix,inode)=idumy(inode);
29 end
30
31 ix=fscanf(in,'%d',[1,1]);
32 gmatn(ibloc) = ix;
33 end
34
35 %-----
36 % read coordinates
37 %-----
38
39 gcord = zeros(gpoin,ndime);
40
41 for ignod = 1:gpoin
42
43 ix=fscanf(in,'%d',[1,1]);
44 dummy = fscanf(in, '%lf %lf', [2,1]);
45
46 for idime =1:ndime
47 gcord(ix,idime) = dummy(idime);
48 end
49 end
50
51
52 end %endfunction;

```

Function

gen_mesh4.m

This function generates the FEM mesh by subdividing the each block defining the geometry. The common nodes at the subdivision boundaries are eliminated; therefore, there are no nodes with different node numbers and sharing the identical Cartesian coordinates.

The function makes calls to the following functions:

- **srfq.m**
- **goto_100.m**

Listing:

```

1 function [nelem,npoin,lnods,coord,
2 matno] = gen_mesh4(nbloc, gpoin,
3 nnode, ndime, ...
2 tnods, gmatn, tcord, tmato)

```

```

3
4 global in;
5 global melem; % maximum number of
elements to be generated
6 global mpoin; % maximum number of
nodes to be generated
7
8 lnode = 8;
9
10 npont = gpoint;
11
12 mnode = 4;
13 if(lnode == 8)
14 mnode=8;
15 end
16 knode = mnode/4;
17 fnode = knode;
18 %
19 coord = zeros(mpoin,ndime);
20 lnods = zeros(melem,mnode);
21 npoin = 0;
22 nelem = 0;
23
24 iflag=0;
25 for ibloc = 1:nBloc
26
27 %-----
28 % read block subdivision data
29 %-----
30 kbloc = fscanf(in,'%d',[1,1]);
31 ndivx = fscanf(in,'%d',[1,1]);
32 ndivy = fscanf(in,'%d',[1,1]);
33
34 weitx = zeros(ndivx+10,1);
35 weity = zeros(ndivy+10,1);
36
37 wx = fscanf(in,'%lf',[1,1]);
38
39 for idivx=1:ndivx
40
41 %weitx(idivx)=fscanf(in,'%lf',
[1,1]);
42
43 weitx(idivx)=wx;
44 end
45
46 wy = fscanf(in,'%lf',[1,1]);
47
48 for idivy=1:ndivy
49
50 %weity(idivy)=fscanf(in,'%lf',
[1,1]);
51
52 weity(idivy)=wy;
53 end
54
55 %-----
56 % divide the block into the elements
57 %-----
58 total = 0.0;
59
60 for idivx=1:ndivx
61
62 if(weitx(idivx) == 0.0)
63 weitx(idivx) =1.0;
64 end
65 total = total+weitx(idivx);
66 end
67 xnorm=2.0/total;
68
69 total = 0.0;
70
71 for idivy=1:ndivy
72
73 if(weity(idivy) == 0.0)
74 weity(idivy) =1.0;
75 end
76 total = total+weity(idivy);
77 end
78 ynorm=2.0/total;
79
80 nxtwo = ndivx*knode+1;
81 nytwo = ndivy*knode+1;
82
83 iasey = 0;
84 etasp =-1.0;
85 kwety = 0;
86 konty = -1;
87
88 %-----
89
90 for iytwo=1:nytwo %do 160
91
92 iasey = iasey+1;
93 if(nnode ~= 8 && iasey == 3)
94 iasey =2;
95 end
96
97 if(nnode == 8 && iasey == 4)
98 iasey =2;

```

```
99 end
100
101 iasex=0;
102 exisp=-1.0;
103 kwetx=0;
104 kontx=-1;
105
106 for ixtwo=1:nxtwo
107
108 iasex=iasex+1;
109 if(nnod~==8 && iasex == 3)
110 iasex=2;
111 end
112
113 if(nnod == 8 && iasex == 4)
114 iasex=2;
115 end
116
117 if(iasex == 2 && iasey ==2 &&
nnod == 8)
118
119 [kwetx,kontx,exisp] = goto_100
(exisp,xnorm,knode,fnode,kwetx,
kontx,weitx);
120
121 continue
122 end
123
124 npoin = npoin+1;
125
126 [shape] = sfrq(exisp,etasp);
127
128 for inode=1:lnode
129
130 jtemp=tnds(ibloc,inode);
131 for idime=1:ndime
132 coord(npoin,idime)=coord(npoin,
idime)+shape(inode) ...
133 *tcord(jtemp,idime);
134 end
135 end
136
137 if(knode == 1)
138 if(iasex ~== 2 || iasey ~==2)
139
140 [kwetx,kontx,exisp] = goto_100
(exisp,xnorm,knode,fnode,kwetx,
kontx,weitx);
141
142 continue
143 end
144
145
146 nelem=nelem+1;
147 jpoint=npoin-nxtwo;
148 lnds(nelem,1)=jpoint-1;
149 lnds(nelem,2)=jpoint;
150 lnds(nelem,3)=npoin;
151 lnds(nelem,4)=npoin-1;
152 matno(nelem)=tmato(ibloc);
153
154 [kwetx,kontx,exisp] = goto_100
(exisp,xnorm,knode,fnode,kwetx,
kontx,weitx);
155
156 continue
157 end
158
159 if(knode == 2)
160
161 if((iasex ~==3) || (iasey ~==3))
162
163 [kwetx,kontx,exisp] = goto_100
(exisp,xnorm,knode,fnode,kwetx,
kontx,weitx);
164
165 continue
166 end
167
168
169 nelem=nelem+1;
170 ipoin=npoin-ixtwo-ndivx+
(ixtwo-1)/2;
171 jpoint=npoin-nxtwo-ndivx-1;
172 lnds(nelem,1)=jpoint-2;
173 lnds(nelem,2)=jpoint-1;
174 lnds(nelem,3)=jpoint;
175 lnds(nelem,4)=ipoin;
176 lnds(nelem,5)=npoin;
177 lnds(nelem,6)=npoin-1;
178 lnds(nelem,7)=npoin-2;
179 lnds(nelem,8)=ipoin-1;
180 matno(nelem)=tmato(ibloc);
181
182 [kwetx,kontx,exisp] = goto_100
(exisp,xnorm,knode,fnode,kwetx,
kontx,weitx);
183
184
185 end
```

