Universidade do Minho
Escola de Engenharia

Alfredo Manuel da Silva Gonçalves de Moura

Simulation of the Nucleation of the Precipitate Al_3Sc in an Aluminum Scandium Alloy using the Kinetic Monte Carlo Method

Tese de Mestrado em Micro/Nano Tecnologias

Trabalho efetuado sob a orientação do
Professor Doutor António Joaquim André Esteves
Departamento de Informática, Universidade do Minho

Dezembro 2012

DECLARAÇÃO

Alfredo Manuel da Silva Gonçalves de Moura

Endereço eletrónico: alfredo.moura@gmail.com
Telefone: (00351) 968483543

Número do Bilhete de Identidade: 12037566

Título da Tese:

Simulation of the nucleation of the precipitate Al_3Sc in an Aluminum Scandium alloy using the kinetic Monte Carlo method

Orientador:

Professor Doutor António Joaquim André Esteves

Departamento de Informática

Escola de Engenharia

Universidade do Minho

Ano de conclusão: 2012

Designação do mestrado: Mestrado em Micro/Nano Tecnologias

Universidade do Minho, __/__/____

Assinatura:_____

Acknowledgements

I would like to thank a number of people, who have encouraged me over the last year to put this master thesis in one final piece. Namely first and foremost, my master thesis supervisor Professor António Joaquim André Esteves whose knowledge, guidance and support has been incredible and very much appreciated. I would like to thank the Informatics Department of the University of Minho for allowing the opportunity to submit my simulation jobs to the SeARCH cluster. Also but not least, I would like to thank my fellow colleagues with whom I collaborate with as a researcher at the Institute of Polymers and Composites (IPC) for their understanding and comprehension for the time that I dispended on this thesis.

Resumo

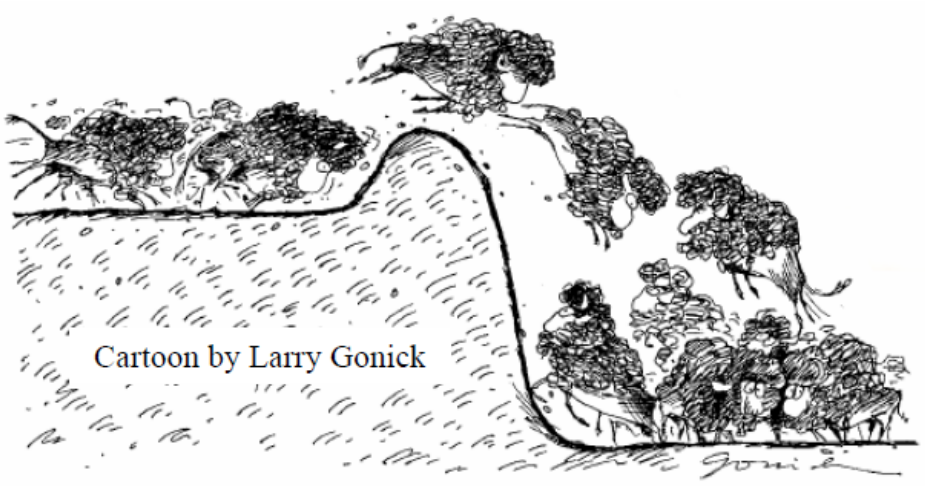
As estruturas de precipitados desempenham uma função fundamental nas ciências dos materiais devido à capacidade de obstruir o movimento de deslocamentos dentro do material.

Esta tese de mestrado debruça-se sobre uma aplicação baseada na mecânica estatística, nomeadamente o método Monte Carlo, no estudo e previsão do fenómeno de precipitação em ligas de alumínio. A liga de alumínio em estudo é a liga alumínio escândio.

Esta tese aborda temas como a mecânica computacional, mecânica estatística, ciências dos materiais, difusão, e ainda métodos de mineração de dados (*data mining*). A tese descreve as condições que influenciam a precipitação e como controlar este fenómeno.

O resultado desta tese é um conjunto de aplicações de *software* que permitem (i) efetuar a simulação de Monte Carlo, (ii) analisar os resultados usando a técnica de mineração DBSCAN e (iii) comparar os resultados da simulação com a teoria clássica da nucleação. Os resultados práticos obtidos com estas aplicações são:

- Relatórios da simulação, da análise dos *clusters*/precipitados com o algoritmo DBSCAN e da aplicação da teoria clássica da nucleação;
- Ficheiros para visualização 3D da simulação (em vários pontos ao longo do tempo);
- Ficheiros para visualização 3D dos precipitados.



Larry Gonick é um cartoonista que desenhou *cartoons* para a revista Discover. É autor de um vasto conjunto de livros das quais quero mencionar: "*The Cartoon Guide to Statistics*" da qual esta figura foi retirada e que me parece adequado na forma como ilustra vários temas que esta tese aborda: estatística, aleatoriedade, "salto" e "barreira".

Abstract

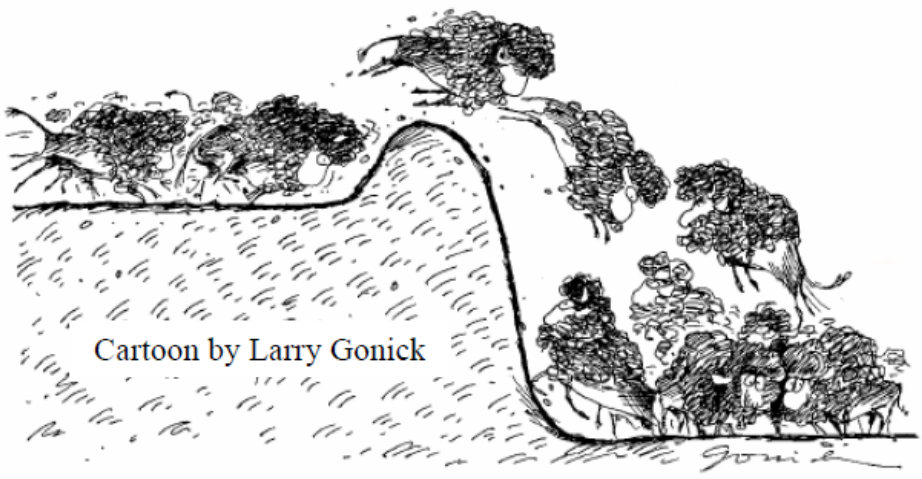
Precipitate structures play a fundamental function in the material science due to the capacity of representing strong obstacles for dislocations movements within the material.

This master thesis focuses on the elaboration and application of mechanical statistics knowledge, namely the kinetic Monte Carlo method, on the study and prediction of the phenomenon of precipitation in an aluminum alloy. The alloy under analysis is the aluminum scandium alloy.

This thesis tackles subjects such as computational mechanics, mechanical statistics (the kinetic Monte Carlo method), material science, the precipitation phenomenon, the diffusion phenomenon, what influences this phenomenon and how to control it and also predict it, as well as data mining (namely clustering).

The outcome of this thesis is a set of software applications that allow (i) to perform Monte Carlo simulations, (ii) to analyze the results using the DBSCAN clustering technique, and (iii) to compare the simulation results with the classical nucleation theory. Practical results obtained with these applications are:

- Reports about the simulation, the analysis of clusters/precipitates with DBSCAN algorithm, and the application of the classical nucleation theory;
- Files for 3D visualization of the simulation (at various points over time);
- Files for 3D visualization of the precipitates.



Larry Gonick is a cartoonist that sketched the bimonthly “Science Classics” cartoon for the science magazine Discover besides being the author of several books as for example: The Cartoon Guide to Statistics. The figure above was taken from this book and by which I do think it is very adequate in illustrating subjects addressed by this thesis, such as statistics, randomness, jump and barrier.

Index

Index of Figures

Index of Tables

Index of Equations

1. Introduction

1.1 Aim and scopes

The main goal of this thesis is to implement a Monte Carlo simulation method to predict and analyze the evolution of the precipitation phenomenon in an aluminum scandium alloy.

To address this task, different matters were incorporated in this thesis, such as material science, statistical mechanics, computation and data mining.

The knowledge about material science is essential for the comprehension of the structure of material, the behavior of material, the understanding of the classical nucleation theory, the notion of diffusion as well as the behavior of diffusion. The knowledge about statistical mechanics is of fundamental importance to predict the behavior of a large quantity of particles. The knowledge on data mining reveals to be essential to extract relevant information from raw data. This thesis, regarding data mining, will focus on the density based clustering with noise method (DBSCAN).

The choice on studying an aluminum scandium alloy is reasoned by the growing interest of the aluminum scandium alloy, as scientific publications can prove this increasing interest over time. Figure 1 confirms this interest and this work will cover the justification of such an interest.

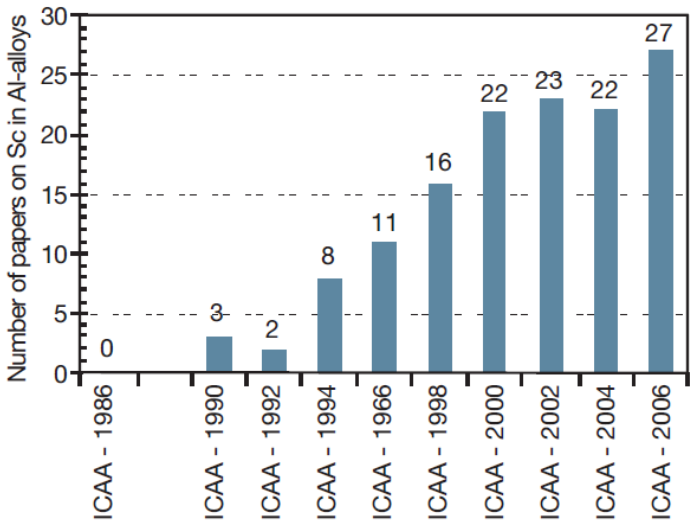


Figure 1 – Number of publications related to the subject of aluminum scandium alloys [Röyset].

1.2 Materials evolution

It is known from the history of mankind that the great leaps in our collective evolution are linked with the development in the materials technology. Historians have named such distinct steps in mankind history such as the "stone age", the "bronze age" and so forth. Over the last 350 years, significant material development moments occurred, some of them must be highlighted:

- 1665 - Robert Hooke. In this very year, Robert Hooke reveals levels of material microstructures never seen before in the publication of Micrografia;
- 1808 - John Dalton. In this year, John Dalton establishes the atomic theory;
- 1824 – In this year, Portland cement is invented by Joseph Aspdin;
- 1839 – Vulcanization is discovered accidentally by Charles Goodyear;
- 1856 - Large-scale steel production is patented by Henry Bessemer;
- 1869 - Mendeleev and Meyer published the Periodic Table of the Chemical Elements;
- 1886 – Cost-effective methods for aluminum production are discovered independently by Charles Hall and Paul Heroult;
- 1900 - Max Planck in this year, formulates the idea of quanta and therefore, quantum mechanics;
- 1909 – Bakelite (an entirely synthetic plastic) is patented by Leo Baekeland;
- 1921 - A. A. Griffith describes the effect of defects in fracture strength;
- 1928 – Hermann Staudinger mentions that polymers are made of small molecules that are linked to form chains;
- 1955 - Synthetic diamond is created by a team of scientists at General Electric;
- 1970 - Optical fibers is developed by researchers at Corning;
- 1985 - First university initiatives that attempt computational materials design also known as "materials by design";
- 1985 - Bucky balls (C60) is discovered at Rice University;
- 1991 - Carbon nanotubes are discovered by Sumio Iijima.

It is well true that our current moment of the mankind history will be "branded" by the future historians, but it is undeniable that it is a period of enormous advances and breakthroughs in the science of materials as for the applicability of this knowledge at the service of mankind.

It is acknowledged that very few metals are used in a state of purity. What mainly happens is that other elements, designated as components, are added and therefore creating alloys and as a consequence, modifying the mechanical properties. The added elements, which are designated as alloying elements, always dissolve in the base metal and to form a solid solution. The solubility can vary between less than 0.01% to 100% and depends on the combination of the elements.

The addition of alloys introduces profound modifications on the overall properties of metals. The majority of alloy additions lead to multiphase materials which are stronger than the single-phase materials. One result of alloying addition is fine grain size. Another phenomenon that plays an extremely important role in the evolution of

microstructures of metallic alloys during a thermo-mechanical treatment is designated as nucleation. This phenomenon can be divided into two possible processes: a phase transformation process (designated as precipitation) and a structural instability process (designated as recrystallization) [Bréchet2006].

The precipitation phenomenon is of most importance in metallic alloys microstructure. Precipitates influence directly the mechanical properties, namely strength. Indirectly, precipitation influences the recrystallization phenomenon and also grain growth. The strength of an alloy depends on the interaction of moving dislocations with precipitates. The capacity to block the motion of dislocations is not only the precipitate itself but also the strain field surrounding the very same precipitate [Marceau2008]. This capacity of dislocation obstruction is implemented through interaction mechanisms, which are a few. Literature classifies two categories of interaction mechanisms which depend mainly on the precipitates size. The categories are particle shearing and particle bypassing or looping [Marceau2008].

Particle shearing includes the mechanisms of:

- Chemical strengthening;
- Stacking-fault strengthening;
- Modulus strengthening;
- Coherency strengthening;
- Order strengthening.

Particle looping stands as a category and as the only mechanism under his category.

The discovery of this phenomenon in aluminum alloys dates back to the year of 1906. A man by the name of Alfred Wilm discovered an increase of hardness when an Al-Cu-Mg alloy was rapidly cooled in cold water from an initial temperature of 550ºC (a process named as quenching). What was also noticed by Alfred Wilm was that as the alloy was left at room temperature it continued to increase in hardness. Basically the alloy hardened with age (over time). This phenomenon is known as age hardening. In the year of 1919, Mercia, Waltenberg and Scott published their observations of the study of an Al-Cu alloy. What they proposed and demonstrated was that as the temperature decreased the solid solubility of the alloy element (copper) also decreased, which led to the explanation of precipitation of copper atoms from a supersaturated solid solution. In 1932 Mercia in a published paper, named these small clusters as "knots". The evidence of these "knots" where demonstrated by the work of Guiner and also by the work of Preston. The interpretation of diffuse x-ray scattering analysis allowed the identification of clusters of atoms in very small zones. Nowadays, these zones are known as Guiner-Preston zones (GP zones) [Jacobs1999].

1.3 Aluminum and aluminum alloys

Aluminum is a material of many characteristics that make it suitable for a variety of applications in numerous industries. Aluminum is characterized as being [Barralis2010]:

- Strong:
  As an example, it is possible to mention that an entire vehicle body can be constituted of aluminum;
- Durable:
  It is a good resistant material not just against corrosion but also against fatigue;
- Conductive:
  Aluminum is an efficient thermal and electrical conducting material;
- Low density:
  Aluminum is a very light metal. The application in vehicles allows weight reduction and as a consequence, a reduction of energy consumption;
- Reflectivity:
  It is a good reflector of visible light as well as heat. These qualities allow the application of aluminum in rescue blankets and light fittings;
- Nonmagnetic:
  As a nonmagnetic material, it is useful for high-voltage applications, as well as for electronics;
- Nontoxic:
  This characteristic enables the use of aluminum in cooking utensils and let us not forget the use of the aluminum foil wrapping in direct contact with fresh food;
- Abundant:
  Various studies mention that aluminum is the most abundant metal on planet Earth;
- Recyclable:
  Aluminum is a 100 percent recyclable material and by which there is no degradation of its material properties and qualities. The re-melting process of aluminum is a process that requires little amounts of energy;
- Workable:
  The excellent workability of aluminum is apparent from the facility with which aluminum can be extruded, rolled, formed and warped.

These are the characteristics that give aluminum its extreme versatility. In the majority of industrial applications, two or more of these characteristics come

prominently into play. For example, it is possible to mention that for transportation equipment (such as aircrafts, cars, trucks and railway rolling stocks) the most important it to combine lightweight and strength. As another example, for chemical and petroleum industry equipment, high resistance to corrosion and high thermal conductivity are of the most importance. Table 1 resumes the aluminum main material properties.

Table 1 – Aluminum properties [Barralis2010].

Aluminum Properties	
Melting Point	660 °C
Crystalline System Structure	FCC (a = 0.4041 nm at 20 °C)
Density at 20 °C	Al=2.7 g/cm^3
Expansion coefficient:	
α=	23.8.10^{-6} K^{-1} (from 20°C to 100°C)
α=	25.4.10^{-6} K^{-1} (from 20°C to 300°C)
α=	28.7.10^{-6} K^{-1} (from 20°C to 600°C)
Mechanical strength of pure Al (99.99 %)	6 Kg/mm^2
Mechanical strength of commercial Al	9-14 Kg/mm^2
Ductility	HB = 17-20
Modulus of elasticity of Al	7000 kg/mm^2
Main impurities	iron, silicon, copper
Main alloy elements	Cu, Mg, Si, Zn, Ni, Ti, Cr, Co, Pb, Sn, besides others
Atomic number	Z = 13
Atomic mass	M = 26.97
Thermal mass capacity at 20 °C	c=950 J.kg^{-1}.k^{-1}
Thermal conductivity at 20 °C	λ=217.6 W.m^{-1}.K^{-1}
Electric resistivity at 20 °C	ρ=2.63 μΩ.cm
The aluminum electric conductivity is equal to 65 % of copper	

The alloy elements (components) that do improve aluminum properties are [EAA2002]:

- Copper:

  This element is applied to increase strength and hardness. Also applied for heat treatability;

- Magnesium:

  Applied with regard to increase hardness and corrosion resistance;

- Manganese:

  Applied to increase strength;

- Silicon:

  Applied to lower melting point and increase castability;

- Silicon and magnesium:

Applied for heat treatability;

- Zinc:

  Applied to increase strength and hardness;

- Zinc and Magnesium:

  Applied to increase strength. Also applied for heat treatability;

- Bismuth:

  Applied to increase machinability;

- Boron:

  Applied to increase electrical conductivity;

- Nickel:

  Applied to increase strength at high temperatures;

- Titanium:

  Applied to increase strength and ductility.