```

186 end
187
188 if(knode == 1)
189 kwety=kwety+1;
190 etasp=etasp+ynorm*weity(kwety) /
  fnode;
191 end
192 if(knode == 2)
193 if(konty < 0 )
194 kwety=kwety+1;
195 end
196 konty=konty*(-1);
197 etasp=etasp+ynorm*weity(kwety) /
  fnode;
198 end
199
200 end
201
202 end
203
204
205 %-----
206 % eliminate the duplicate nodes
207 % sharing same locations
208 %-----
209
210 nrepn=0;
211
212 for ipoin=1:npoint
213
214 if(nrepn ~= 0)
215 for irepn=1:nrepn
216 iflag=0;
217 if(ipoin == lrepn(irepn))
218 iflag=1;
219 break
220 end
221 end
222 end
223
224 if(iflag == 0)
225 lpoind=ipoin+1;
226 for jpoind=lpoind:npoint
227   total=abs(coord(ipoin,1)
228     -coord(jpoind,1)) + ...
229     abs(coord(ipoin,2)-coord
230       (jpoind,2));
231 if(total > 0.00001)
232   nrepn =nrepn+1;
233   lrepn(nrepn)=jpoind;
234   lasoc(nrepn)=ipoin;
235   %end
236 end
237
238
239 end
240 end
241
242 if(nrepn ~= 0)
243   index=0;
244   for ipoin=1:npoint
245     for irepn=1:nrepn
246       iflag=0;
247       if(lrepn(irepn) == ipoin)
248         iflag =1;
249         break;
250       end
251     end
252   end
253
254 if(iflag == 0)
255 continue
256 end
257
258 index=index+1;
259 lfinn(index)=lrepn(irepn);
260 lfasc(index)=lasoc(irepn);
261 end
262
263 for irepn=1:nrepn
264   lrepn(irepn)=lfinn(irepn);
265   lasoc(irepn)=lfasc(irepn);
266 end %do_250
267
268 for irepn=1:nrepn
269   for ielem=1:nelem
270     for inode=1:mnode
271       if(lnods(ielem,inode) == lrepn
272         (irepn))
273         lnods(ielem,inode)=lasoc(irepn);
274       end
275     end
276   end
277
278 for ipoin=1:npoint
279
280 for irepn=1:nrepn

```

```

281 iflag=0;                                331 if(nposi < lrepn(1))
282 if(ipoin == lrepn(irepn))               332 continue;
283 iflag=1;                                333 end
284 break;                                 334
285 end                                     335 idiff=nposi-nrepn;
286 end                                     336 if(nposi <= lrepn(nrepn) )
287                                         337 for irepn=1:nrepn
288 if(iflag == 1)                           338 krepn=nrepn-irepn+1;
289 continue;                             339 if(nposi < lrepn(krepn) )
290 end                                     340 idiff=nposi-krepn+1;
291                                         341 end
292                                         342
293 if(ipoin < lrepn(1))                  343 end
294 continue;                            344 end
295 end                                     345
296                                         346 lnods(ielem,inode)=idiff;
297 idiff=ipoin-nrepn;                   347 end
298 if(ipoin <= lrepn(irepn))             348 end
299 for irepn=1:nrepn                   349 end
300 krepn=nrepn-irepn+1;                350
301 if(ipoin < lrepn(krepn) )            351 npoin=npoin-nrepn;
302 idiff=ipoin-krepn+1;                 352
303 end                                     353 end %endfunction
304 end                                     354
305 end                                     355 =====
306
307 for idime=1:ndime
308 coord(idiff,idime)=coord
      (ipoin,idime);
309 end
310
311 end
312
313
314 for ielem=1:nelem
315 iflag=0;
316 for inode=1:mnode
317
318 nposi=lnods(ielem,inode);
319 for irepn=1:nrepn
320 iflag=0;
321 if(nposi == lrepn(irepn) )
322 iflag=1;
323 break;
324 end
325 end
326
327 if(iflag == 1)
328 continue;
329 end
330                                         331 if(nposi < lrepn(1))
332 continue;
333 end
334
335 idiff=nposi-nrepn;
336 if(nposi <= lrepn(nrepn) )
337 for irepn=1:nrepn
338 krepn=nrepn-irepn+1;
339 if(nposi < lrepn(krepn) )
340 idiff=nposi-krepn+1;
341 end
342
343 end
344 end
345
346 lnods(ielem,inode)=idiff;
347 end
348 end
349 end
350
351 npoin=npoin-nrepn;
352
353 end %endfunction
354
355 =====

```

Function**goto_100.m**

This function acts as an equivalent to the goto statements in the original code.

Listing:

```

1  function [kwetx,kontx,exisp] =
   goto_100(exisp,xnorm,knode,fnode,
            kwetx,kontx,weitx)
2
3  if(knode == 1)
4      kwetx = kwetx+1;
5      exisp=exisp+xnorm*weitx(kwetx) /
            fnode;
6  end
7
8
9  if(knode == 2)
10     if(kontx < 0)
11         kwetx = kwetx+1;
12     end

```

```

13
14 kontx=kontx*(-1);
15 exisp=exisp+xnorm*weitx(kwetx) /
fnode;
16 end
17
18
19 end %endfunction

```

Function

sfrq.m

This function evaluates the shape functions of eight-node isoparametric elements.

Listing:

```

1 function [shape] = sfrq(s,t)
2
3 ss=s*s;
4 tt=t*t;
5 st=s*t;
6 sst=s*s*t;
7 stt=s*t*t;
8
9 shape(1)=(-1.0+st+ss+tt-sst-stt)/
4.0;
10 shape(2)=(1.0-t-ss+sst)/2.0;
11 shape(3)=(-1.0-st+ss+tt-sst+stt)/
4.0;
12 shape(4)=(1.0+s-tt-stt)/2.0;
13 shape(5)=(-1.0+st+ss+tt+sst+stt)/
4.0;
14 shape(6)=(1.0+t-ss-sst)/2.0;
15 shape(7)=(-1.0-st+ss+tt+sst-stt)/
4.0;
16 shape(8)=(1.0-s-tt+stt)/2.0;
17
18 end %endfunction

```

Function

split .m

If the geometry is to be meshed with triangular isoparametric elements, first four node elements are generated in function gen_mesh4.m and these four node elements are divided along their shortest diagonal forming two triangular

elements in function split. The element connectivity list also updated resulting from this subdivision of four node elements.

Listing:

```

1 function [nelem,lnods,npoin,coord,
matno]=split(nelem,ndime,lnods,%
npoin,coord,nnode,matno)
3
4
5 kount=0;
6 mnode=4;
8
9 for ielem=1:nelem
10 notal=nelem+ielem;
11 matno(notal)=matno(ielem);
12 for inode=1:mnode
13 lnods(notal,inode)=lnods(ielem,
inode);
14 end
15 end
16
17 for ielem=1:nelem
18 notal=nelem+ielem;
19 for inode=1:mnode
20 index=lnods(notal,inode);
21 ltemp(inode)=index;
22 for idime=1:ndime
23 corde(inode,idime)=coord(index,
idime);
24 end
25 end
26
27 diag1=sqrt((corde(1,1)-corde
(3,1))^2+(corde(1,2)-...
corde(3,2))^2);
28 diag2=sqrt((corde(2,1)-corde
(4,1))^2+(corde(2,2)-...
corde(4,2))^2);
31 differ=diag1-diag2;
32 if(differ <=1.0e-9)
33 kount=kount+1;
34 lnods(kount,1)=ltemp(1);
35 lnods(kount,2)=ltemp(2);
36 lnods(kount,3)=ltemp(3);
37 matno(kount)=matno(notal);
38 kount=kount+1;
39 lnods(kount,1)=ltemp(1);
40 lnods(kount,2)=ltemp(3);

```

```

41 lnods(kount,3)=ltemp(4);
42 matno(kount)=matno(notal);
43 end
44 if(difer > 1.0e-9)
45 kount=kount+1;
46 lnods(kount,1)=ltemp(1);
47 lnods(kount,2)=ltemp(2);
48 lnods(kount,3)=ltemp(4);
49 matno(kount)=matno(notal);
50 kount=kount+1;
51 lnods(kount,1)=ltemp(2);
52 lnods(kount,2)=ltemp(3);
53 lnods(kount,3)=ltemp(4);
54 matno(kount)=matno(notal);
55 end
56 end
57 nelem=nelem*2;
58
59 end %endfunction

```

Function**output.m**

This function outputs the Cartesian coordinates of the nodes and the element connectivity list to file. This file needs to be supplemented with the control parameters, material parameters, and the boundary conditions following the format in *input_fem_pf.m* and *input_fem_elast.m* functions. The code also produces a graphical output file which can be viewed in gnuplot with the command, *plot “filename” w l*, as shown in Fig. C.1.

Listing:

```

1 function out_put(npoin,nelem,nnode,
      ndime,lnods,coord,matno)
2
3 global out1;
4 global out2;
5
6
7 %-----
8 % write to file with the FEM input
   file format
9 %-----
10
11 fprintf(out1,'%5d %5d\n',npoin,
      nelem);

```

```

12 for ielem=1:nelem
13 fprintf(out1,'%5d',ielem);
14 for inode=1:nnode
15 fprintf(out1,'%5d',lnods(ielem,
      inode));
16 end
17 fprintf(out1,'%5d',matno(ielem));
18 fprintf(out1,'\n');
19 end
20
21 for ipoin=1:npoin
22 fprintf(out1,'%5d',ipoin);
23 for idime=1:ndime
24 fprintf(out1,'%14.6e',coord(ipoin,
      idime));
25 end
26 fprintf(out1,'\n');
27 end
28
29 %-----
30 % write to file for the gnuplot
31 %-----
32
33 for ielem=1:nelem
34 fprintf(out2,'#element no: %5d\n',
      ielem);
35 for inode=1:nnode
36 lnode=lnods(ielem,inode);
37 fprintf(out2,'%14.6e %14.6e\n',
      coord(lnode,1),coord(lnode,2));
38 end
39 lnode=lnods(ielem,1);
40 fprintf(out2,'%14.6e %14.6e\n',
      coord(lnode,1),coord(lnode,2));
41 fprintf(out2,'\n');
42 end
43
44
45 end %endfunction

```

Example Input File**mesh_hole_1.inp**

Listing:

<pre> 1 40 8 4 2 2 1 1 2 3 9 14 13 12 8 1 3 2 3 4 5 10 16 15 14 9 1 4 3 5 6 7 11 18 17 16 10 1 </pre>