Other elements that are also used as alloy elements are: chromium, vanadium, zirconium and scandium [EAA2002].

1.4 Scandium

Looking up the periodic table, scandium is the element number 21. It is an element known as a light metal, characterized by having a density value of $\rho \approx 3 gcm^{-3}$ and a melting point of 1541ºC. It is labeled in materials science literature as a "*rare earth metal*" (R.E.M) [Naukin1965].

Besides these characteristics, scandium is a soft, silver-white looking material. It oxidizes quite easily, tarnishing to pink or yellow. When in contact with water, a chemical reaction occurs in which hydrogen is released. It reacts chemically very well with aluminum, magnesium and zirconium. Table 2 resumes the scandium main material properties.

1.5 Aluminum-scandium alloy

According to the literature, the first steps in the development of Aluminum-Scandium alloys began in the Soviet Union, encouraged by military demands [Ahmad2003].

In the most recent years, scandium has been introduced into several commercial aluminum alloys (for example: Al5052 and Al7075) and by this, bringing great improvement in their mechanical and physical characteristics [Ahmad2003].

Table 2 – Scandium main properties [Webelements2012].

Scandium Properties	
Melting Point	1541 °C
Boiling Point	2836 °C
Density	2.985 g/cm^3
Heat Fusion	14.1 KJmol^{-1}
Heat Vaporization	332.7 KJmol^{-1}
Specific Heat Capacity	25.52 JmolK
Young´s modulus	74.4 GPa
Shear modulus	29.1 GPa
Bulk modulus	56.6 GPa
Poisson ratio	0.279
Brinell hardness	750 MPa
Atomic weight	44.95591
Thermal conductivity	16 W(mk)
Thermal expansion	0.0000102 K^{-1}
Electric resistivity	ρ=5.5x10^{-7} Ω

The addition of scandium in aluminum alloys offers a significant number of advantages in comparison with other high strength aluminum alloys. Some of these benefits are:

- Inhibition of recrystallization:

  This capacity of inhibition of recrystallization is far more effective than the transition metals zirconium, chromium, manganese, vanadium and titanium.

- Strengthening increase:

  Studies mention that the addition of scandium provides the highest increment of strengthening per atom in comparison to other aluminum alloy elements.

- Refinement of grain size:

  As an example, the addition of scandium makes it possible to obtain continuously cast billets with non-dendritic structures.

- Reduction and elimination of hot cracking in welding:

  As an example, the hot-crack sensitive aluminum alloy 2618 when added scandium reduces its crack susceptibility [Ahmad2003].

The next figures show a comparative analysis between conventional aluminum and aluminum with the presence of scandium. Figure 2 refers to the yield strength comparative between the two materials and as for Figure 3, it refers to the fatigue life comparative. Clearly both figures demonstrate increasing gains by the addition of scandium to the aluminum.

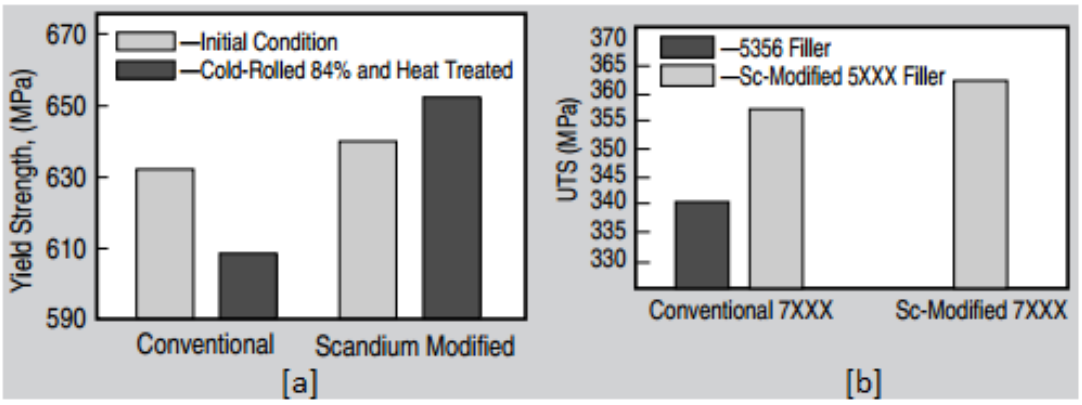


Figure 2 – Comparison between conventional and scandium modified aluminum. [a] compares the yield strength and [b] compares tensile strength [Ahmad2003].

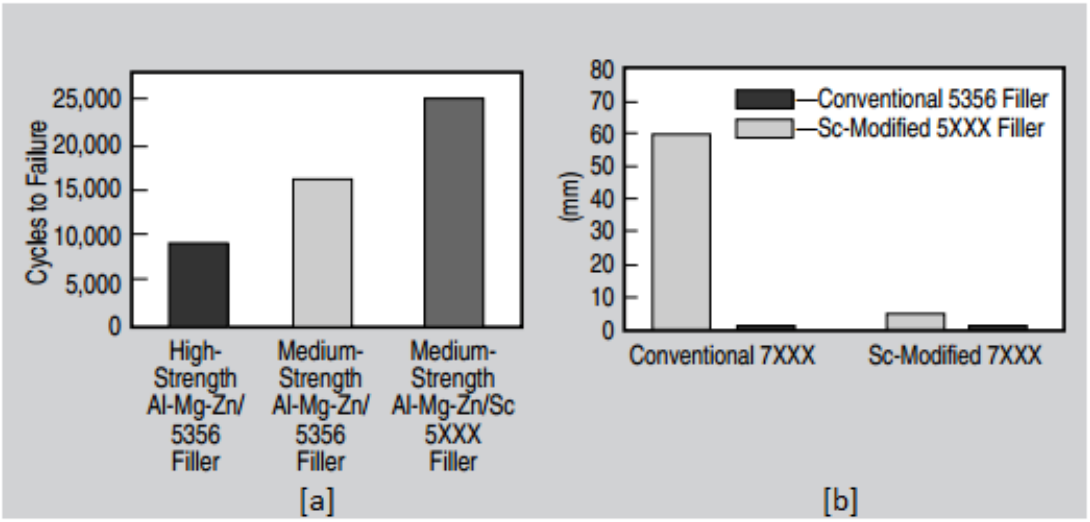


Figure 3 – Comparison between conventional and scandium modified aluminum. [a] compares the fatigue life and [b] compares the Houldcroft test [Ahmad2003].

Figure 4 shows the Al-Sc binary phase diagram. Looking at this figure, aluminum scandium is slightly hypereutectic and therefore, very small amounts of Al_3Sc precipitates could form prior to the solidification of the aluminum phase [Ahmad2003].

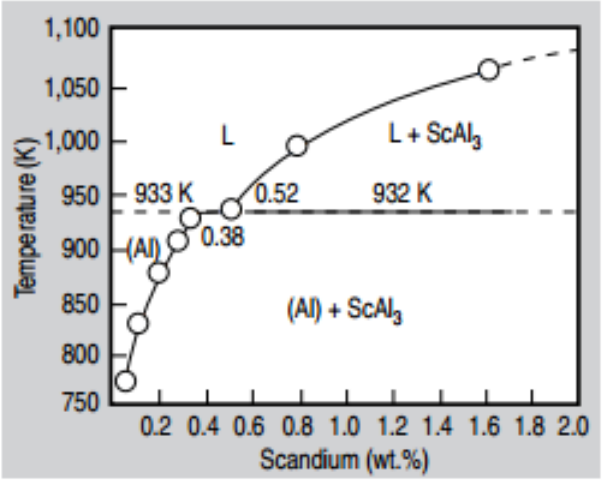


Figure 4 – Binary phase diagram of aluminum and scandium [Ahmad2003].

The aluminum scandium alloys have potential in aerospace applications, including bulk heads, heat shields, running gear, fuel systems as well as exhaust systems. The promising automotive applications that can be highlighted are: wheels, bumpers, frames, pistons, air bag canisters. Another area of applicability is the sports industry: softball and baseball bats and also bicycle frames are just same examples [Ahmad2003].

The thesis is structured in six chapters plus several appendices:

- Chapter 1 describes in a profound manner, a introduction to the knowledge held in this thesis;
- Chapter 2 references the related work published in the recent years covering the application of the kinetic Monte Carlo method in the simulation of precipitation phenomenon in several alloys besides the aluminum scandium alloy;
- Chapter 3 covers the theoretical background needed to implement the precipitation simulation;
- Chapter 4 details the methodology used to simulate the precipitation of Al_3Sc structures used in this work;
- Chapter 5 covers the analysis and discussion of the Monte Carlo simulation results as well as a comparison of these results with the classical nucleation theory;
- Chapter 6 presents the main conclusions of this thesis and future work;
- Finally, the appendices will feature besides the simulation code, simulation images, simulation architecture, simulation graphics and simulation reports.

2. Related Work

As computation extends its capacities increasingly, so has the scientific field of nucleation and precipitation modeling. The process of modeling nucleation and precipitation has been achieved at different scales, each one having its own advantages and disadvantages:

- Looking at the atomic scale, the atomistic kinetic Monte Carlo method is widely used. With the adequate computation resources, it is possible to simulate both the initial and relatively late stages of the precipitation process, and that includes coarsening.
- Looking at the mesoscopic level, the most common techniques are: cluster dynamics and classical modeling methods.
- Looking at the macroscopic scale it is known the application of the Johnson-Mehl-Avrami-Kolmogorov (JMAK) model or simply analyzing precipitation with conventional transmission and high-resolution electron microscopies (HREM) [Deschamps2010].

The physical parameters that describe the precipitation processes are very important, but also quite difficult to model.

2.1 Atomistic Monte Carlo simulations

It has increased the number of publications and studies related with the subject of modeling the precipitation kinetics at the atomistic level. It is possible to encounter studies based on the method of cluster dynamics and other studies based on the Monte Carlo method.

The main materials subjected to such studies are alloy materials such as Fe-Cu, Fe-P-C, Fe-Cu-Ni-Si, Al-Cu. Aluminum alloys have also their share of studies by which we would like to outline and focus on the Al-Sc alloy.

Peter Binkele and Siegfried Schmauder have published studies about precipitation in binary systems using atomistic Monte Carlo simulations, at a temperature of 773K over $500x10^9$ Monte Carlo steps (MCS). The Monte Carlo method applied was a rejection-free residence time algorithm, which allowed an improvement in time calculation in comparison with a Metropolis algorithm. They were able to give information concerning the precipitation average radius dimension over the "aging time". As for conclusions: their work states that the beginning mean radius of precipitation grows proportional to $t^{0.180}$ and in a much advance state the growth approaches the classical value of $1/3$ [Binkele2003].

In another study published by Siegfried Schmauder and Peter Binkele on the precipitation of Cu-precipitates in steels it was possible to conclude that with longer simulation times, a significant decrease of the number of small precipitates and therefore an increase of the averaged precipitates radius. The rejection-free residence time algorithm was also applied, allowing an improvement in calculation time. Up to $75x10^{10}$ Monte Carlo steps (MCS) were simulated on a fixed BCC crystal lattice [Schmauder2002].

P. Binkele, P. Kizler and S. Schmauder also run atomistic Monte Carlo simulations of the diffusion of phosphorous (P) and carbon (C) to grain boundaries in BCC iron. This study simulated both diffusion mechanisms: vacancy and interstitial. The vacancy mechanism is performed using a rejection-free residence time algorithm. The simulation of the interstitial mechanism was performed with a combination of a residence time algorithm and a Metropolis algorithm. Several simulation temperatures were used (773 K, 873 K, 973 K, 1073 K) and simulated up to a $9.0x10^{11}$ Monte Carlo steps [Binkele2004].

Emmanuel Clouet *et al.* have published studies of atomistic Monte Carlo simulations not just based on binary systems but also on ternary systems. The binary Al-Sc alloy simulations produced results (with $210x10^9$ MCS) for the precipitates size and radius, and a comparison with the classical nucleation theory. The simulation applied a residence time algorithm using an FCC rigid lattice [Clouet2004a].

In the work by David Bombac and Goran Kuglar it was simulated a Fe-Cu alloy using a Monte Carlo method. The simulation was based on a residence time algorithm and used a BCC rigid lattice structure and applied a temperature of 873K. The outputs of their study are the number of precipitates and their dimension (diameter) [Bombac2010].

Emmanuel Clouet *et al.* have published several studies regarding the alloy under study in the present thesis (Al-Sc), not just in binary situation but also in a ternary situation (Al-Zr-Sc).

In another publication by Clouet *et al.*, about the precipitation of Zr and Sc in an aluminum alloy, the simulation applied a Monte Carlo residence time algorithm to analyze the homogeneous and heterogeneous precipitation. The study concludes that kinetic Monte Carlo allowed the improvement of classical descriptions for the different precipitation stages [Clouet2006].

Emmanuel Clouet and Frédéric Soisson have published a summary of recent applications of the atomistic diffusion model and of the kinetic Monte Carlo method. The summary covers homogeneous and heterogeneous precipitation caused by thermal aging as well as phase transformation caused under irradiation. To conclude this publication the authors mention that atomistic kinetic Monte Carlo simulations provide a convenient way to simulate and model precipitation kinetics in alloys [Clouet2010].

The work by Lae *et al.* documents a study in which cluster dynamics simulation is applied to an aluminum scandium alloy and also an aluminum zirconium alloy and the achieved results are compared with Monte Carlo simulation results. The study found a good agreement between both simulations results [Lae2004].

Another author of several studies related with atomistic Monte Carlo simulations is Frédéric Soisson and coworkers and also Georges Martins. Fréderic Soisson and Chu-Chun Fu elaborated a study in which the first stages of the coherent precipitation of copper in α-Fe were simulated using the Monte Carlo method based on a residence time algorithm and using a rigid lattice. The first stage of this publication describes the study of the thermodynamic and diffusion properties of Fe-Cu alloys using *ab initio* calculations. The second stage of the publication refers to the kinetic Monte Carlo simulation method, as to follow the precipitation kinetics [Soisson2007].

Georges Martin and Frédéric Soisson have summarized some of the models (transition probabilities models) most commonly used in Monte Carlo simulations. The models summarized, concern only the vacancy diffusion mechanisms [Martin2005].

E. Vicent *et al.* provided a critical review of the different kinetic Monte Carlo models used to simulate Cu precipitation. Their work covers the application of *ab initio* calculations, cut bond models and Monte Carlo method. They concluded that the mixing energy is the most important parameter for the kinetics of precipitation since this parameter not just determines the solubility limit of the system, but also the driving force of the precipitation [Vincent2008].

C. Hin elaborated a study of the precipitation of grain boundary precipitation of Ni_3Al in a nickel alloy. This work was elaborated using a kinetic Monte Carlo method, namely the residence time algorithm and the use of a rigid lattice [Hin2009].

Monte Carlo simulations have also been used on the study of other phenomena. Grain growth, abnormal grain growth, thin film deposition and growth, sintering for nuclear fuel aging, bubble formation in nuclear fuels are just some of those phenomena [Plimpton2009].

Examples of grain growth and abnormal grain growth studies are the works of Elizabeth Holm *et al*. [Holm2010]. Dealing with this phenomenon are also the researchers A. D. Rollett, M. P. Anderson, D. J. Srolovitz.

Literature mentions that P. Andersson and his co-workers elaborated the first study of grain growth, applying the Monte Carlo Potts model [Escke2011].

2.2 Experimental approaches

The work developed by Marquis *et al.* describes the process of dating the morphology of Al_3Sc precipitates experimentally by using conventional transmission and high-resolution electron microscopy (HREM). In this study and for the first time it was possible to determine the exact morphology of Al_3Sc precipitates in the following alloys: Al-0.1 wt% Sc and Al-0.3 wt% Sc. The study determined the equilibrium shape: the Great Rhombicuboctahedron with its 26 facets. The obtained conclusions are [Marquis2001]:

- The effect of scandium content on nucleation and on the precipitates morphology was observed through the morphological evolution of Al_3Sc precipitates as a function of annealing time and temperature;
- The number density of precipitates increases with increasing scandium concentration at constant annealing temperature, that is with increasing supersaturation;
- The activation energy for diffusion of Sc in the Al matrix obtained from the coarsening experiments is of 164 ± 9 KJ/mol, which agrees favorably with the activation energy obtained from tracer diffusion measurements of Sc in Al (173 KJ/mol), and the value calculated from first principles (154 KJ/mol).