```

5 4 12 13 14 20 25 24 23 19 1 54 3 24 24
6 5 16 17 18 22 29 28 27 21 1 55 1.0 1.0
7 6 23 24 25 31 36 35 34 30 1 56 4 24 16
8 7 25 26 27 32 38 37 36 31 1 57 1.0 1.0
9 8 27 28 29 33 40 39 38 32 1 58 5 24 16
10 1 0.0 0.0 59 1.0 1.0
11 2 1.875 0.0 60 6 24 24
12 3 3.75 0.0 61 1.0 1.0
13 4 5.0 0.0 62 7 16 24
14 5 6.25 0.0 63 1.0 1.0
15 6 8.125 0.0 64 8 24 24
16 7 10.0 0.0 65 1.0 1.0
17 8 0.0 1.875
18 9 3.75 1.875
19 10 6.25 1.875
20 11 10.0 1.875
21 12 0.0 3.75
22 13 1.875 3.75
23 14 3.75 3.75
24 15 5.0 3.75
25 16 6.25 3.75
26 17 8.125 3.75
27 18 10.0 3.75
28 19 0.0 5.0
29 20 3.75 5.0
30 21 6.25 5.0
31 22 10.0 5.0
32 23 0.0 6.25
33 24 1.875 6.25
34 25 3.75 6.25
35 26 5.0 6.25
36 27 6.25 6.25
37 28 8.125 6.25
38 29 10.0 6.25
39 30 0.0 8.125
40 31 3.75 8.125
41 32 6.25 8.125
42 33 10.0 8.125
43 34 0.0 10.0
44 35 1.875 10.0
45 36 3.75 10.0
46 37 5.0 10.0
47 38 6.25 10.0
48 39 8.125 10.0
49 40 10.0 10.0
50 1 24 24
51 1.0 1.0
52 2 16 24
53 1.0 1.0

```

Line numbers:

1:	Number of nodes, number of subdivisions, element type for generated mesh (three is for triangular elements, four is four node elements, and eight is for eight node elements), number of materials.
2–9:	Subdivision number, next eight numbers are for subdivision connectivity list (subdivisions are described with eight nodes and their listing is anti-clock direction), material number for this subdivision.
10–49:	Node number, Cartesian coordinates of the node.
50:	Subdivision number, number of elements in the <i>x</i> -direction into which the subdivision is to be divided, number of elements in the <i>y</i> -direction into which the subdivision is to be divided.
51:	Weighting factors for the divisions in the <i>x</i> and <i>y</i> -directions, respectively.
52–65:	Lines 50 and 51 are repeated for each remaining subdivisions.

The tabulated output file, *mesh_hole_fem.inp*, needs to be supplemented with the control parameters, material parameters, and the boundary conditions following the format in *input_fem_pf.m* and *input_fem_elast.m*.

The graphical output given in Fig. C.1 shows the resulting FEM mesh with four-node isoparametric elements.

References

1. Hinton E, Owen DRJ (1979) An introduction to finite element computations. Pineridge Press, Swansea, UK

Appendix D

In this section, the necessary formulas are presented for the extension of the three solutions methodologies developed in the text to three-dimension.

The reader should be aware of the numbers of unknowns, hence, the computational effort required to solve the number of equations increases substantially in three-dimension. For example, in a simulation cell discretized with $128 \times 128 \times 128$ grid points in three-dimension, the number of unknowns are 2097152; whereas, a simulation cell discretized with 128×128 grid points yields only 16384 unknowns in two-dimensions.

For the finite difference and the Fourier spectral methods, the extension to three-dimension is straightforward and requires little effort. On the

other hand, the similar extension of finite-element method involves more substantial changes to the codes, even though the overall structures of the codes remain the same.

D.1 Extension of Two-Dimensional Finite Difference Method to Three-Dimension

For three-dimensional stencil, the finite difference approximation of the Laplace operator ∇^2 involves minimum seven points as shown in Fig. D.1.

With these seven points, the Laplace operator takes the form:

$$(\nabla^2 u_{i,j,k}) = \frac{u_{i-1,j,k} - 2u_{i,j,k} + u_{i+1,j,k}}{\Delta x^2} + \frac{u_{i,j-1,k} - 2u_{i,j,k} + u_{i,j+1,k}}{\Delta y^2} + \frac{u_{i,j,k-1} - 2u_{i,j,k} + u_{i,j,k+1}}{\Delta z^2} \quad (\text{D.1})$$

where Δx , Δy , and Δz are the grid spacing between two grid points in the x -, y -, and z -directions, respectively. In three-dimension, the periodicity now should be enforced for the grid points lying on the six surfaces, rather than the edges as in two-dimension.

D.2 Extension of Two-Dimensional Fourier Spectral Method to Three-Dimensions

A three-dimensional Fourier spectral phase-field method is already developed in *case*

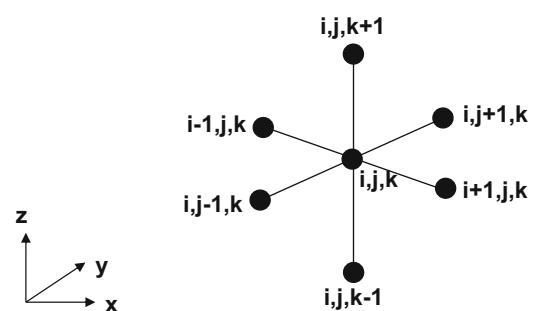


Fig. D.1 A schematic node ordering representation for three-dimensional finite difference grid

study-XVI with the source codes *pfc_3D_v1.m* (in longhand format) and *pfc_3D_v2.m* (optimized for Matlab/Octave). These two source codes will provide a template for the solution of other phase-field models by utilizing the Fourier spectral method. In addition, functions for three-dimensional Green's tensor are given in Appendix B.

As can be seen from these codes, apart from preparing Fourier coefficients for three-dimension, the solution utilizes two build in functions in Matlab/Octave. These are: *fftn()* for the forward Fourier transformations and *ifftn()* for the inverse Fourier transformations.

D.3 Extension of Two-Dimensional FEM Method to Three-Dimension