The study developed by K. E. Knipling and co-workers investigated the precipitation strengthening of the binary alloys Al-0.1Sc and Al-0.1Zr and the ternary alloy Al-0.1Sc-0.1Zr. The study goes on to detail the composition, radii, volume fraction, and number densities of Al_3Sc and Al_3Zr precipitates. These measurements were directly obtained by using atom-probe tomography. They concluded that the Al_3Sc precipitation begins at temperatures within the range of 200ºC to 250ºC [Knipling2010].

3. Theoretical Background for Al_3Sc Precipitation Simulation

3.1 Face-centered cubic system

Aluminum is a face-centered cubic system (FCC). A face-centered cubic system is defined as a crystalline structure, in other words, an arranged pattern of atoms, molecules which repeats its structure in the three dimensions. It is defined as a space lattice, the distribution of atoms in the three dimensions and in which every lattice site (atom, point) has an identical surrounding. The space lattice has a geometrical structure that is completely defined by the designated lattices constant, which are defined usually by the letters *a*, *b*, *c* and the designated inter axial angles α, β, γ.

The face-centered cubic system is a crystalline pattern in which besides the vertex atoms, there is a center atom located on each face of the cubic structure as Figure 5 demonstrates. In this structure, the distance along the unit cell edge is called the lattice parameter (represented by the letter *a*). This parameter is identical in all 3 axes.

A face centered cubic structure has 8 vertex atoms shared by 8 other cells and 6 face center atoms and each one shared by two cells. The number of atoms per cell is given by Equation 1.

$$8 \times \frac{1}{8} + 6 \times \frac{1}{2} = 4\ atoms\ per\ FCC\ cell \qquad (\ \text{Equation 1}\ )$$

In the face center cubic structure unit cell, the atoms contact each other across the cubic face diagonal. As so, Equation 2 relates the length of the cube side "a" with the atomic radius "R" as follows:

$$\sqrt{2}a = 4R \Leftrightarrow a = \frac{4R}{\sqrt{2}} \qquad (\ \text{Equation 2}\ )$$

Figure 6 shows the first and second nearest neighbors of a face-centered cubic lattice site. The first nearest neighborhood is composed of 12 lattice sites and the second nearest neighborhood is composed of 6 lattice sites. If we move on to the third nearest neighborhood, it is composed of 24 lattice sites.

Table 3 summarizes the main characteristics of a face centered cubic structure such as aluminum.

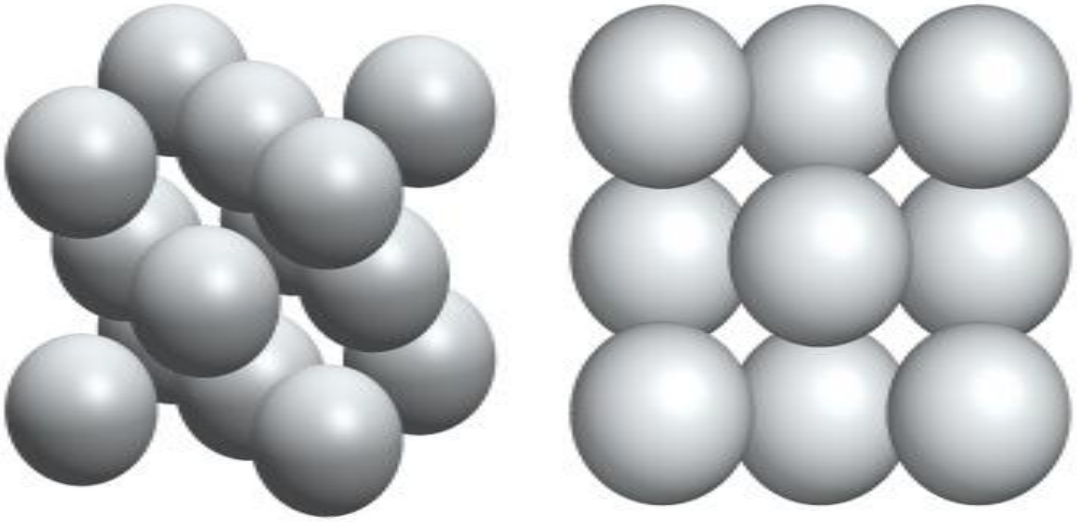


Figure 5 – FCC lattice.

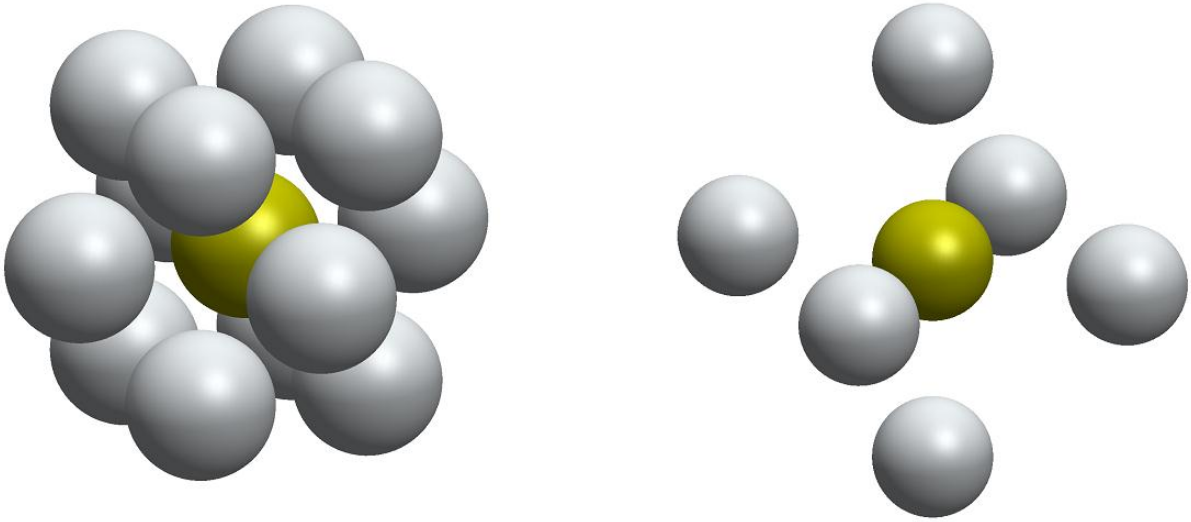


Figure 6 – First (a) and second (b) nearest neighbors.

Table 3 – FCC system characteristics [Barralis2010].

Characteristic	Value
Atoms per cell	4
Number of first nearest neighbors	12
Nearest neighbor distance	$(a\sqrt{2})/2$
Number of second nearest neighbors	6
Second nearest neighbor distance	a

3.2 Aluminum-scandium

We can ask what are the effects of scandium in aluminum alloys. In the literature it is affirmed that the addition of scandium to aluminum alloys has three consequences of upper importance:

- The grain refinement during casting and welding processes;
- The precipitation hardening phenomenon;
- The grain structure control.

What unites these three consequences? They are all related to the formation of particles of Al_3Sc, the phase that is in equilibrium with aluminum.

The Al_3Sc structure can be described as an ordered face-center cubic (FCC) structure. In terms of crystallographic terminology the Al_3Sc is a simple cubic lattice with four atoms: one Sc atom (at the cubes vertexes) and three Al atoms (at the faces center) that occupy each lattice point. This atomic arrangement is designated as $L1_2$ arrangement.

Figure 7 was obtained from a TEAM (transmission electron aberration-corrected microscope) microscope, in the Lawrence Berkley National Laboratory. It shows an $L1_2$ structure in which the center face atoms are aluminum atoms and the vertex atoms are scandium atoms.

3.3 Kinetics of precipitation

The heat treatment processes, also known as precipitation hardening or age hardening, are applied on alloys in order to enhance the strength and hardness of the alloy. This entails the formation of dispersed uniform particles of a second phase within the original phase matrix. As mentioned earlier, the so called precipitate particles act as obstacles to dislocation movements. In order for a an alloy system to be strengthen by the formation of precipitates, it must be obvious from its phase diagram that starting in the solid solution zone, as the temperature decreases, the solid solubility decreases.

The precipitation hardening process involves three main steps, known as solution treatment, quenching, and aging. Solution treatment, also known as solutionizing, is the step where the alloy is heated above its solvus temperature until a homogenous solid solution (α) is produced. Quenching is the step where the solid (α) is rapidly cooled and forms a supersaturated solid solution that is not a structure in equilibrium. Aging is the step in which the supersaturated structure is heated below its solvus temperature in order to produce precipitates. As the supersaturated structure is unstable, the solute atoms diffuse towards nucleation sites, increasing the size of the precipitates [Jacobs1999].

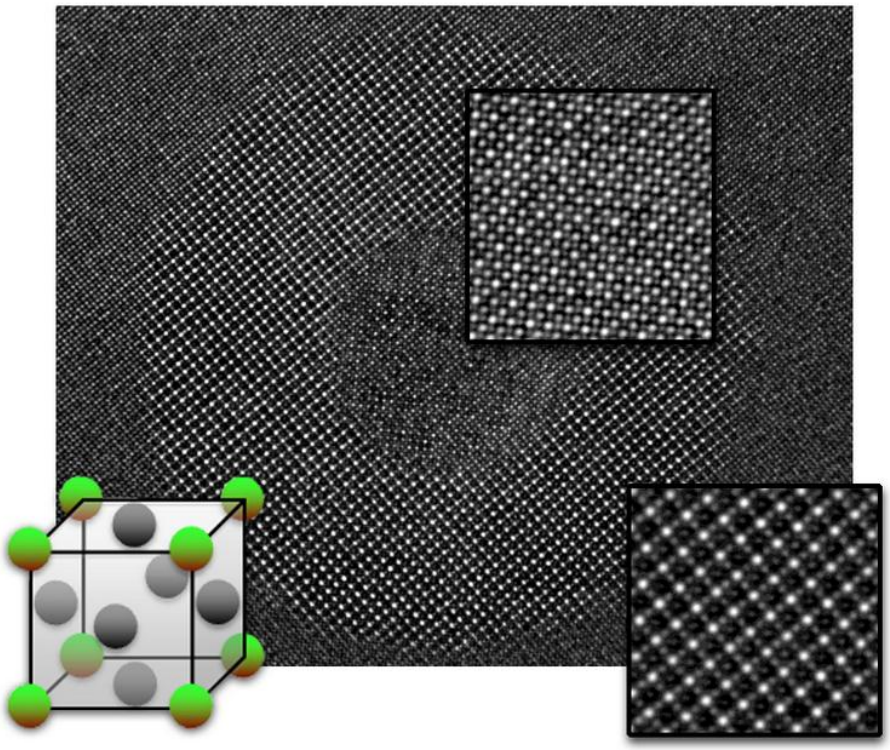


Figure 7 – Al_3Sc precipitate system structure.

The literature mentions that the Al_3Sc phase can be obtained by four different methods (see Figure 8):

- In a situation in which occurs solidification of hypereutectic alloys (Sc > approx. 0.6 wt - %). In this situation Al_3Sc is the first phase to be formed;
- In a situation in which occurs solidification of hypo– and hypereutectic alloys. In these situations, the last phase to be formed is: eutectic Al + Al_3Sc;
- In a situation of supersaturated solid solution, Al_3Sc can precipitate discontinuously;
- In a situation of supersaturated solid solution Al_3Sc can precipitate continuously (nucleation and growth) **[**Røyset2005**]**.

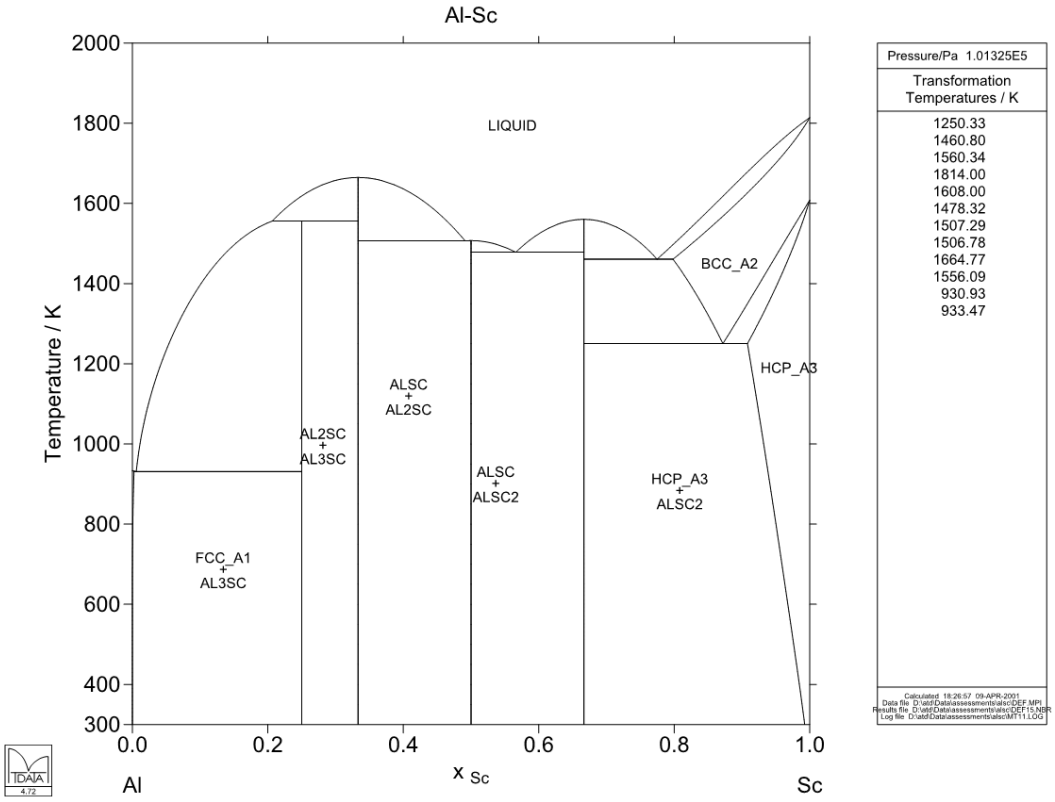


Figure 8 - Al–Sc binary phase diagram [Röyset].

Table 4 summarizes the crystal structure data obtained from an Al-Sc binary phase diagram.

Table 4 – Aluminum Scandium crystal structure data [Okamoto1991].

Phase	Composition at % Sc	Pearson Symbol	Space Group	Structure Designation	Prototype
(Al)	0 to ≈0.2	cF4	$Fm\bar{3}m$	$A1$	Cu
Al_3Sc	25	cP4	$Pm\bar{3}m$	$L1_2$	$AuCu_3$
Al_2Sc	33.3	cF24	$Fd\bar{3}m$	$C15$	Cu_2Mg
AlSc	50	cP2	$Pm\bar{3}m$	$B2$	$CsCl$
$AlSc_2$	66.7	hP2	$P6_3/mmc$	$B8_2$	Ni_2In
βSc	≈92 to 100	cI2	$Im\bar{3}m$	$A2$	W
αSc	96 to 100	hP2	$P6_3/mmc$	$A3$	Mg

The continuous precipitation of Al_3Sc is identified by being a bulk decomposition process of a supersaturated solid solution of scandium in aluminum and can be characterized by three stages: beginning with the nucleation stage, followed by the growth stage and, in the end, a coarsening stage [Røyset2005]. This continuous precipitation reaction occurs largely by homogeneous nucleation and diffusion controlled growth. Besides the most frequently reported nucleation that occurs

homogeneously throughout the aluminum matrix, there are studies in which the nucleation occurs on dislocation and grain boundaries. Such nucleation is designated as heterogeneous.

The shape of continuously precipitated Al_3Sc particles is most frequently reported to be spherical.

Literature mentions that the kinetics of Al_3Sc precipitations is dependent on two main factors: the heat treatment temperature and also on the scandium content in the alloy.

Looking at the atomic scale, the coherent precipitation (in which, it is neglected the associated coherency strain) is described as thermal activated atomic jumps on a lattice structure. In other words, solute atoms change lattice sites by jumping into a nearest neighbor vacant site. What results from this, is the formation and evaporation of solute clusters. The clusters capacity to grow or to shrink will depend on the content of solute and the solute jump frequency.

Literature mentions that Scandium, in a supersaturated solution, may precipitate as finely dispersed Al_3Sc particles in the temperature range of 250°C – 350°C, as shown in Figure 9 [Röyset].

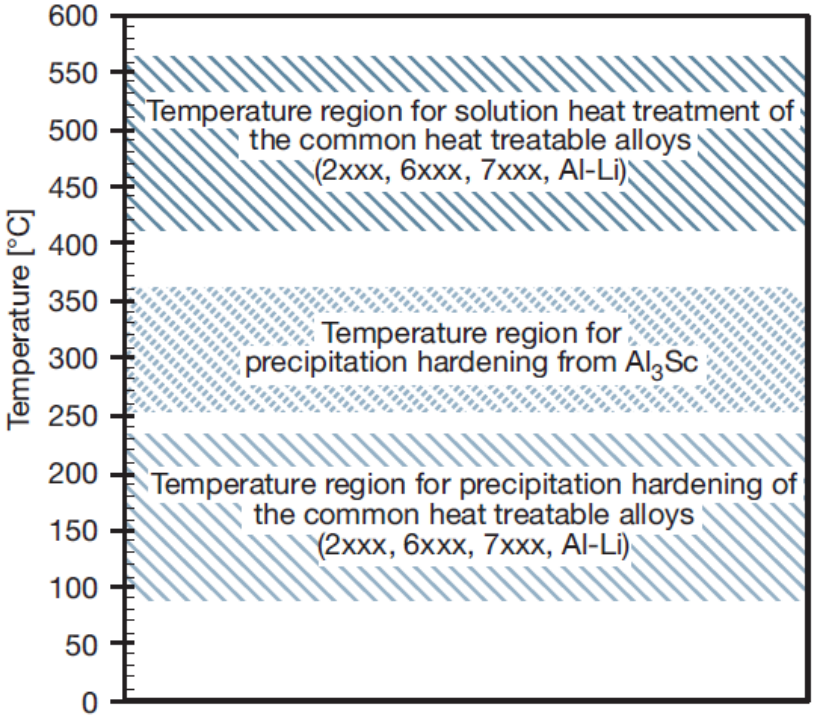


Figure 9 – Temperature regions for precipitation phenomenon [Röyset].

3.4 Vacancies and precipitation

It is well known that the precipitation is controlled by the rate of atomic migration in an alloy material and that the temperature has an enormous influence on this process [Smallman1999]. In fact, vacancies play a presence in every crystalline material at finite temperature and this presence plays a significant role in the mechanical behavior of the material. The relation between equilibrium concentration of vacancies and temperature is exponential. As temperature rises the vacancy concentration increases and as so, the rate of atomic migration also increases [Marceau2008].

Atomic migration occurs via a vacancy mechanism. The presence of vacancy point defects is essential for the vacancy mechanism. As a simple description of this process, an atom that is associated to a lattice site, jumps (also applied the word *hop*) into a lattice site that is atom vacant and that is in its closest neighborhood.

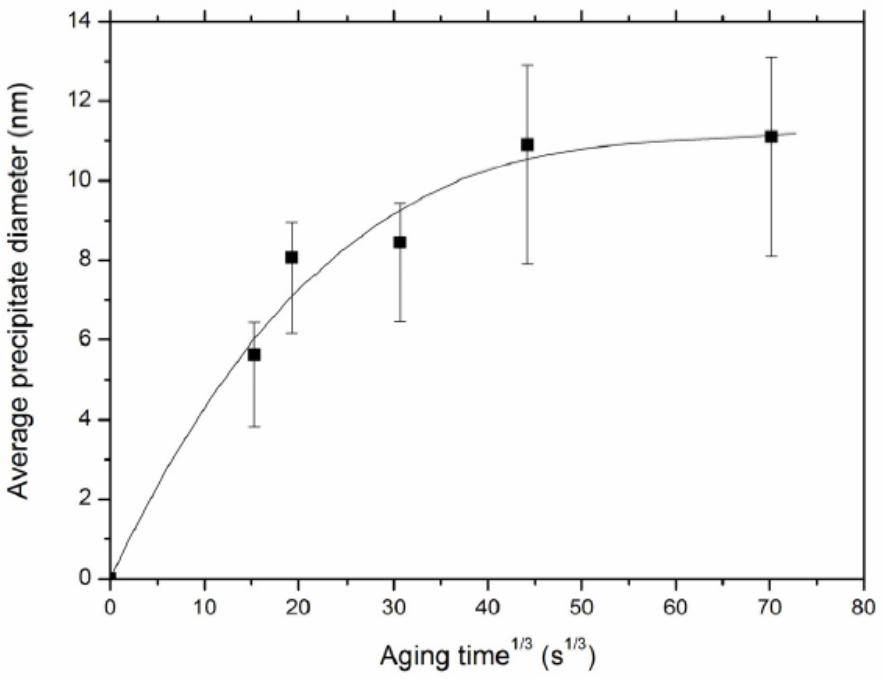


Figure 10 – Precipitation size measurements obtained by image processing techniques. This exercise underwent a temperature of 350°C.

In Figure 10 it is plotted the particle size as a function of the aging time. Looking at the figure it is possible to conclude that the precipitation growth doesn't agree with the $t^{\frac{1}{3}}$ relation anticipated by the Lifshitz-Slyozov-Wagner (LSW) theory.

3.5 Diffusion

The transport of atoms through a lattice structure may occur in various ways. Such mechanisms are named: interstitial diffusion and vacancy diffusion. Interstitial diffusion occurs by the migration of small atoms through the interstitial positions of the lattice structure. Examples of such small atoms are: carbon, nitrogen or hydrogen. Literature also mentions that in pure metals, self-diffusion occurs when there is no mass transport occurrence, but in which atoms migrate in a random manner throughout the crystal structure. Inter-diffusion in the case of alloys occurs by mass transport with the purpose of minimizing compositional differences.

Vacancy diffusion is known to be as the most energetically favorable process and in which occurs the interchange of positions (lattice sites) by an atom and a neighbor vacancy. In the vacancy diffusion mechanism, the jumping (hopping) performed by an atom to a neighbor vacant site depends on the following probabilities [Smallman1999]:

- The probability that the site is vacant;
- The probability that it has the required activation energy.

The diffusion coefficient (*D*) for self-diffusion is given by the Equation 3.

$$D = \frac{1}{6}a^2 f v \times exp\left[\left(S_f + S_m\right)/k\right] \times exp\left[-E_f/kT\right] \times exp\left[-E_m/kT\right] = D_o \times exp\left[-\left(E_f + E_m\right)/kT\right] \qquad \text{(Equation 3)}$$

In this equation, f is a correlation factor, ν is the attempt frequency, E_f is the energy of formation of a vacancy, E_m is the energy of migration, and the designated activation energy is the sum of the two mentioned energies (E_f and E_m).

Temperature and chemical composition are the two fundamental factors that affect the diffusion coefficient *D*. The diffusion rate increases with the increase of temperature since this affects the activation energy term of Equation 3. Diffusion also increases with the increase of lattice irregularities.

The diffusion in alloys is a process of transfer of atoms of different alloy components, resulting in changing chemical composition of some of the alloy regions. The self-diffusion is a process of transfer of atoms of a certain element among themselves. The diffusion mechanism is determined by the energy barrier that an atom must overcome in order to change its location. Some diffusion (self-diffusion) mechanisms are represented in Figure 11.

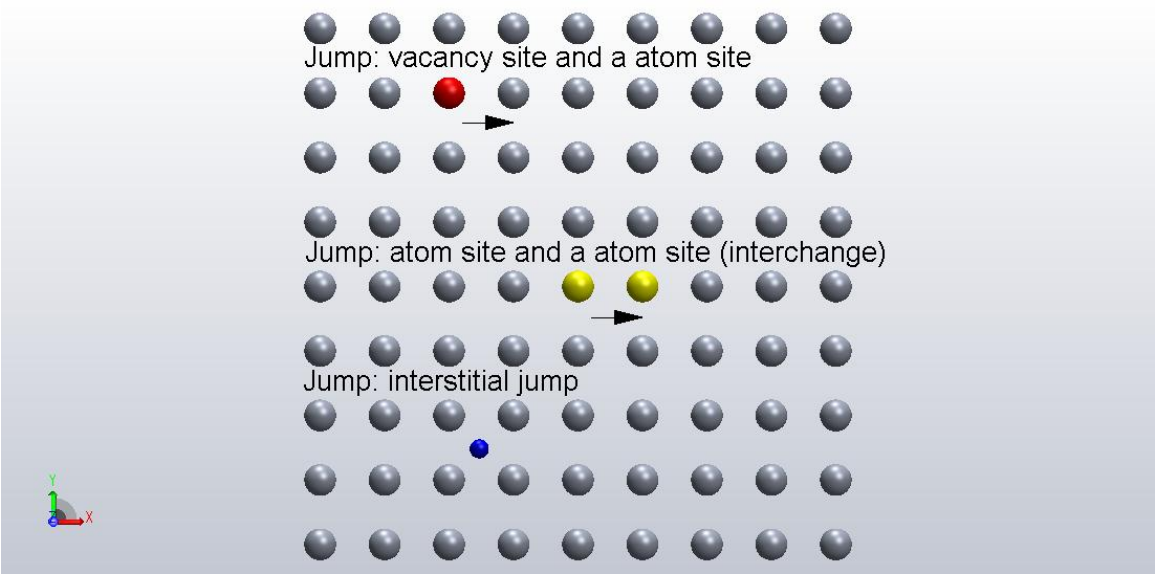


Figure 11 – Types of diffusion mechanisms in alloys.

Metallic atoms diffuse mostly by the vacancy mechanism and elements with small atom sizes (Hydrogen, Nitrogen and Carbon) diffuse by the interstitial mechanism.

Besides diffusion within the crystal, also grain boundary (surface) diffusion may occur. As the grain boundaries regions are saturated with crystal lattice imperfections, energy barrier (activation energy) here are relatively small, therefore the diffusion rate along these regions is much higher, than the volume diffusion rate.

The classical law, describing the diffusion process for a single spatial dimension, is designated as Fick's Laws and it is represented by Equation 4:

$$J_x = -D\frac{\partial c}{\partial x} \qquad \text{(Equation 4)}$$

This equation relates the diffusive flux (J_x) to the concentration of atoms (c). It postulates that the flux goes from regions of high concentration to regions of low concentration, with a magnitude that is proportional to the concentration gradient.

3.6 Solubility

In a general form, solubility is understood as the maximum amount of solute that is dissolved in a solvent in equilibrium (balanced stage between reactants and products). In the situation of maximum solubility, there is no possibility for more of the solute to be dissolved in the solvent. From this situation forward, the solution is identified as a saturated solution.

3.7 Metastability

The concept of less stability then a most stable state is designated as metastability.

A metastable phase is characterized by being a thermodynamically stable phase at room temperature. Under external action, the phase may suffer state changes as it moves on to a more stable phase. To pass from one state to the other, it is mandatory to go through the saddle point.

3.8 Nucleation

The formation of precipitates requires necessarily the nucleation of the new phase in localized regions of the system.

This process will proceed through four main stages:

- Stage I: designated as the incubation period. This stage is characterized by a metastable matrix phase and in which no stable particles of a new phase have been formed. Although, during this stage, small clusters (named also as embryos) continuously form and deform in the metastable matrix. As time follows, the clusters increase in dimension, gaining a stable increase of dimension and as so, reducing the probability to decompose back to the metastable matrix. At this point, we have stable nuclei of the new phase, which continue to grow and permanently do not decompose. Nucleation is taking place.
- Stage II: identified as the quasi-steady-state nucleation, is the period during which stable nuclei are being created in a constant rate.
- Stage III: this moment enhances a reduction of the nucleation rate, and as so, the number of stable nuclei maintains constant.
- Stage IV: in this last stage there is no formation of new stable nuclei.

The classical nucleation theory focuses on the stages I and II.

The classical nucleation theory (CNT) enables the prediction of the nucleation rate. In several published works ([Clouet2004a], [Clouet2005a]), it is mentioned that comparisons between the Monte Carlo simulations with the classical nucleation theory show very similar predictions. We also carried out this comparison on our work.

Homogeneous nucleation

The homogeneous nucleation occurs in locations of the system in which it is absent of crystal defects or impurity particles.

Heterogeneous nucleation

On the other side, heterogeneous nucleation takes place in defected localizations.

Based on the classical nucleation theory, several steps are needed to obtain a measurement of the kinetics of nucleation. So, to calculate the steady-state nucleation rate, which is the number of supercritical nuclei which form per unit time in a unit volume, it is necessary to calculate the equilibrium concentration, the impurity diffusion, the average interface free energy, the nucleation free energy, the nucleation barrier, the condensation rate and the Zeldovitch factor. The sequence of equations that follows (Equation 5 until Equation 18) is used in the work by Emmanuel Clouet [Clouet2004a] as a possible way to obtain the mentioned parameters specifically for the aluminum-scandium alloy.

Equilibrium Concentration

$$X_{Sc}^{eq} = exp[(-0.701 + 230 \times 10^{-6}T)\ eV/kT] \qquad (\text{Equation 5})$$

Impurity Diffusion

$$D_{Sc} = 5.31 \times 10^{-4} exp(-1,79\ eV/kT) \qquad (\text{Equation 6})$$

Interface free energy

The interface free energy between the $L1_2$ precipitate and the aluminum solid solution is obtained by using the sequence of equations from Equation 7 to Equation 12.

$$\frac{\sqrt{2}}{2}\sigma_{100} < \sigma_{110} < \sqrt{2}\sigma_{100} \qquad (\text{Equation 7})$$

$$\frac{\sqrt{6}}{3}\sigma_{110} < \sigma_{111} < 2\frac{\sqrt{6}}{3}\sigma_{110} - \frac{\sqrt{3}}{3}\sigma_{100} \qquad (\text{Equation 8})$$

$$\Gamma_{100} = 4\left(\sigma_{100} - \sqrt{2}\sigma_{110}\right)^2 - 2\left(\sigma_{100} - 2\sqrt{2}\sigma_{110} + \sqrt{3}\sigma_{111}\right)^2 \qquad (\text{Equation 9})$$

$$\Gamma_{110} = 2\sqrt{2}\left(-2\sigma_{100} + \sqrt{2}\sigma_{110}\right)\left(\sqrt{2}\sigma_{110} - \sqrt{3}\sigma_{111}\right) \qquad (\text{Equation 10})$$

$$\Gamma_{111} = 3\frac{\sqrt{3}}{2}\left(-\sigma_{100}^2 - 2\sigma_{110}^2 + \sigma_{111}^2\right) + \frac{3}{2}\sigma_{100}\left(4\sqrt{6}\sigma_{110} - 6\sigma_{111}\right) \quad \text{(Equation 11)}$$

$$\bar{\sigma} = \sqrt[3]{\frac{1}{4\pi}\left(6\sigma_{100}\Gamma_{100} + 12\sigma_{110}\Gamma_{110} + 8\sigma_{111}\Gamma_{111}\right)} \quad \text{(Equation 12)}$$

Nucleation free energy – ideal form

The ideal form for the calculation of the nucleation free energy is given by Equation 13. In this equation, X_{Sc}^0 is the scandium nominal concentration and X_{Sc}^{eq} is the scandium solubility limit.

$$\Delta G_{ideal}^{nuc}\left(X_{Sc}^0\right) = \frac{3}{4}\ln\left(\frac{1 - X_{Sc}^{eq}}{1 - X_{Sc}^{o}}\right) + \frac{1}{4}\ln\left(\frac{X_{Sc}^{eq}}{X_{Sc}^0}\right) \quad \text{(Equation 13)}$$

Nucleation barrier

The nucleation barrier, also known as the formation free energy, is defined by Equation 14.

$$\Delta G_n = n\Delta G_{ideal}^{nuc}\left(X_{Sc}^0\right) + \left(\frac{9\pi}{4}\right)^{1/3} n^{2/3} a^2 \bar{\sigma} \quad \text{(Equation 14)}$$

Condensation rate

The condensation rate is calculated by Equation 15 in which ΔG_{ideal}^{nuc} is the nucleation free energy, $\bar{\sigma}$ is the interface free energy, D_{Sc} is impurity diffusion and X_{Sc}^0 is the scandium nominal concentration.

$$\beta^* = -32\pi\frac{a^2\bar{\sigma}}{\Delta G_{ideal}^{nuc}}\frac{D_x}{a^2}X_{Sc}^0 \quad \text{(Equation 15)}$$

Zeldovitch factor

The Zeldovitch factor is defined by Equation 16, where ΔG_{ideal}^{nuc} is the nucleation free energy and $\bar{\sigma}$ is the interface free energy.

$$Z = \frac{(\Delta G_{ideal}^{nuc})^2}{2\pi(a^2\bar{\sigma})^{3/2}\sqrt{kT}} \qquad (\text{Equation 16})$$

Cluster size distribution

Cluster size distribution defines the probability to encounter a cluster with a dimension of n atoms in a solid solution and also being characterized as $L1_2$ structure. Equation 17 describes this metric.

$$C_n = exp\left(-\frac{\Delta G_{ideal}^{nuc}}{kT}\right) \qquad (\text{Equation 17})$$

Steady-state nucleation rate

Equation 18 describes the steady-state nucleation rate. In this equation, N_S represents the number of nucleation sites (the number of lattice sites), ΔG_n is the nucleation barrier, Z represents the Zeldovitch factor and β^* is the condensation rate for clusters.

$$J^{st} = N_s Z \beta^* exp\left(-\frac{\Delta G_n}{kT}\right) \qquad (\text{Equation 18})$$

3.9 Mechanical computation and statistical mechanics

The global objective of statistical mechanics is predominantly to develop predictive tools for the computation of properties and local structure of material in the state of fluids, solids and even on a phase transition, knowing the nature of molecules that form the systems as well as the intra- and intermolecular interactions.