Similar to two-dimensional elements (Fig. 6.1), for three-dimensional eight-node and 20-node isoparametric elements the local coordinate system defined by ζ , η , and ξ is located at the center of the elements, as shown in Fig. D.2. They take the values of $-1 \leq \zeta \leq 1$, $-1 \leq \eta \leq 1$, and $-1 \leq \xi \leq 1$.

Their shape functions must satisfy the conditions:

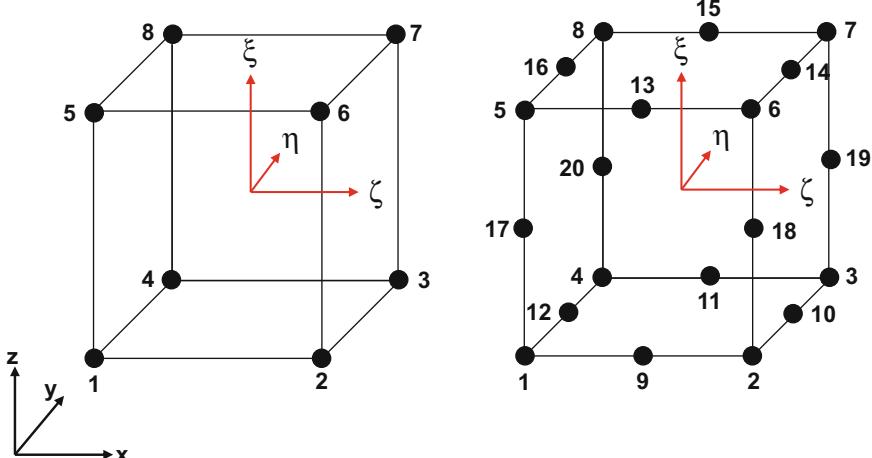


Fig. D.2 Global and local definition of eight-node and 20-node isoparametric elements

$$\sum N_i^{(e)}(\zeta, \eta, \xi) = 1 \quad (D.2)$$

and

$$N_i^{(e)}(\zeta_j, \eta_j, \xi_j) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \quad (D.3)$$

For eight-node isoparametric elements the shape functions meeting these requirements are:

$$\begin{aligned} N_1 &= \frac{1}{8}(1 - \zeta)(1 - \eta)(1 - \xi) \\ N_2 &= \frac{1}{8}(1 + \zeta)(1 - \eta)(1 - \xi) \\ N_3 &= \frac{1}{8}(1 + \zeta)(1 + \eta)(1 - \xi) \\ N_4 &= \frac{1}{8}(1 - \zeta)(1 + \eta)(1 - \xi) \\ N_5 &= \frac{1}{8}(1 - \zeta)(1 - \eta)(1 + \xi) \\ N_6 &= \frac{1}{8}(1 + \zeta)(1 - \eta)(1 + \xi) \\ N_7 &= \frac{1}{8}(1 + \zeta)(1 + \eta)(1 + \xi) \\ N_8 &= \frac{1}{8}(1 - \zeta)(1 + \eta)(1 + \xi) \end{aligned} \quad (D.4)$$

The corresponding shape functions for 20-node isoparametric elements for corner nodes are:

$$\begin{aligned}
N_1 &= -\frac{1}{8}(1-\zeta)(1-\eta)(1-\xi)(2+\zeta+\eta+\xi) \\
N_2 &= -\frac{1}{8}(1+\zeta)(1-\eta)(1-\xi)(2-\zeta+\eta+\xi) \\
N_3 &= -\frac{1}{8}(1+\zeta)(1+\eta)(1-\xi)(2-\zeta-\eta+\xi) \\
N_4 &= -\frac{1}{8}(1-\zeta)(1+\eta)(1-\xi)(2+\zeta-\eta+\xi) \\
N_5 &= -\frac{1}{8}(1-\zeta)(1-\eta)(1+\xi)(2+\zeta+\eta-\xi) \\
N_6 &= -\frac{1}{8}(1+\zeta)(1-\eta)(1+\xi)(2-\zeta+\eta-\xi) \\
N_7 &= -\frac{1}{8}(1+\zeta)(1+\eta)(1+\xi)(2-\zeta-\eta-\xi) \\
N_8 &= -\frac{1}{8}(1-\zeta)(1+\eta)(1+\xi)(2+\zeta-\eta-\xi)
\end{aligned} \tag{D.5a}$$

and for mid side nodes are

$$\begin{aligned}
N_9 &= \frac{1}{4}(1-\zeta)(1+\zeta)(1-\eta)(1-\xi) & N_{10} &= \frac{1}{4}(1-\eta)(1+\eta)(1+\zeta)(1-\xi) \\
N_{11} &= \frac{1}{4}(1-\zeta)(1+\zeta)(1+\eta)(1-\xi) & N_{12} &= \frac{1}{4}(1-\eta)(1+\eta)(1-\zeta)(1-\xi) \\
N_{13} &= \frac{1}{4}(1-\zeta)(1+\zeta)(1-\eta)(1+\xi) & N_{14} &= \frac{1}{4}(1-\eta)(1+\eta)(1+\zeta)(1+\xi) \\
N_{15} &= \frac{1}{4}(1-\zeta)(1+\zeta)(1+\eta)(1+\xi) & N_{16} &= \frac{1}{4}(1-\eta)(1+\eta)(1-\zeta)(1+\xi) \\
N_{17} &= \frac{1}{4}(1-\xi)(1+\xi)(1-\zeta)(1-\eta) & N_{18} &= \frac{1}{4}(1-\xi)(1+\xi)(1+\zeta)(1-\eta) \\
N_{19} &= \frac{1}{4}(1-\xi)(1+\xi)(1+\zeta)(1+\eta) & N_{20} &= \frac{1}{4}(1-\xi)(1+\xi)(1-\zeta)(1+\eta)
\end{aligned} \tag{D.5b}$$

Any point (ζ, η, ξ) within an element the x -, y -, and z -coordinates are again obtained from isoparametric representation:

$$\begin{aligned}
x(\zeta, \eta, \xi) &= \sum_{i=1}^n N_i^{(e)} x_i^{(e)} \\
y(\zeta, \eta, \xi) &= \sum_{i=1}^n N_i^{(e)} y_i^{(e)} \\
z(\zeta, \eta, \xi) &= \sum_{i=1}^n N_i^{(e)} z_i^{(e)}
\end{aligned} \tag{D.6}$$

where $N_i^{(e)}(\zeta, \eta, \xi)$ are the shape functions of the element (Eqs. D.4, D.5a and D.5b) and $x_i^{(e)}$, $y_i^{(e)}$, $z_i^{(e)}$ are the nodal coordinates in the Cartesian coordinate system.

The Cartesian derivative of any function, f , defined over the element follows the similar expressions (Eqs. 6.2 through 6.6). Thus, in three-dimension:

$$f(\zeta, \eta, \xi) = \sum_i^n N_i^{(e)} f_i^{(e)} \tag{D.7}$$

in which $f_i^{(e)}$ is the value of f at node i . Utilizing the chain rule of differentiation:

$$\begin{aligned}\frac{\partial f}{\partial x} &= \frac{\partial f}{\partial \zeta} \cdot \frac{\partial \zeta}{\partial x} + \frac{\partial f}{\partial \eta} \cdot \frac{\partial \eta}{\partial x} + \frac{\partial f}{\partial \xi} \cdot \frac{\partial \xi}{\partial x} \\ \frac{\partial f}{\partial y} &= \frac{\partial f}{\partial \zeta} \cdot \frac{\partial \zeta}{\partial y} + \frac{\partial f}{\partial \eta} \cdot \frac{\partial \eta}{\partial y} + \frac{\partial f}{\partial \xi} \cdot \frac{\partial \xi}{\partial y} \\ \frac{\partial f}{\partial z} &= \frac{\partial f}{\partial \zeta} \cdot \frac{\partial \zeta}{\partial z} + \frac{\partial f}{\partial \eta} \cdot \frac{\partial \eta}{\partial z} + \frac{\partial f}{\partial \xi} \cdot \frac{\partial \xi}{\partial z}\end{aligned}\quad (\text{D.8})$$

where

$$\frac{\partial f}{\partial \zeta} = \sum_i^n \frac{\partial N_i^{(e)}}{\partial \zeta} \cdot f_i^{(e)}, \quad \frac{\partial f}{\partial \eta} = \sum_i^n \frac{\partial N_i^{(e)}}{\partial \eta} \cdot f_i^{(e)}, \quad \frac{\partial f}{\partial \xi} = \sum_i^n \frac{\partial N_i^{(e)}}{\partial \xi} \cdot f_i^{(e)} \quad (\text{D.9})$$

The terms in Eq. D.8 $\frac{\partial \xi}{\partial x}, \frac{\partial \eta}{\partial x}, \dots$ are obtained, again, forming Jacobian matrix similar to Eq. 6.5:

$$J^e = \begin{bmatrix} \sum_i^n \frac{\partial N_i^{(e)}}{\partial \zeta} \cdot x_i^{(e)} & \sum_i^n \frac{\partial N_i^{(e)}}{\partial \zeta} \cdot y_i^{(e)} & \sum_i^n \frac{\partial N_i^{(e)}}{\partial \zeta} \cdot z_i^{(e)} \\ \sum_i^n \frac{\partial N_i^{(e)}}{\partial \eta} \cdot x_i^{(e)} & \sum_i^n \frac{\partial N_i^{(e)}}{\partial \eta} \cdot y_i^{(e)} & \sum_i^n \frac{\partial N_i^{(e)}}{\partial \eta} \cdot z_i^{(e)} \\ \sum_i^n \frac{\partial N_i^{(e)}}{\partial \xi} \cdot x_i^{(e)} & \sum_i^n \frac{\partial N_i^{(e)}}{\partial \xi} \cdot y_i^{(e)} & \sum_i^n \frac{\partial N_i^{(e)}}{\partial \xi} \cdot z_i^{(e)} \end{bmatrix} \quad (\text{D.10})$$