Statistical mechanics can be defined as a mathematical theory that allows the understanding of the properties of systems with a large numbers of particles ($\sim 10^{23}$). The particles can be in the gas state, liquid state or even the solid state. Knowing the average value for functions of the microscopic quantities, the behavior of the macroscopic variables can be predicted. This is a statistical mechanics approach.

The precision of statistical mechanics depends on three factors: the precision of molecular and intermolecular properties, the precisions of the parameters available for the material in analysis, and the precision of the statistical mechanical theory used for such analysis.

Based on the previous definition, a Monte Carlo method is a statistical mechanics approach.

3.10 Micro and nanotechnology

Phenomena that occur at small scales, namely at nanometer and femtosecond scales, can, and usually have, an enormous impact on what occurs at a larger space and time scale. As the demand (necessity), or simply our pursuit to be able to understand and fully control the behavior of products, modeling and simulating processes at various scales and linking various scales (known as multiscale) have emerged as one of the areas of increasing interest in applied science and engineering.

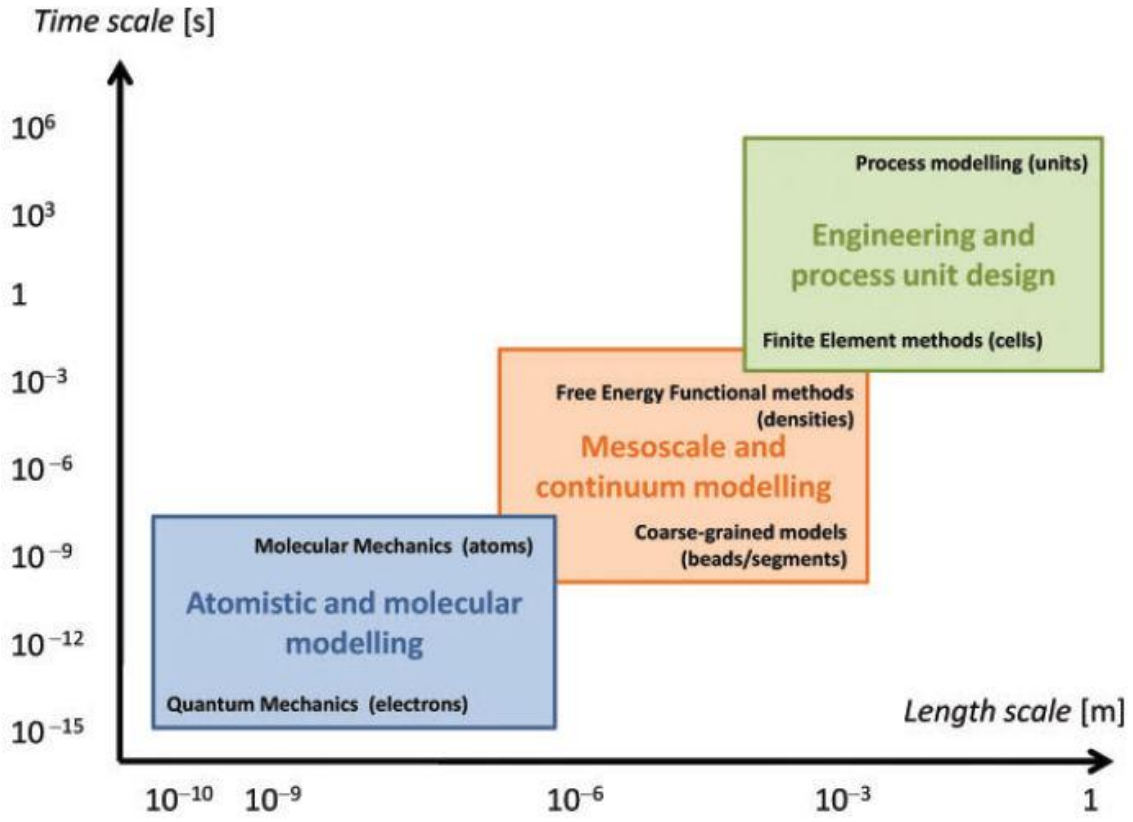


Figure 12 – Boundaries of simulation models.

It is well known that during the previous decade (2000-2010) great advances have been made in the field of nanotechnology. A few examples due to the role of modeling and simulation can be advanced [Lundstrom]:

- The understanding of current flow at the molecular scale;
- Evolution of microelectronics into nanoelectronics;
- Discovery of iron based high temperature superconductors (high-Tc or HTS);
- Improvements in the sensitivity of biosensors.

Table 5 – Ranking of computational based methods [Sarker2009].

Method	**Methods with highest degree of accuracy**
Target	The target of these methods is to research of both electronic and atomic ground state, optical and magnetic properties of weakly interacting and also strongly interacting correlated systems.
Input	The inputs are the atomic species, coordinates, system's symmetry, interaction parameter.
Output	The outputs are the total energy, excitation energy, spin densities, and force on atoms.
Examples	Examples are *ab initio* methods for electronic structure calculations of correlated systems, quantum MC, quantum chemistry and many body.
Method	**Methods with second highest degree of accuracy**
Target	The targets are the accurate calculation of ground state structure by local optimization, calculation of mechanical, magnetic and optical properties of small clusters, and perfect crystals of weakly interacting electron systems, estimation of reaction barrier and paths.
Input	The atomic species and their coordinates, symmetry of the structure, pseudo potential or Hamiltonian parameters for the species considered.
Output	The total energy, charge and spin densities, forces on atoms, electron energy eigenvalues, capability of doing Molecular Dynamics, vibrational modes and phonon Spectrum.
Examples	*ab initio* methods for normal Fermi liquid systems based on either Hartree Fock (HD) or Density Functional Theories.
Method	**Semi empirical methods**
Target	The target is to search for ground state structure by genetic algorithm (GA), simulated annealing (SA) or local optimization if a good guess for the structure is known, simulation of growth or some reaction mechanisms, and calculation of response functions.
Input	The atomic species and their coordinates, the parameters of the inter-particle potential, temperature and parameters of the thermostat or other thermodynamic variables.
Output	Output of tight binding (TB): total energy, charge and spin densities, force on atoms, particle trajectories, phonon calculation, mechanical magnetic and optical properties of clusters and crystals.
Examples	The semi-empirical methods of large systems or long time scale, tight binding or linear combination of atomic orbitals (LCAO), and molecular dynamics based on classical potential or force field.
Method	**Stochastic methods**
Target	Research of long timescale non-equilibrium phenomena such as transport, growth, diffusion, annealing, reaction mechanisms and also calculation of equilibrium quantities and thermodynamic properties.
Input	The parameters of the inter-particle potential, temperature and parameters of the thermostat or other thermodynamic variables.
Output	The statistics of several quantities such as energy, magnetization, atomic displacements.
Examples	Monte Carlo walk towards equilibrium, kinetic or dynamical Monte Carlo (growth and other non-equilibrium phenomena).

In order to predict material design, it is well known the obligation of improving the simulation methods to achieve the maximum accuracy. Over the last decade (2000-2010), significant progress has occurred in the individual components of a predictive materials simulation framework:

- Quantum mechanics (challenge: increased accuracy);
- Force fields (challenge: extract properties for materials design);
- Biological predictions (challenge: simulation in liquids and on biological timescales);
- Mesoscale dynamics (challenge: simulation on relevant time and size scales);
- Integration (challenge: multiscale).

Figure 12 displays the various scale boundaries of the models used in simulation.

The computer based methods used in simulation of various properties of nanoscale systems differ in their level of accuracy and time complexity to perform such calculations. Based on these facts, it is possible to classify the various methods in groups, as Table 5 demonstrates.

4. Methodology used to Simulate the Al_3Sc Precipitation

This chapter will enhance the various aspects of the simulation technique developed in this thesis. This chapter provides a detailed overview of the kinetic Monte Carlo (kMC) method and the background of applicability besides material science. In addition to the specific details of the simulation technique, the parameters employed as well will be presented.

4.1 Atomistic Monte Carlo simulation method

Methods in which algorithms use random numbers or pseudorandom numbers to solve computational problems are generally called Monte Carlo methods. The Monte Carlo simulation applications range from material science to biology to quantum physics besides other fields such as computer imaging, architecture and economics. The name Monte Carlo was first suggested by Nicholas Metropolis as a reference to the famous casino in Monaco.

A Monte Carlo (MC) simulation attempts to follow the dependence of time of a model. A model whose evolution does not progress in a rigorously predefined fashion, but rather in a stochastic manner that depends on a sequence of random numbers generated during the simulation. A second simulation with a different sequence of random numbers will not give identical results but will return values which will agree with those obtained from the first simulation sequence to within same statistical error [Landau2000].

The MC technique is not just applied in processes of equilibrium, but also in non-equilibrium processes that do exist in physics, chemistry and material science. In these processes, time evolution is of great interest. It is true that molecular dynamics is applied to accurately model the evolution of systems at an atomic scale, but never the less the time scales are very limited. Therefore the application of kinetic Monte Carlo is efficient in these dynamical processes.

The phrase “Monte Carlo method” is applied in such a way that it aggregates a vast number of techniques for solving problems based on random numbers and statistical probabilities. The general principle refers that “whatever method that applies random numbers to obtain a solution for a problem is termed as Monte Carlo”. Examples of MC methods that are encountered in the literature are:

- Classical Monte Carlo;
- Metropolis Monte Carlo;
- Quantum Monte Carlo;
- Path-Integral Monte Carlo;
- Simulation Monte Carlo;

- Kinetic Monte Carlo.

In material science, the Monte Carlo method is mostly applied to simulate the microstructural evolution of the system. In general, the system subjected to analysis is not in an equilibrium state and the main goal is to study the kinetics of the processes that lead to equilibrium.

The kinetic Monte Carlo (kMC) has been used in the following physical system simulations:

- Surface diffusion;
- Molecular beam epitaxy (MBE) growth;
- Chemical vapor deposition (CVD) growth;
- Vacancy diffusion in alloys;
- Compositional patterning in alloys driven by irradiation;
- Polymers: topological constraints versus entanglement;
- Coarsening of domain evolution;
- Dislocation motion;
- Defect mobility and clustering in irradiated solids.

Although the upper exposition is far from complete, Monte Carlo is also applied outside of the purely physics domain. Non-physics areas of application of Monte Carlo methods are for example [Landau2012]:

- Protein folding;
- Sociophysics;
- Econophysics;
- Traffics simulations;
- Medicine.

4.2 Atomistic Monte Carlo simulation main steps

In order to eliminate "rejection" in discrete Monte Carlo simulations, Bortz, Kalos and Lebowitz developed the rejection-free n-fold way algorithm. This algorithm is also known as the residence-time algorithm, as the Bortz-Kalos-Liebowitz (BKL) algorithm or most commonly as the kinetic Monte Carlo (kMC) algorithm [Amar2006].

The algorithm calculates all the possible transitions rates $w_{ij} = T_{ij}P_{ij}^{acc}$, for all possible trial configurations j (with $j \neq i$) and for a given initial configuration i, and then directly selects the new configuration j with a probability proportional to w_{ij}. By the moment the configuration is selected, it is always accepted [Amar2006].

Let N be the number of possible new states, then the new configuration j can be selected by first calculating the partial sums $S_0^i = 0$ and $S_n^i = \sum_{k=1}^{n} w_{ik}$ for $n = 1\ to\ N$

and then generating a uniform random number r between $0\ and\ S_N^i$. At this point, a search is performed to find the value of j such that $S_{j-1}^i < r < S_j^i$, after which a transition to state j is carried out [Amar2006].

In a typical dynamic Metropolis Monte Carlo simulation the trial configuration selection rate T_{ij} is a constant $1/\tau$ for all possible transitions. Then, for the corresponding n-fold way simulation, we can write that $w_\alpha = \tau^{-1} P_\alpha^{acc}$, where P_α^{acc} is the acceptance probability for class α. The average time for a transition then depends on the initial configuration i and is given by $\langle \Delta t_i \rangle = t / \sum_\alpha n_\alpha P_\alpha^{acc}$, while the time for a particular transition is given by $\Delta t_i = -\ln(r)\langle \Delta t_i \rangle$, where r is a uniform random number between 0 and 1 [Amar2006].

We will consider a face centered cubic (FCC) structure with N_s sites and N_a atoms. The number of vacancies is $N_v = N_s - N_a$. We will define Z as the number of (nearest) neighbor sites, for each lattice site, with whom atoms may be exchanged. In each Monte Carlo step, the configuration of the structure is defined by indication of the type of atom that occupies every lattice site: type-A atom, type-B atom, or vacancy. Each configuration can lead to a certain number of new configurations in the next MC step. The number of possible new configurations N_{ch} is defined by Equation 19 [Martin2005].

$$\boldsymbol{N_{ch} = N_v * Z - number\_of\_vacancy\_vacancy\_bonds} \qquad \textbf{(Equation 19)}$$

The probability of choosing one of the configurations is given by Equation 20.

$$\boldsymbol{W_{i,j} = \nu_{i,j} exp\left(-\frac{\Delta H_{i,j}}{k_B T}\right)} \qquad \textbf{(Equation 20)}$$

A simulation with one vacancy has a vacancy concentration of $1/N_s$, which is often much larger than a typical equilibrium vacancy concentration C_v^e. Then the time evolution of the simulation is faster than the real process, by a factor equal to the vacancy supersaturation: $(1/N_s)/C_v^e$. The real time t is longer than the Monte Carlo time t_{MC}, calculated by Equation 21.

$$\boldsymbol{t = \frac{t_{MC}}{(N_s C_v^e)}} \qquad \textbf{(Equation 21)}$$

The kinetic Monte Carlo applies what is called as the detailed balance. Equation 22 is designated as the "detailed balance" in which $P_n(t)$ is the probability of the system being in state n at time t, and $W_{n\to m}$ is the transition rate for $n \to m$.

$$\boldsymbol{P_n(t) W_{n\to m} = P_m(t) W_{m\to n}} \qquad \textbf{(Equation 22)}$$

Random Generation

Since MC simulations are commonly very long simulation sequences, they require a fast and efficient production of random numbers. In fact, when we mention random numbers, we should actually mention pseudo-random numbers. Possible algorithms for random number generation are [Landau2012]:

- Congruential method;
- Mixed congruential method;
- Shift register algorithms;
- Lagged Fibonacci generators.

4.3 Theoretical explanation of the simulation of Al_3Sc precipitation with kMC

The composition of an alloy can be expressed in the form of molar fraction, which is also known as atom percentage, or in the form of weight percentage. The atom percentage is the number of moles of an element relatively to the total number of moles in the alloy. The weight percentage represents the weight of a particular component relatively to the total weight of the alloy. The designated "lever rule" allows the calculation of a binary alloy phase amount (Equation 23 and Equation 24).

$$W_A = \frac{wt\ of\ A}{wt\ of\ A + wt\ of\ B} \times 100\% \qquad (\ \text{Equation 23}\ )$$

$$W_B = \frac{wt\ of\ B}{wt\ of\ A + wt\ of\ B} \times 100\% \qquad (\ \text{Equation 24}\ )$$

$$W_A + W_B = 100\% \qquad (\ \text{Equation 25}\ )$$

Based on the lever rule equations, a 10% scandium concentration at a temperature of 600K and analyzing the respective phase diagram we will obtain the phase amounts calculated by Equation 26 and Equation 27.

$$W_{Sc} = \frac{25 - 10}{25 - 2} = 0.65 \qquad (\ \text{Equation 26}\ )$$

$$W_{Al} = \frac{10 - 2}{25 - 2} = 0.35 \qquad (\ \text{Equation 27}\ )$$

$$W_{Sc} + W_{Al} = 1 = 100\% \qquad \text{(Equation 28)}$$

The simulations carried out in the present thesis were conducted with scandium concentrations of 0.5, 0.75, 1, 1.25, 2, 3, 4 and 5 percent. Table 6 shows the respective scandium weight percent.

Table 6 – Scandium weight percent.

Scandium Weight Percent [w_{Sc}]	
0.25% Sc	≈ 0.45
0.50% Sc	≈ 0.85
0.75% Sc	≈ 1.20
1.00% Sc	≈ 1.70
1.25% Sc	≈ 0,20
2.00% Sc	≈ 3.50
3.00% Sc	≈ 5,00
4.00% Sc	≈ 7,00
5.00% Sc	≈ 8,00

4.3.1 Vacancy exchange frequency

The referred diffusion phenomenon occurs via the so called vacancy jumps with one of its twelve first nearest neighbors. The vacancy exchange frequency ($\Gamma_{Al,V}$) with an aluminum neighbor is given by Equation 29.

$$\Gamma_{Al,V} = \nu_{Al} exp\left(-\frac{\Delta E_{Al,V}}{kT}\right) \qquad \text{(Equation 29)}$$

The vacancy exchange frequency ($\Gamma_{Sc,V}$) with a scandium neighbor is given by an equivalent equation (Equation 30).

$$\Gamma_{Sc,V} = \nu_{Sc} exp\left(-\frac{\Delta E_{Sc,V}}{kT}\right) \qquad \text{(Equation 30)}$$

In the previous two equations, ν_{Al} and ν_{Sc} represent an attempt frequency for an aluminum atom and a scandium atom, respectively. $\Delta E_{Al,V}$ and $\Delta E_{Sc,V}$ are the activation energy and represent the necessary energy change required to move the aluminum or scandium atom from its initial stable position to the saddle point position.