and inverting it:

$$[J^e]^{-1} = \frac{1}{\det J^e} \begin{bmatrix} \frac{\partial \zeta}{\partial x} & \frac{\partial \eta}{\partial x} & \frac{\partial \xi}{\partial x} \\ \frac{\partial \zeta}{\partial y} & \frac{\partial \eta}{\partial y} & \frac{\partial \xi}{\partial y} \\ \frac{\partial \zeta}{\partial z} & \frac{\partial \eta}{\partial z} & \frac{\partial \xi}{\partial z} \end{bmatrix} \quad (\text{D.11})$$

where $\det J^e$ is the determinate of the Jacobian matrix. Any function $f(x, y, z)$ defined over the elements can be integrated as:

$$\iiint f(x, y, z) dx dy dz = \int_{-1}^1 \int_{-1}^1 \int_{-1}^1 f(\zeta, \eta, \xi) \det J^e d\zeta d\eta d\xi = \int_{-1}^1 \int_{-1}^1 \int_{-1}^1 g(\zeta, \eta, \xi) d\zeta d\eta d\xi \quad (\text{D.12})$$

By utilizing Gauss–Legendre rule similar to two-dimensional case, Eq. 6.8:

$$\int_{-1}^1 \int_{-1}^1 \int_{-1}^1 g(\zeta, \eta, \xi) d\zeta d\eta d\xi = \sum_k^m \sum_j^m \sum_i^m W_i W_j W_k g(\bar{\zeta}_i, \bar{\eta}_j, \bar{\xi}_k) \quad (\text{D.13})$$

in which W_i , W_j , and W_k are the weight coefficients and $g(\bar{\zeta}_i, \bar{\eta}_j, \bar{\xi}_k)$ is the value of the function at the sampling point $(\bar{\zeta}_i, \bar{\eta}_j, \bar{\xi}_k)$.

For linear elasticity, again Hook's law provides the constitutive relationship between the stresses and strains within the element. Thus,

$$\begin{aligned} \sigma &= D\epsilon \\ \sigma &= [\sigma_{11}, \sigma_{22}, \sigma_{33}, \tau_{12}, \tau_{23}, \tau_{13}]^T \\ \epsilon &= [\epsilon_{11}, \epsilon_{22}, \epsilon_{33}, \gamma_{12}, \gamma_{23}, \gamma_{13}]^T \end{aligned} \quad (\text{D.14})$$

$$D = \frac{E(1-\nu)}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1 & \frac{\nu}{1-\nu} & \frac{1}{1-\nu} & 0 & 0 & 0 \\ & 1 & \frac{\nu}{1-\nu} & 0 & 0 & 0 \\ & & 1 & 0 & 0 & 0 \\ & & & \frac{(1-2\nu)}{2(1-\nu)} & 0 & 0 \\ & & & & \frac{(1-2\nu)}{2(1-\nu)} & 0 \\ & & & & & \frac{(1-2\nu)}{2(1-\nu)} \end{bmatrix} \quad (\text{D.15})$$

in which E is the Young's modulus and ν is the Poisson's ratio. If the displacement vector in the Cartesian coordinates, x -, y -, z -directions is

$$\begin{aligned} \epsilon_{11} &= \frac{\partial u}{\partial x}, \quad \epsilon_{11} = \frac{\partial v}{\partial y}, \quad \epsilon_{33} = \frac{\partial w}{\partial z} \\ \gamma_{12} &= \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}, \quad \gamma_{23} = \frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}, \quad \gamma_{13} = \frac{\partial w}{\partial x} + \frac{\partial u}{\partial z} \end{aligned} \quad (\text{D.16})$$

in which σ_{11} , σ_{22} , σ_{33} and ϵ_{11} , ϵ_{22} , ϵ_{33} are the normal stress and strain components, respectively, and τ_{12} , τ_{23} , τ_{13} and γ_{12} , γ_{23} , γ_{13} are the shear stresses and strains, respectively.

For isotropic linear elastic material the stress-strain matrix, D , is:

defined as $\delta = [u, v, w]^T$, then strain terms appearing in Eq. D.14 are obtained from:

and they can be calculated at the element integration points as

$$\boldsymbol{\epsilon} = \mathbf{B}\boldsymbol{\delta}^{(e)} = \sum_i^n B_i \delta_i \quad (\text{D.17})$$

and the strain matrix \mathbf{B} takes the form of:

$$B_i = \begin{bmatrix} \frac{\partial N_i}{\partial x} & 0 & 0 \\ 0 & \frac{\partial N_i}{\partial y} & 0 \\ 0 & 0 & \frac{\partial N_i}{\partial z} \\ \frac{\partial N_i}{\partial y} & \frac{\partial N_i}{\partial x} & 0 \\ 0 & \frac{\partial N_i}{\partial z} & \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} & 0 & \frac{\partial N_i}{\partial x} \end{bmatrix} \quad (\text{D.18})$$

Apart from representation of nodal coordinates with a three-dimensional array, the element connectivity list should also follow the node numbering sequence in Fig. D.2 for the three-dimensional elements in the input files.

Index

A

Adams–Bashforth method, 11
Adams–Moulton method, 11
Al–Cu alloy system, 339
Allen–Cahn equations, 1, 3, 41, 44, 110, 113, 117, 122, 218, 219, 222, 274, 279, 310
Anisotropy (in interfacial energy), 69–71
Applied force vector, 175

B

Backward (implicit) Euler method, 11, 174
Basic Linear Algebra Subroutines (BLAS), 13
Bi-crystal
 classical PFC model, 355
 configuration, 356, 363
 constant deformation rate, 362
 deformation studies, 362
 dislocations, 364
 free energy, 361
 free energy density, 363
 kinetic evolution equation, 361
 mechanical properties, 363
 microstructure, 362, 363
 MPFC, 361
 objective, 361
 pfc_2D_v2.m, 364
 pfc_def_v2.m, 364, 366–368
 plastic deformation, 361
 precipitation behavior, 160
 short-range interactions, 361
 strain rate, 362
Brittle fracture, 303
Bulk diffusion, 51, 60, 63
Burger’s vector, 158

C

Cahn–Hilliard equation, 2, 26, 29, 102, 138, 156, 196, 197, 206, 211, 238, 239, 243, 248, 252
Cahn–Hilliard model, 21
Cahn–Hilliard phase-field equation, 104, 107, 144, 147