4.3.2 Activation energy

The activation energy (ΔE) is the energy barrier that must be exceeded in order to promote a chemical reaction. Figure 13 illustrates this barrier that must be overcame. The next sequence of equations describes the activation energy in respect to an aluminum atom (Equation 31 until Equation 35) and in respect to a scandium atom (Equation 36 until Equation 40). Figure 14 illustrates a possible vacancy surrounding.

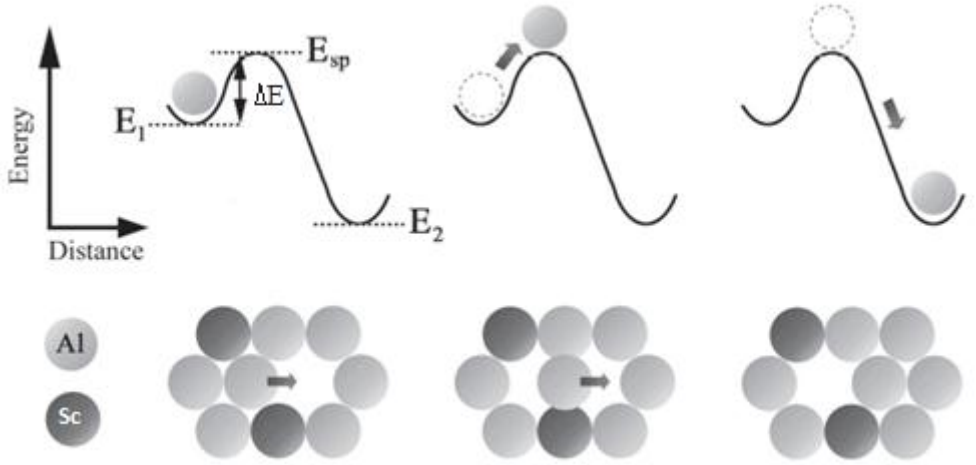


Figure 13 – Energy involved in overcoming a vacancy jump barrier [Geuser2010].

$$\Delta E_{Al,V} = E_{sp,Al} - \sum_{j} \varepsilon V_j^{(1)} - \sum_{j \neq V} \varepsilon Al_j^{(1)} - \sum_{j} \varepsilon Al_j^{(2)} \qquad \text{(Equation 31)}$$

$$\Delta E_{Al,V} = E_{sp,Al} - n_{AlAl}^{(1)} \epsilon_{AlAl}^{(1)} - n_{AlSc}^{(1)} \epsilon_{AlSc}^{(1)} - n_{AlAl}^{(2)} \epsilon_{AlAl}^{(2)} - n_{AlSc}^{(2)} \epsilon_{AlSc}^{(2)} - n_{AlV}^{(1)} \epsilon_{AlV}^{(1)} - n_{ScV}^{(1)} \epsilon_{ScV}^{(1)} \qquad \text{(Equation 32)}$$

In Equation 32, the terms $n_{AlAl}^{(1)}$, $n_{AlSc}^{(1)}$, $n_{AlAl}^{(2)}$, $n_{AlSc}^{(2)}$ represent the number of aluminum-aluminum and aluminum-scandium bonds, at the first and second nearest neighborhoods of an aluminum atom. $n_{AlV}^{(1)}$ and $n_{ScV}^{(1)}$ are the number of aluminum-vacancy and scandium-vacancy bonds at the first nearest neighborhood, respectively.

For an FCC structure there are twelve first nearest neighbor lattice sites (Z_1=12) and six second nearest neighbor lattice sites (Z_2=6), which results in Equation 33, Equation 34, and Equation 35. These three equations allow us to calculate the numbers of bonds necessary to compute $\Delta E_{Al,V}$.

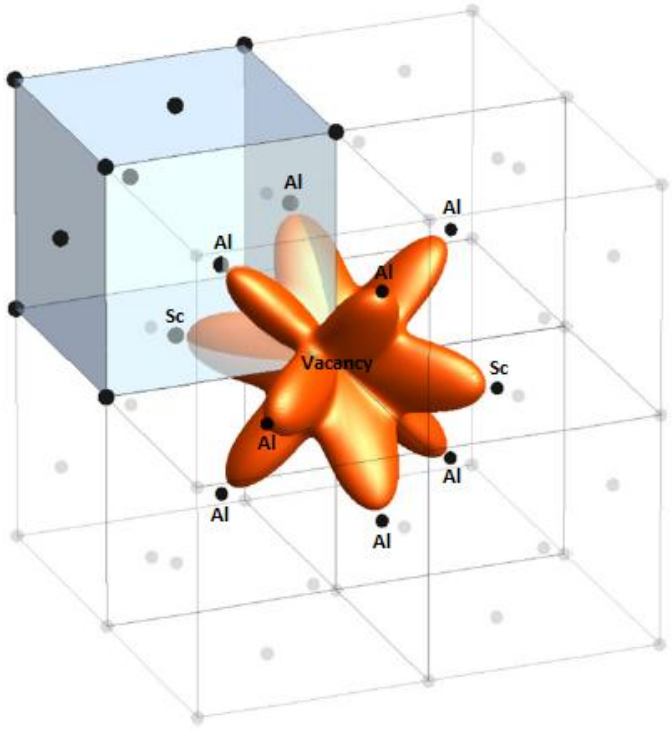


Figure 14 – The twelve first nearest neighbors of a vacancy.

$$n_{AlAl}^{(1)} + n_{AlSc}^{(1)} = Z_1 - 1 \qquad \text{(Equation 33)}$$

$$n_{AlAl}^{(2)} + n_{AlSc}^{(2)} = Z_2 \qquad \text{(Equation 34)}$$

$$n_{AlV}^{(1)} + n_{ScV}^{(1)} = Z_1 \qquad \text{(Equation 35)}$$

$$\Delta E_{Sc,V} = E_{Sp,Sc} - \sum_{j} \varepsilon V_j^{(1)} - \sum_{j \neq V} \varepsilon Sc_j^{(1)} - \sum_{j} \varepsilon Sc_j^{(2)} \qquad \text{(Equation 36)}$$

$$\Delta E_{Sc,V} = E_{Sp,Sc} - n_{AlSc}^{(1)} \epsilon_{AlSc}^{(1)} - n_{ScSc}^{(1)} \epsilon_{ScSc}^{(1)} - n_{AlSc}^{(2)} \epsilon_{AlSc}^{(2)} - n_{ScSc}^{(2)} \epsilon_{ScSc}^{(2)} - n_{AlV}^{(1)} \epsilon_{AlV}^{(1)} - n_{ScV}^{(1)} \epsilon_{ScV}^{(1)} \qquad \text{(Equation 37)}$$

In Equation 37, the terms $n_{ScSc}^{(1)}$, $n_{AlSc}^{(1)}$, $n_{ScSc}^{(2)}$, $n_{AlSc}^{(2)}$ represent the number of scandium-scandium and aluminum-scandium bonds, at the first and second nearest neighborhoods of a scandium atom. $n_{ScV}^{(1)}$ and $n_{AlV}^{(1)}$ are the number of scandium-vacancy and aluminum-vacancy bonds at the first nearest neighborhood, respectively. Assuming an FCC structure, we obtain Equation 38, Equation 39 and Equation 40. These three equations allow us to calculate the numbers of bonds necessary to compute $\Delta E_{Sc,V}$.

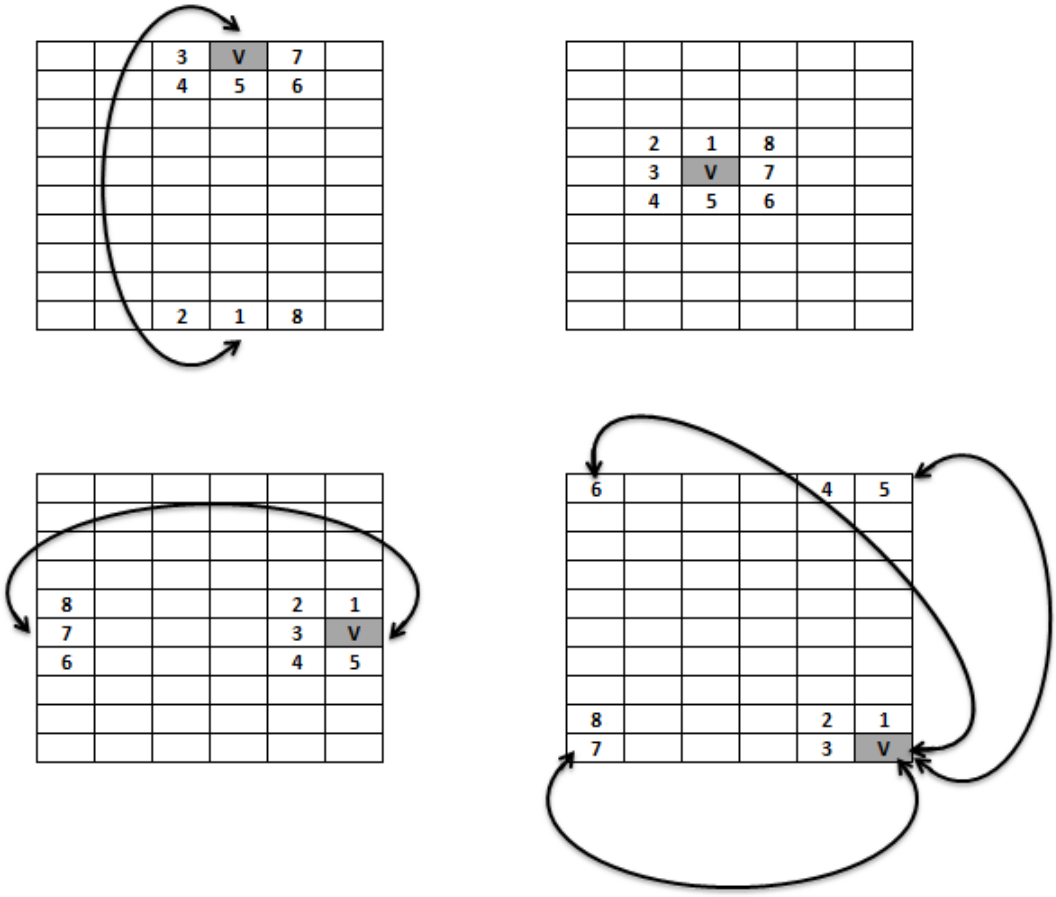


Figure 15 – First nearest neighbors of a vacancy (V) on a 2D lattice.

$$n_{ScSc}^{(1)} + n_{AlSc}^{(1)} = Z_1 - 1 \qquad \text{(Equation 38)}$$

$$n_{ScSc}^{(2)} + n_{AlSc}^{(2)} = Z_2 \qquad \text{(Equation 39)}$$

$$n_{AlV}^{(1)} + n_{ScV}^{(1)} = Z_1 \qquad \text{(Equation 40)}$$

Considering a 2D lattice, for an easier visualization, Figure 15 illustrates four scenarios of a vacancy neighborhood. In three of these cases, the vacancy is located on a lattice edge or vertex, and the vacancy neighborhood has to be defined with the application of periodic boundary conditions (PBC). Figure 16 and Figure 17 show the first and second neighbors of a vacancy in a 3D FCC lattice. Figure 16 illustrates the application of PBC in three dimensions.

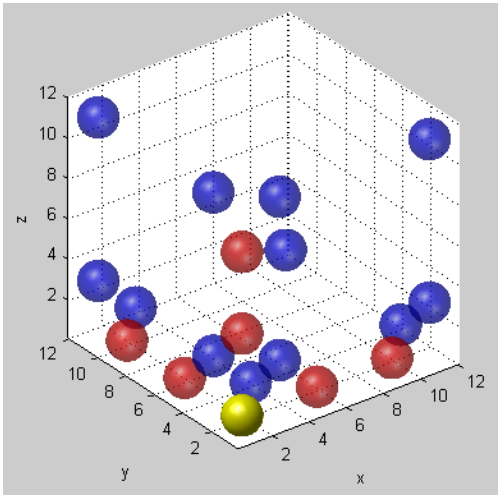


Figure 16 – First (blue atoms) and second (red atoms) nearest neighbors of a vacancy (yellow atom) located at a 3D lattice vertex.

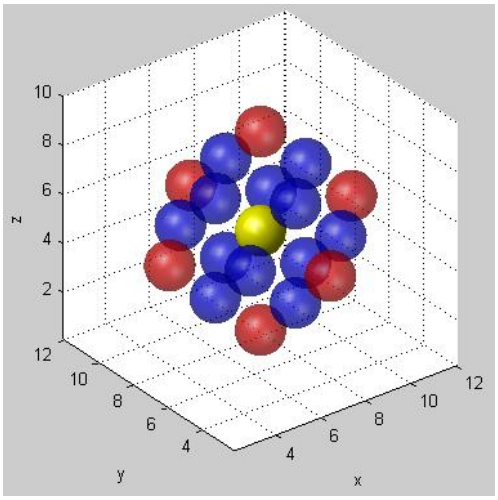


Figure 17 – First (blue atoms) and second (red atoms) nearest neighbors of a vacancy (yellow atom) located inside the 3D lattice.

4.3.3 Jump selection

As mentioned previously, a vacancy site is surrounded by twelve first nearest neighbors. In every step of the kMC algorithm, it is necessary to calculate the jump rate of the vacancy for each of its twelve "jumping candidates". In other words, it is calculated twelve independent jumping frequencies Γ_1, Γ_2, Γ_3, until Γ_{12}. One of these twelve possibilities is selected applying the following procedure:

- generate a random number with a value between 0 and 1;
- calculate the 12 summations of jump frequencies $\sum_{i=1}^{n} \Gamma_i$, with n=1..12;
- select the $\boldsymbol{n}^{\text{th}}$ jump frequency that verifies Equation 41.

$$\sum_{i=1}^{n} \boldsymbol{\Gamma_i} \leq \boldsymbol{random\ number} \leq \sum_{i=1}^{n+1} \boldsymbol{\Gamma_i} \qquad \textbf{(Equation 41)}$$

Figure 18 allows a clear visualization of the application of this procedure. In the example depicted in this figure, the vacancy will jump to its 4$^{\text{th}}$ first nearest neighbor.

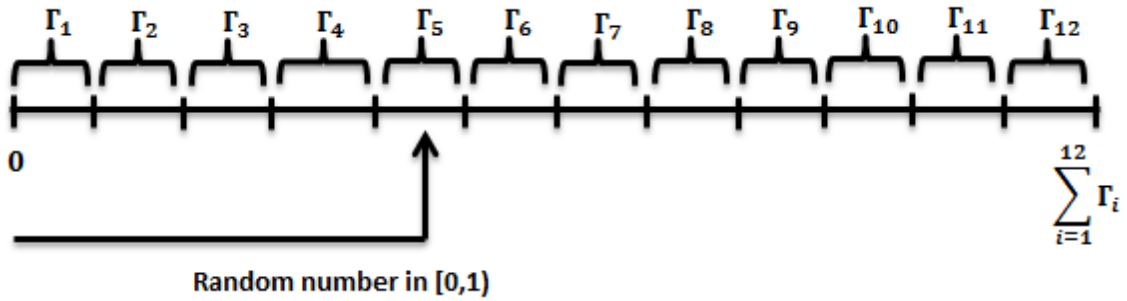


Figure 18 – Random selection of the jump frequency.

To give a clear picture of the jump selection procedure we will use two examples. For example 1, Table 7 contains the distribution of jump frequencies (Γ_n) and the distribution of accumulated jump frequencies ($\sum_{i=1..n}\Gamma_i$), at a certain Monte Carlo step. The jump frequencies and the accumulated values are drawn as a circular graph in Figure 19. The larger the slice associated with a certain neighbor *i*, the higher is the probability of that neighbor to be selected. In example 1, the most probable jumps are neighbors 2 and 3.

Table 7 – Vacancy exchange frequency: example 1.

n	Γ_n	$\Sigma_{i=1..n}\Gamma_i$
1	0.034580	0,034580
2	0.327099	0,361679
3	0.327099	0,688778
4	0.034580	0,723358
5	0.034580	0,757938
6	0.034580	0,792518
7	0.034580	0,827098
8	0.034580	0,861678
9	0.034580	0,896258
10	0.034580	0,930838
11	0.034580	0,965418
12	0.034580	1,000000

The second example is demonstrated by Table 8 and Figure 20. In this example, there are 3 equally probable jumps: neighbors 2, 3 and 6. The other jumps have negligible probability.

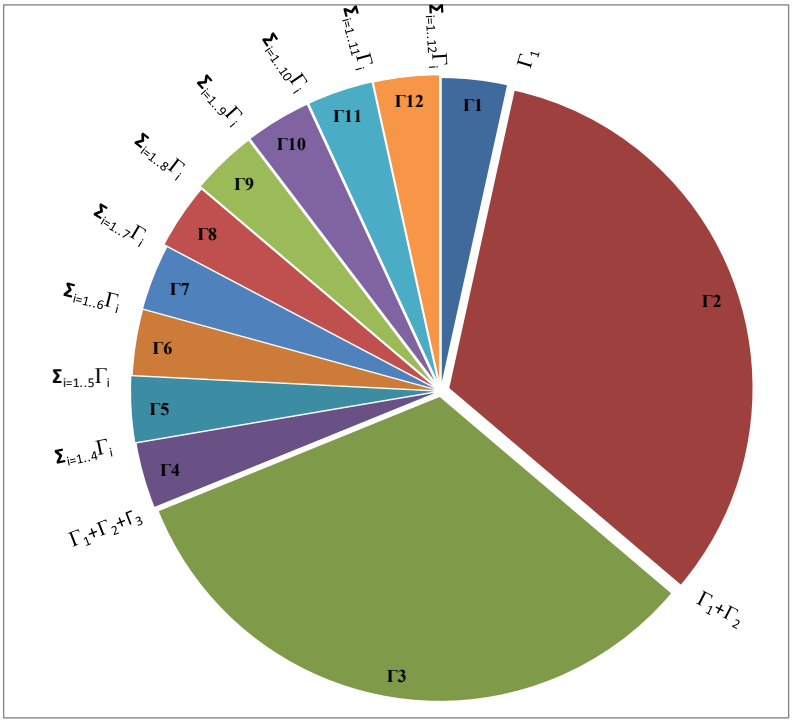


Figure 19 – Vacancy exchange frequency distribution: example 1.

Table 8 – Vacancy exchange frequency values (Γ_i): example 2.

n	Γ_n	$\Sigma_{i=1..n}\Gamma_i$
1	0.000041	0.000041
2	0.333210	0.333251
3	0.333210	0.666461
4	0.000041	0.666502
5	0.000041	0.666543
6	0.333210	0.999753
7	0.000041	0.999794
8	0.000041	0.999835
9	0.000041	0.999876
10	0.000041	0.999917
11	0.000041	0.999958
12	0.000041	1.000000

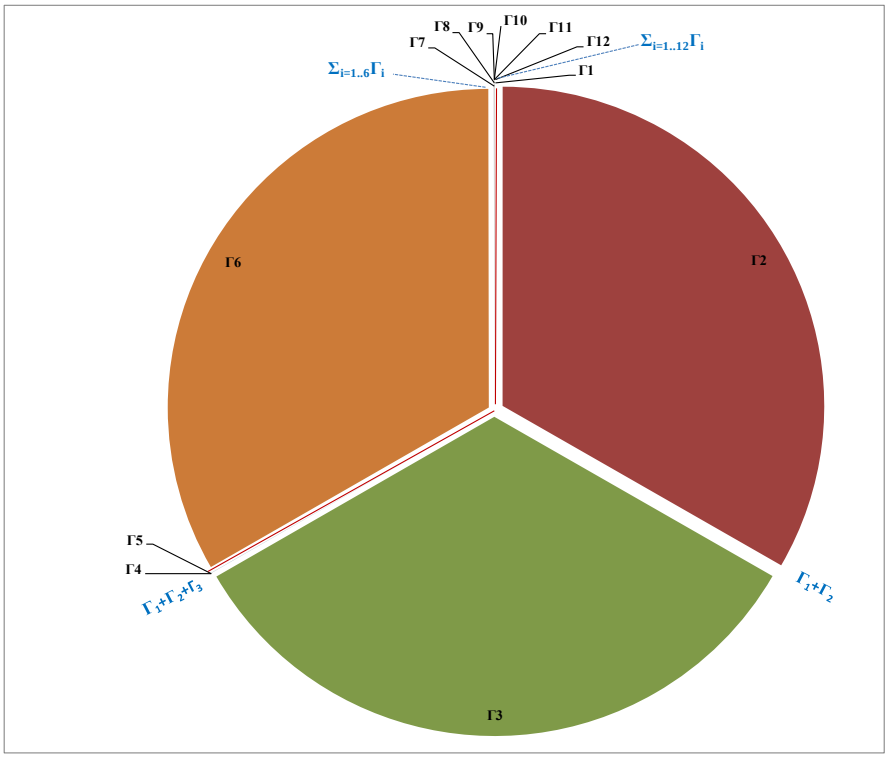


Figure 20 – Vacancy exchange frequency distribution: example 2.

4.3.4 Real time

Equation 42 computes the kMC average residence time, also known by time step increment of the kMC algorithm (t_{MC}^{sim}).

$$t_{MC}^{sim} = \left(\sum_{i=1}^{12} \Gamma_i \right)^{-1} \qquad \text{(Equation 42)}$$

This is not the real time increment of a MC step. To calculate the real time associated with a step (t_{MC}^{real}), we have to multiply the simulated time (Equation 42) by a factor that takes in to account the difference between the simulated (C_V^{sim}) and the real vacancy concentration (C_V^{real}). As so the real time increment of the kMC algorithm is given by Equation 43.

$$t_{MC}^{real} = \left(\frac{C_V^{sim}}{C_V^{real}} \right) * \left(\sum_{i=1}^{12} \Gamma_i \right)^{-1} = \left(\frac{C_V^{sim}}{C_V^{real}} \right) * t_{MC}^{sim} \qquad \text{(Equation 43)}$$

4.3.5 Real vacancy concentration

One of the biggest challenges in kMC is to get real time from a simulation that does not deal with time explicitly. To do so, we need the simulated vacancy concentration and an estimate for the real vacancy concentration. In the literature there is not a unique approach to this problem. In [Clouet2004a] the real time increment is calculated with Equation 44.

$$\boldsymbol{t_{MC}^{real} = \frac{1}{N_S * \left(1 - 13x_{Sc}^{0}\right) * C_V^{real}} * t_{MC}^{sim}} \qquad \textbf{(Equation 44)}$$

where N_s is the number of lattice sites, x_{Sc}^{0} is the nominal concentration of Sc in the lattice, C_V^{real} is real vacancy concentration of Sc in pure Al and t_{MC}^{sim} is computed with Equation 42. In [Binkele2003], where a BCC lattice is simulated, the real time increment is calculated with Equation 45.

$$\boldsymbol{t_{MC}^{real} = \frac{1}{N_S * C_V^{real}} * t_{MC}^{sim} = \frac{1}{N_S * 280 * \exp(\frac{E_V^F}{kT})} * t_{MC}^{sim}} \qquad \textbf{(Equation 45)}$$

where N_s is the number of lattice sites, E_V^F is the vacancy formation energy in pure Al and t_{MC}^{sim} is computed with an equation similar to Equation 42, changing the number of sums from 12 (FCC lattice) to 8 (BCC lattice). Assuming that the number of vacations (N_V) is given by Equation 46, the vacancy formation energy is calculated with Equation 47.

$$\boldsymbol{N_V = N_S * exp\left(\frac{-E_V^F}{kT}\right)} \qquad \textbf{(Equation 46)}$$

$$\boldsymbol{E_V^F = -kT * \ln(\frac{N_V}{N_S})} \qquad \textbf{(Equation 47)}$$

The vacancy formation energy E_V^F (Equation 47) and then the number of vacations N_V (Equation 46) can be computed if we know the following quantities:

- The number of vacancies in aluminum ($\boldsymbol{N_V}$) is $7.55\times10^{23}/m^3$, at equilibrium and a temperature $\boldsymbol{T}$=500ºC;
- The atomic weight of aluminum ($\boldsymbol{Z_{Al}}$) is 26.981538 g/mol, at $\boldsymbol{T}$=500ºC;
- The number of atoms per mol ($\boldsymbol{N_A}$), or the Avogadro's number, is 6.02×10^{23};
- The aluminum density is (ρ_{Al}) is 2.62 g/cm^3, at $\boldsymbol{T}$=500ºC;
- The Boltzmann's constant ($\boldsymbol{k}$) is $8.62 \times 10^{-5} \frac{eV}{K}$.
- The number of aluminum atoms in 1 m^3 ($\boldsymbol{N_S}$) is obtained with Equation 48.

$$N_S = \frac{N_A * \rho_{Al}}{Z_{Al}} \qquad \text{(Equation 48)}$$

With the previous values, the computed vacancy formation energy is 0.75 eV for a temperature of 500ºC. Other authors consider E_V^F = 0.66 eV.

Since there is no consensual form of computing the real time increment, we decided to make use of experimental data for the real vacancy concentration, which is plotted in Figure 21 [Hatch1984].

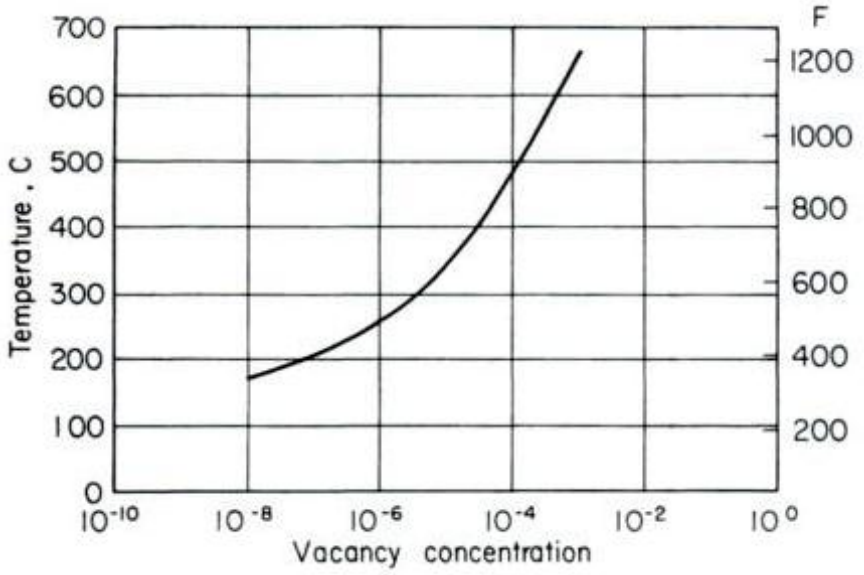


Figure 21 – Experimental vacancy concentration in pure Al as a function of temperature [Hatch1984].

Making use of Eureqa Formulize[1], a data mining software that searches for mathematical patterns from tabulated data, it was possible to approximate the experimental vacancy concentration, plotted in Figure 21, with an analytic expression (Equation 49).

$$\begin{aligned} C_{V_{real}} = & -0.005792301654 + 5.281432466e^{-5} \times T \\ & - 1.916781695e^{-7} \times T^2 \\ & + 3.466630615e^{-10} \times T^3 \\ & - 3.132467044e^{-13} \times T^4 \\ & + 1.135950846e^{-16} \times T^5 \end{aligned} \qquad \text{(Equation 49)}$$

[1] http://www.nutonian.com/documentation

4.3.6 Precipitation measurements

It is critical to establish a criterion to decide if a cluster of atoms is a stable precipitate or not. To establish this criterion, we must be able to identify the atoms that belong to each cluster and decide if it is stable or not. We adopted the following criterion to classify a $L1_2$ cluster as an Al_3Sc precipitate: (i) *all the 12 first nearest neighbors of a scandium atom are aluminum, and (ii) at least one of its 6 second nearest neighbors is scandium, and (iii) the number of Sc atoms in the cluster is higher than a critical value* n^*_{Sc}. We used a critical value of 13, which is suggested by the classical nucleation theory.

Stable precipitates average size $\langle n_{Sc} \rangle_{sp}$

The calculation of the precipitates average size is based on the number of clustered atoms divided by the number of clusters.

Solid solution concentration

The solid solution concentration represents the number of scandium atoms that do not belong to stable precipitates. Atoms located in clusters whose size is smaller than n^*_{Sc} are considered to be in solid solution. Based in these principles, Equation 50 calculates the solid solution concentration.

$$x_{Sc} = \sum_{n_{Sc}=1}^{n^*_{Sc}} n_{Sc} C_{n_{Sc}} \qquad \text{(Equation 50)}$$

where $C_{n_{Sc}}$ is the number of $L1_2$ clusters containing n_{Sc} scandium atoms, normalized by the number of atoms in the lattice.

Precipitates concentration

The precipitates concentration is simply the difference between the nominal scandium concentration and the solid solution concentration.

Mean radius

To calculate the mean radius of a precipitate, it is assumed that the precipitate is a sphere having the same volume as this geometric figure. In this case, Equation 51 and Equation 52 are applied to obtain the volume and mean radius of an Al_3Sc precipitate:

$$V_{sphericalPrecipitate} = \frac{4}{3}\pi R^3 = \frac{N}{2}a^2 \qquad \text{(Equation 51)}$$

where N is the number of atoms in the precipitate, a is the FCC lattice constant, and R is spherical precipitate radius.

$$R = \left(\frac{3a^2}{8\pi}\right)^{1/3} \qquad \text{(Equation 52)}$$

4.4 Simulation parameters

The parameters summarized in Table 9, Table 10 and Table 11, used in the kinetic Monte Carlo simulation implementation, were obtained from the work of Emmanuel Clouet [Clouet2004a].

Table 9 – First and second nearest-neighbor pair effective energies (in eV).

Parameter	Value	Unit
$\epsilon^{(1)}_{AlAl}$	-0.560	eV
$\epsilon^{(1)}_{ScSc}$	-0.650	eV
$\epsilon^{(1)}_{AlSc}$	-0.759+21.0x10^{-6}T	eV
$\epsilon^{(1)}_{VV}$	-0.084	eV
$\epsilon^{(2)}_{AlSc}$	+0.113-33.4x10^{-6}T	eV
$\epsilon^{(1)}_{AlV}$	-0.222	eV
$\epsilon^{(1)}_{ScV}$	-0.757	eV

Table 10 – Kinetic parameters: contribution of the jumping atom to the saddle point energy e^{sp}_{α} and the attempt frequency ν_{α}, for $\alpha \in \{Al, Sc\}$.

Parameter	Value	Unit
e^{sp}_{Al}	-8.219	eV
e^{sp}_{Sc}	-9.434	eV
ν_{Al}	1.36x10^{14}	Hz
ν_{Sc}	4x10^{15}	Hz

From the energies $\epsilon^{(1)}_{AlAl}$ and $\epsilon^{(1)}_{ScSc}$ it is possible to calculate $\epsilon^{(2)}_{AlAl} = \frac{\epsilon^{(1)}_{AlAl}}{2}$ and $\epsilon^{(2)}_{ScSc} = \frac{\epsilon^{(1)}_{ScSc}}{2}$.

Table 11 – Second nearest neighbor pair effective energies (in eV).

Parameter	Value	Unit
$\epsilon^{(2)}_{AlAl}$	-0.280	eV
$\epsilon^{(2)}_{ScSc}$	-0.325	eV

4.5 Deduction of simulation parameters

This section presents a brief overview of the parameters deduction.

Cohesive energies

The cohesive energy of a crystal structure represents the amount of energy that is required to separate the crystal structure apart into a set of free atoms:

$$\boldsymbol{cohesive\ energy} = \boldsymbol{energy\ of\ the\ free\ atom} - \boldsymbol{cystal\ energy} \qquad \text{(Equation 53)}$$

Literature mentions that the magnitude of the cohesive energy is not the same in different solids, ranging from 1 to 10 eV/atom.

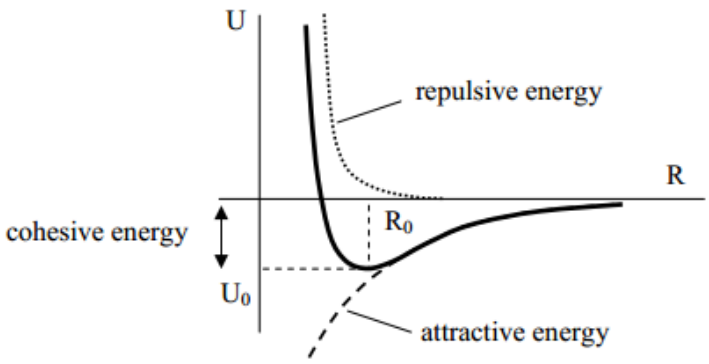


Figure 22 – The various energies exerted between atoms.

Figure 22 is a layout of the typical potential energy that models the interactions between two atoms. At the distance $R = R_0$ the potential energy reaches its minimum. For $R > R_0$ the potential energy has a gradual increase and approaches 0 at $R = \infty$. For $R < R_0$ the potential energy increases very rapidly and tends to infinity at $R = 0$. As systems tend to have the lowest possible energy, the situation of most stability is at $R = R_0$ designated as the equilibrium atomic distance. The corresponding energy U_0 is the cohesive energy.

Mentioned this, the cohesive energy indicates the inter-atomic potential energy per atom at the most stable state.

$$\boldsymbol{E_{coh,Al}} = \frac{\boldsymbol{z_1}}{\boldsymbol{2}}\boldsymbol{\varepsilon_{AlAl}^{(1)}} + \frac{\boldsymbol{z_2}}{\boldsymbol{2}}\boldsymbol{\varepsilon_{AlAl}^{(2)}} \qquad \text{(Equation 54)}$$

$$E_{coh,Sc} = \frac{z_1}{2}\varepsilon_{ScSc}^{(1)} + \frac{z_2}{2}\varepsilon_{ScSc}^{(2)} \qquad \text{(Equation 55)}$$

where Z_1=12 (Z_2=6) is the number of first (second) nearest neighbors in a FCC lattice and the ε's are the pair effective energies presented before.