CALculation of PHAse Diagrams (CALPHAD), 121
Cancer metastasis, 82
Capacity matrix, 173
Cartesian coordinate system, 170, 174

Cartesian coordinates, 239
Cell motility, 81
Chain rule of differentiation, 170, 173, 174
Chemotaxis, 82
Collocation method, 10
Conductivity matrix, 173
Cottrell atmosphere, 156
Crank–Nicholson method, 11, 174
Cubic anisotropy, 140
Cubic Hermite elements, 197

D

Deformation
 bi-crystal (*see* Bi-crystal)
 morphology, 356
Dendritic-solidification, 2, 69–73, 77
Density functional theory (DFT), 338
Dislocations, 238
 dipole, 159
 lattice defects, 355
 and motions, 361
 plastic deformation, 361
 polycrystal microstructure, 341
 pre-melting at grain boundaries, 339

E

Edge dislocation dipole, eigenstrain approach, 157
Eigenstrains, 238, 265, 273, 279
 Cr precipitates, 157
 dependency of, 138
 dislocations, 166
 elastic energy, 157
 misfit strains, 140
 nonzero, 157
 position- and composition-dependent, 138
Elastic energy, 273
Elastic inhomogeneities and applied stresses, phase separation
 elastic anisotropy, 144
 fft_raft_v1.m program, 144–147
 fft_raft_v2.m program, 147–149
 Fourier space, 140
 green_tensor.m function, 149, 150
 grid points with algorithm, 140

- Elastic inhomogeneities and applied stresses, phase separation (*cont.*)
 hard particles, 141, 143
 intrinsic stress-strain fields, 138
 Kronecker delta function, 140
 matrix and precipitation phases, 141
 mechanical equilibrium, 138–140
 nickel-based super alloys, 138
 objective, 137
 phase-field model, 138
 phase separation behavior, 140
 solve_elasticity_v1.m function, 150–153
 solve_elasticity_v2.m function, 153–155
- Elasticity matrix, 174
- Elastic-plastic behavior, 303, 336
- Elastic-strain energy, 159
- 475 °C embrittlement, 156
- Energy barrier, 197, 238, 274, 278, 306
- Euler finite difference algorithm, 111
- Euler method, 174
- Euler time marching scheme, 104
- Explicit Euler finite difference algorithm, 112, 114
- External loading, 303, 310
- F**
- Fe–Cu–Mn–Ni alloy
 CALCulation of PHase Diagrams, 121
 Cu precipitates, 124
 Cu-rich phases, 121
 Fe_Cu_Mn_Ni_free_energy.m function, 132–137
 fft_FeCuNiMn_v1.m program, 125–128
 fft_FeCuNiMn_v2.m program, 128–131
 Init_FeCuNiMn_micro.m function, 131, 132
 kinetic Monte Carlo algorithm, 121
 Ni–Mn-rich intermetallic B2 rings, 121
 non-conserved phase transformations, 124
 objective, 121
 phase-field model, 122–123
- FEM discretization, 172–173
- Finite difference algorithms
 backward difference, 17
 centered difference, 17
 centered second difference, 17
 description, 17
 disadvantages, 17
 five-point stencil, 18
 forward difference, 17
 one-dimensional transient heat conduction, 19
 source codes, 19–21
- Finite difference method, 10, 17, 22
- Finite element analysis (FEA), 169
- Finite element method (FEM), 10, 169, 196, 218
- First-order phase transformations, 197, 273
- Five-point stencil, 18, 34
- Force vector
 due to body forces, 175
 due to initial internal strains, 175
 due to initial internal stresses, 175
- Forward (explicit) Euler method, 11, 174
- Fourier-spectral method, 340, 343, 346, 348, 351, 355, 364
- binary alloy with semi-implicit
 Cahn–Hilliard equation, 102, 103
 explicit Euler time marching scheme, 104
 fft_ch_v1.m program, 104–107
 fft_ch_v2.m program, 107–109
 numerical implementation, 103
 objective, 102
 Ostwald ripening process, 103
 phase-field model, 102–103
 prepare_fft.m function, 109, 110
- grain growth with semi-implicit
 evolution kinetics with identical simulation parameters, 112
 fft_ca_v1.m program, 113–117
 fft_ca_v2.m program, 117–120
 free_energ_fft_ca_v1.m function, 120
 free_energ_fft_ca_v2.m function, 120, 121
 non-conserved Allen–Cahn equation, 110
 numerical implementation, 111
 objective, 110
 phase-field model, 110–111
 simulations, 111
- one-dimensional transient heat conduction
 coarse spatial and temporal discretization, 101
 constant thermal conductivity, density and heat capacity, 99
 forward Euler time marching, 100
 forward Fourier transform, 99
 Fourier space, 99
 heat_1d_fft.m program, 100–102
- Fracture, 303, 308, 310
- Free-energy, 2–5, 70
- Free-energy density, 363
- FreeMat, 13
- Front-tracking method, 36
- G**
- Galerkin method, 10
- Gauss–Legendre rule, 171, 199
- Gibbs energy, 123
- Ginzburg–Landau equation, 274
- Ginzburg–Landau model, 337
- Ginzburg–Landau type, 70
- Global stiffness matrix, 220, 240
- Gnuplot, 15
- Grain boundary
 diffusion, 60, 63
 dislocations, 356
 migration, 51
 PFC model, 361
 spinodal decomposition, 339
- Grain growth, 35–37, 40, 51, 82, 83, 218, 220–222, 237
 bicrystal configuration, 356
 density field, 355 (*see also* Fourier-spectral method, grain growth with semi-implicit)
- MD, 355
- mechanical and physical properties, 355

- mutual annihilations in shrinking process, 356
objective, 355
PFC, 355
`pfc_2D_v1.m` and `pfc_2D_v2.m`, 356
`pfc_poly_v1.m`, 355, 356, 358–361
phase-field crystal model, 355
source codes, 356–361
Green's tensor, 140, 144, 146, 147, 149–151, 153, 161,
163, 164, 166
Green's theorem, 172
- H**
Heat balance, 171
`heat_1d_fft.m` program, 100
Hook's law, 174
- I**
Implicit-Euler time integration, 239
Isoparametric representation, 169, 171
Isothermal phase transformation, 125
- J**
Jacobian matrix, 170, 171, 173, 174
- K**
Kronecker delta function, 140, 158, 238
- L**
Landau polynomial, 274
Langevin noise, 5
Linear Algebra Package (LAPACK), 13
Linear elasticity, 176
- M**
Martensite, 273
Martensitic transformations, 272, 273, 298, 303
Material-specific properties, 369
Matlab/Octave implementation, 169
Matlab/Octave optimized mode, 243
Matlab programming
 features, 13
 introduction, 13
 linear algebra computations, 13
 modular approach, 14
 syntax, 14
Metastable body-centered cubic (*bcc*) phase, 121
Method of weighted residual, 9
Microstructures, 1, 3–5
 deformation simulation, 364
 triangular and stripe phases, 342
Misfit strains, 140, 141, 144, 157, 158
Molecular dynamics (MD)
 classical, 355
 simulations, 361
Monte Carlo/Potts model, 36, 82
Multi-cellular systems, 81
Multi-component phase-field model, 110, 121
- N**
Nano-crystalline materials, 355
Newton–Raphson algorithm, 198, 204, 206, 219
Newton–Raphson equation, 240
Newton–Raphson scheme, 276, 307
Newton–Raphson solution algorithm, 211
Nondimensional parameters, 142
Nucleation, 5
Numerical algorithm, 369
Numerical integration of isoparametric elements,
169–171
- O**
Octave programming, 13, 14
Ostwald ripening process, 23, 103, 199
- P**
Paraview, 15
Partial differential equation (PDE)
 equation, 9
 spatial discretization, 9
Penalty term, 306
Phase separation, 21, 23, 24, 273, 274. *See also*
 Elastic inhomogeneities and applied stresses
microstructure, 104
semi-implicit spectral and explicit Euler finite
difference, 105
- Phase-field crystal model (PFC)
 adoptive meshing, 340
 application, 339
 atomistic events, 337
 body-centered cubic, 341
 classical, 340
 codes `pfc_2D_v1.m` and `pfc_2D_v2.m`, 356
 constant (liquid) phase, 338
 deformation behavior of bicrystal, 361–363
 deformation simulation, 362
 elastic strain energy, 338
 face-centered cubic, 338
 Fourier spectral method, 340
 free energy, 337, 338
 free energy functional, 340
 Ginzburg–Landau theory, 337
 grain growth, 355, 356
 mutual annihilations in shrinking process, 356
 numerical implementation, 340
 objective, 339
 order parameter, 337
 parabolic equation, 340
 `pfc_2D_v2.m`, 346–348
 `pfc_3D_v1.m`, 348–351
 `pfc_3D_v2.m`, 351–353
 `pfc_def_v2.m`, 364–368
 `pfc_poly_v1.m`, 356, 358–361
 `prepare_fft_3d.m`, 353
 real Al–Cu alloy system, 339
 simulations, 341

- Phase-field crystal model (PFC) (*cont.*)
 source code *pfc_2D_v1.m*, 341, 343–345
write_vtk_grid_values_3D.m, 354
- Phase-field fracture model, 303
- Phase-field method
 Allen–Cahn dynamics, 3
 dendritic solidification, 2
 free energy, 2
 Ginzburg–Landau equation, 1
 Langevin noise, 5
 microstructure, 1
 solid–liquid transformations, 3
 Steinbach’s models, 4
 Tiaden’s model, 3, 4
- Phase-field models, 22, 82–83, 124, 197, 219, 237, 238, 273–274, 303–305, 369
- Plane-strain, 175, 176, 308
- Plane-stress, 174, 183
- Polycrystal
 microstructure, 341
 phase-field crystal model, 355
- Polycrystalline microstructure, 113
- Principles of virtual work, 174–176
- R**
- Radau rule, 171
- Read this book on SpringerLink, 15
- Readers, 369
- Runge–Kutta methods, 11
- S**
- Scilab, 13
- Semi-implicit Fourier spectral method, 112, 114
- Shape functions, 169, 170, 172–174, 179, 182, 183, 197, 198, 205, 216, 219, 220, 239, 275, 283, 305
- Sharp-interface approach, 1
- Simulation-specific properties, 369
- Sintering (solid-state), 51–54, 57, 61, 83
- Solute drag, 339
- Source codes, 176–196
- Spectral accuracy, 99
- Spectral method, 10
- Spherical bicrystal configuration, 356
- Spherical grain, time evolution, 112
- Spinodal decomposition, 21, 338, 339
 Fe–Cr alloy
 Burger’s vector, 157
 Cottrell atmosphere, 156
dislo_strain.m function, 166, 167
- elastic inhomogeneity, 157
- engineering materials, 156
- FeCr_chem_potent_v1.m function, 167
- FeCr_chem_potent_v2.m function, 167, 168
- Fe-rich α and Cr-rich α' phases, 156
- fft_FeCr_v1.m program, 161–163
- ft_FeCr_v2.m program, 164–166
- initial concentration field, 160
- lattice defects, 156
- microstructure, 159
- misfit strains, Cr phase, 158
- nonzero eigenstrain terms, 157
- objective, 156
- phase-field model, 156, 157
- precipitation behavior, 160
- simulations, 159
- FEM implementation, 197
- numerical implementation, 197–199
- objectives, 196
- phase-field model, 197
- results, 199
- Steinbach’s models, 4
- Stiffness matrix, 175
- Stress-relief mechanism, 238
- Stress–strain fields, 239
- Strong and weak forms, FEM formulation, 169, 171–174
- Surface diffusion, 51, 53, 60, 63
- Surface evolver, 36
- Surface heat transfer matrix, 173
- Surface integral, 172
- Syntax of Matlab/Octave, 369
- T**
- Tau method, 10
- Test/weight function, 9
- Thermo-Calc, 121
- Thin-films, 238, 242, 271
- Tiaden’s model, 3, 4
- Time evolution, 23
- Transient heat conduction, 17, 19, 99, 171
- Transient heat transfer, 173–174
- Trial function, 9, 10
- V**
- Vertex dynamics, 36, 82
- W**
- Weak form formulation, 172–173, 197, 239, 275, 305