Mixing energy

$$\omega_{AlSc} = \frac{z_1}{2}\left(\varepsilon_{AlAl}^{(1)} + \varepsilon_{ScSc}^{(1)} - 2\varepsilon_{AlSc}^{(1)}\right) + \frac{z_2}{2}\left(\varepsilon_{AlAl}^{(2)} + \varepsilon_{ScSc}^{(2)} - 2\varepsilon_{AlSc}^{(2)}\right) \qquad \text{(Equation 56)}$$

Vacancy formation energies

Vacancy formation energy indicates how much energy is needed to form a vacancy. If this value is negative, this indicates that energy is released while forming a vacancy.

The vacancy formation energy for an aluminum atom and a scandium atom are presented in Equation 57 and Equation 58.

$$E_{V,for,Al} = z_1\varepsilon_{AlAl}^{(1)} + E_{coh,Al} \qquad \text{(Equation 57)}$$

$$E_{V,for,Sc} = z_1\varepsilon_{ScSc}^{(1)} + E_{coh,Sc} \qquad \text{(Equation 58)}$$

Knowing the cohesive energies, the mixing energy and the vacancy formation energies, we have all the information necessary to deduce the pair effective energies.

4.6 Implementation of the kMC algorithm

The kMC algorithm implementation was developed in two stages: (i) in the initial stage it was implemented in Matlab, with the intention of developing faster a standalone application with an appealing graphical user interface and (ii) in the second stage we migrate to an implementation in C language.

Appendix B contains the flowcharts for the modules used by the Matlab implementation of the Monte Carlo simulation. This appendix includes flowcharts for the following modules:

- Main
- Monte Carlo Main
- Monte Carlo
- Construct 3D FCC Lattice
- Random Vacancy Lattice
- Nearest Neighbor
- Activation Energy
- Vacancy Exchange Frequency
- Random Vacancy Selection
- Real Time Calculation
- Visualization and Report.

After having explored the Matlab version, it was decided to migrate to an implementation in C language. This decision was sustained mainly on three reasons. First, and of great importance, to overcome the performance limitations presented by the Matlab version. Second, to be able to run the simulations in an open source environment, using a free compiler, such as GCC, on Windows or Linux operating system. Third, to be able to utilize the computational resources available on the SeARCH cluster. The SeARCH cluster could be used to accelerate simulations in 3 ways: (i) running multiple sequential simulations at same time, with different parameters, (ii) running a parallel simulation on the same machine using OpenMP, or (iii) running a parallel simulation on several machines using MPI. The last option was not tried since the second alternative was implemented and did not succeed on accelerating the sequential version. The OpenMP version will be explained later in this document.

Appendix G and Appendix H contain the Matlab code for the initial implementation of the kMC simulation of Al_3Sc precipitation, and a few screenshots of its graphical user interface. This code is composed of the files listed in Table 12.

Table 12 – Matlab functions of the initial implementation of kMC.

activationEnergy.m	al_sc_parameters.m
fcclattice.m	neighbors.m
pdbDataFile.m	precipitation.m
printable_report_simulation.m	randomVacancySelection.m
realTime_Calculation.m	vacancyExchangeSelection.m
vtkDataFile.m	

Appendix I documents the C code developed in the present thesis for (i) the kMC simulation, (ii) the clustering analysis with DBSCAN and (iii) the classical nucleation theory (CNT). This code is composed by the files listed in Table 13.

Table 13 – All C files developed in this thesis.

Source and header files for the C implementation of kMC, DBSCAN and CNT	
activationEnergy.c	ArrayList.c
ArrayList.h	ArrayUtil.c
ArrayUtil.h	clustering1.c
cnt_config.c	cnt_functions1.c
cnt_includes.h	cnt_main.c
config.c	dbscan1.c
dbscan2.c	fcclattice.c
main.c	main_includes.h
main2.c	neighbors.c
readFiles.c	utils.c
writeFiles.c	

Figure 23 and Figure 24 describe, in pseudocode, the main steps of the kMC simulation algorithm developed in C language.

```
main:
  Read the simulation configuration file
  Allocate memory space to store the lattice (atoms' coordinates, type and neighbors)
  Compute the coordinates of all FCC lattice sites
  Compute the correction factor to apply to each MC step simulation time → tsCorrection
  Compute the position of the 1st and 2nd nearest neighbors of all sites
  Write initial configuration (atoms' position and type) to file
  // start the Monte Carlo simulation
  Initialize the simulated time to zero → timeSim
  Define the number of current snapshot to 1 → numSnap
  Save the initial simulation time read from the system → start

  // run 'mcs_compAvgTime' MC steps to compute an average step time
  avgStepTime = 0
  mcs = 0
  while (mcs < mcs_compAvgTime) do
      Calculate the activation energy → Eact
      Calculate the vacancy exchange frequency and the real time of this MC step → vEF, ts
      ts = ts*tsCorrection  // corrected simulated time for current MCS
      timeSim = timeSim + ts  // accumulated simulated time
      avgStepTime = averageStepTime + ts
      Select a 1st nearest neighbor for the new position of the vacancy
      Swap the vacancy with the selected neighbor
      Increment mcs
  endWhile
```

Figure 23 – Kinetic Monte Carlo algorithm in pseudocode (part 1).

```
avgStepTime = avgStepTime / mcs_compAvgTime // average step time
rejectStepTime = rejectStepTime *FACTOR_REJECT_MCS // rejection step time

// run the remaining steps of MC simulation
while (mcs < TOTAL_MCS) do
    Calculate the activation energy → Eact
    Calculate the vacancy exchange frequency and the real time of this MC step → vEF, ts
    ts = ts*tsCorrection  // corrected simulated time for current MCS
    // step time exceed a threshold that is considered a computation error
    if (ts > rejectStepTime) then
        Increment errorSteps
        ts = avgStepTime // replace computed step time by average step time
    endIf
    timeSim = timeSim + ts // accumulated simulated time
    Select a 1st nearest neighbor for the new position of the vacancy
    Swap the vacancy with the selected neighbor
    // if it is a snapshop point
    if (mcs = snapshots[numSnap]) then
        Save simulation data to VTK | PDB | XYZ file(s)
        snapshotTime[numSnap-1] = timeSim // save the snapshot time
        Increment numSnap
    endIf
    Increment mcs
endWhile
Save the final simulation time read from the system → end
timeRun = end - start
Print "The duration of simulation is" timeRun
Print "Simulated time is" timeSim
Print "Number of steps with a erroneous duration is" errorSteps
Write a simulation report to file
Free allocated memory space
end main
```

Figure 24 – Kinetic Monte Carlo algorithm in pseudocode (part 2).

Figure 25 presents pseudocode of the function that writes the kMC simulatin report to a text file. The report details several input information of the simulation and details the real time obtained at each snapshot moment of the simulation as well as the computational time of the entire simulation.

```
writeReportFile:
  Get the current time
  Convert current time to local time representation
  Convert date and time to the standard format
  Generate the report file name
  Open report file
  Write header to file (date, hardware, and operating system information)
  Write simulation material information to file
  Write simulation parameters information to file
  Write snapshots simulated time to file
  Write simulation duration to file
  Close report file
end writeReportFile
```

Figure 25 – Function that writes a kMC simulatin report to file in pseudocode.

The kMC simulation is controlled by a number of input parameters. These parameters are input into the simulation through a configuration text file. The configuration file holds the parameters not just for the Monte Carlo simulation, such as the number of steps, the temperature applied, the dimension of the simulation box (lattice dimension), the aluminum and scandium simulation parameters (energy parameters), but also for the DBSCAN clustering analysis. Table 14 shows an example of the configuration file with all the input parameters necessary to the MC simulation and the posterior clustering analysis.

The C implementation of the kinetic Monte Carlo algorithm is composed of the files identified in Table 15. These files are documented in Appendix I. To compile all the source files with GCC the following command was used[2]:

gcc -O3 activationEnergy.c config.c utils.c fcclattice.c neighbors.c writeFiles.c main.c -lm -o main_O3

Appendix C presents an example of a Monte Carlo simulation report in which it is detailed the information of the number of aluminum and scandium atoms defined initially, the number of Monte Carlo steps, the temperature applied, the simulated time at each snapshot as well as the total computational time of the simulation.

[2] The C code was compiled with the "*-O3*" optimization option to achieve the best performance.

Table 14 – Configuration file containing the parameters for MC simulation and DSBSCAN.

Parameter	Parameter Value	Parameter description
al_Radius	1.18	Aluminum atom radius (Angstrom)
sc_Radius	1.84	Scandium atom radius (Angstrom)
Lc	4.05	Aluminum lattice constant (Angstrom)
nbz	50	Number of unit cells in the z direction
nby	50	Number of unit cells in the y direction
nbx	50	Number of unit cells in the x direction
num_Al_Sites	0	Number of Aluminum atoms/sites
num_Sc_Sites	0	Number of Scandium atoms/sites
num_V_Sites	0	Number of Vacancy sites
ScC	1.0	Scandium percentage (0.0-100.0)
mcs	5e11	Simulation Monte Carlo steps
T	873.15	Simulation temperature (Kelvin)
snap	10	Number of snapshots to save during simulation
kB	8.6173324e-5	Boltzmann Constant (ElectronVolt/Kelvin)
wrXYZ	FALSE	Don't Write XYZ files
wrVTK	TRUE	Write VTK files
wrPDB	FALSE	Don't Write PDB files
wrReport	TRUE	Write simulation report file
coordFile	AlSc_50x50x50	Base name of the output file(s)
selectedType	SC	Selected type(s) of atoms to write in VTK file (AL, SC, VACANCY, ALSC, ALV, SCV, ALL)
typeVTKdata	OnlyAtoms	Type of data to write in VTK file: only atoms (OnlyAtoms) OR atoms and cubes (AtomsCubes)
E_AlAl_1	-0.560	1st nearest neighbor pair effective energy (Al-Al)
E_ScSc_1	-0.650	1st nearest neighbor pair effective energy (Sc-Sc)
E_AlSc_1	-0.759	1st nearest neighbor pair effective energy (Al-Sc) -0.759+21*10^-6*T
E_AlSc_2	0.113	2nd nearest neighbor pair effective energy (Al-Sc) 0.113-33.4*10^-6*T
E_AlAl_2	0.0	2nd nearest neighbor pair effective energy (Al-Al)
E_ScSc_2	0.0	2nd nearest neighbor pair effective energy (Sc-Sc)
E_AlV_1	-0.222	1st nearest neighbor pair effective energy (Al-Vacancy)
E_ScV_1	-0.757	1st nearest neighbor pair effective energy (Sc-Vacancy)
e_spAl	-8.219	Aluminum saddle point energy
e_spSc	-9.434	Scandium saddle point energy
vAl	1.36e14	Aluminum attempt frequency
vSc	4e15	Scandium attempt frequency
eps	6.20	The radius used to define the neighborhood of each atom (DBSCAN algorithm)
minPts	4	Minimum number of neighbors that makes an atom to be a core atom of a cluster (DBSCAN algorithm)
storeDistances	FALSE	Store (TRUE) or do NOT store (FALSE) the distance between each pair of atoms in advance (DBSCAN algorithm)

Table 15 – Source and header files for the C implementation of kMC.

activationEnergy.c	config.c
fcclattice.c	main.c
main_includes.h	neighbors.c
utils.c	writeFiles.c

4.7 Kinetic Monte Carlo implementation with OpenMP

Next it is presented the algorithm of the main function used to implement the kinetic Monte Carlo simulation with multiple threads of execution, through the OpenMP library (Figure 26 and Figure 27). The lines starting with #pragma omp specify OpenMP directives, for example to create the parallel threads or to synchronize threads.

Table 16 summarizes the computation time needed by a MC simulation with different number of threads. The number of MC steps simulates was 10^7, the lattice included 10*10*10*4 atoms, we used C code with OpenMP, and the code was compiled with **gcc** and the "*-O3*" optimization level.

Table 16 - Computation time, needed by a MC simulation, as a function of the number of threads.

Number of threads	Average computation time (s)
1	25
2	46
4	52
8	62
12	70

As we can see from Table 16, the utilization of an increasing number of threads is counterproductive. For this reason, the effort of coding in this thesis was put on an optimized sequential version of MC, compiled with "*-O3*" option.

The poor performance achieved by the presented parallel implementation results from 3 facts: (i) the problem we are dealing with is not inherently parallel, since the MC simulation has only one vacancy, (ii) the work assigned to each thread is small and does not compensate the computation overhead introduced by the threads, and (iii) there are several parts of the code that have to be executed by one thread only.

main_OMP:

Read the simulation configuration file

Check if the selected number of threads (*numThreads*) to use on execution is in range [1:12], otherwise terminate the simulation

Allocate memory space to store the lattice: atoms' coordinates, type and neighbors

Compute the coordinates of all FCC lattice sites

Compute the position of the 1st and 2nd nearest neighbors of all sites

Write initial configuration (atoms' position and type) to file

// Start the Monte Carlo simulation

Initialize the simulated time (*totalT*) to zero

Define the number of current snapshot (*numSnap*) to 1

Save the initial simulation time read from the system → *start*

// Specify the number of threads to be created (numThreads)

omp_set_num_threads(numThreads)

Initialize the MC step (*mcs*) to zero

// Create the threads

#pragma omp parallel private

(idT,i,j,nPos,nType,nnPos,nnType,n_AlAl_1,n_AlSc_1,
n_ScSc_1,n_AlV_1a,n_ScV_1a,n_AlAl_2,n_AlSc_2,n_ScSc_2,expoent)

{

idT = *omp_get_thread_num()* *// ID of each thread*

nT = *omp_get_num_threads()* *// Number of threads*

while (*mcs*<numberMCStoSimulate) **do**

if (*idT* = 0) **then** *// This section is run by thread with id=0 only*

Count the number of vacancy's first neighbors of Al and Sc type

endIf

#pragma omp barrier *// Synchronize all threads*

Calculate the activation energy associated with ***i***-th vacancy neighbor

i = *idT*

while (*i* < NUMBER_1ST_NEIGHBORS) **do**

Compute the activation energy associated with i-th vacancy neighbor using [Binkele&Schmauder 2003] equation 3 → *Eact[i]*

i = *i* + *nT*

endWhile

Compute the absolute vacancy exchange frequency with its 12 1st neighbors → *vEF[i]*

Figure 26 – Kinetic Monte Carlo algorithm using OpenMP in pseudocode (part 1).

```
#pragma omp barrier // Synchronize all threads
    if (idT = 0) then // This section is run by thread with id=0 only
        Compute the sum of all 1st neighbors absolute exchange frequencies →
            sumAbsoluteVef
    endIf
    #pragma omp barrier // Synchronize all threads
    Compute the relative vacancy exchange frequency with its 1st neighbors →
        vEF[i] = vEF[i]/sumAbsoluteVef
    if (idT = 0) then // This section is run by thread with id=0 only
        Compute the sum of all relative vacancy exchange frequencies with 1st neighbors
        // Update the simulated time with the real time associated with the present MC step
        totalT = totalT + 1/sumAbsoluteVef
        Select randomly a 1st nearest neighbor for the new position of the vacancy,
            based on the vacancy exchange frequencies
        Swap the vacancy with the selected neighbor
        if (mcs = snapshots[numSnap]) then
            Save simulation data (atoms' type & position) to VTK/XYZ/PDB file at snapshop
            Increment the number of the current snapshot (numSnap)
        endIf
    endIf // (end of) This section is run by thread with id=0 only
    #pragma omp barrier // Synchronize all threads
    if (idT = 0) then
        Increment the MC step (mcs)
    endIf
    #pragma omp barrier // Synchronize all threads
endWhile // (end of) cycle relative to the number of MCS
} // (end of) multiple threads
Save the final simulation time read from the system → end
Computation time = end - start
Print the total simulated time (totalT)
Write a simulation report to file
Free allocated memory space
end main_OMP
```

Figure 27 – Kinetic Monte Carlo algorithm using OpenMP in pseudocode (part 2).

4.8 Clustering analysis

4.8.1 Density Based Clustering with Noise (DBSCAN) algorithm

One of the data mining most crucial methods is designated as clustering analysis. The main goal of clustering analysis is basically dividing data into groups, designated as clusters, which share certain characteristics. In other words, within each identified cluster, the objects are similar or related to one another and also different or unrelated to the objects of other identified clusters.

The concept of clustering is used in this work to identify Al_3Sc precipitates in a spatial dataset, which in our case is a 3D matrix representing the atoms of the simulated lattice. The implemented clustering algorithm is designated by DBSCAN, which stands for Density Based Spatial Clustering of Applications with Noise [Ester1996].

A representative list of clustering algorithms adequate for dealing with large spatial datasets includes:

- CLARANS: clustering large applications based on randomized search;
- DBCLASD: distribution based clustering of large spatial databases;
- DBSCAN: density based clustering of applications with noise;
- OPTICS: ordering points to identify the clustering structure.

As a member of the density based clustering approaches, DBSCAN identifies regions of high density agglomerations in an immense low density surrounding. Its major advantage is that objects with arbitrary shape can be identified and as so it does not need, as other methods do, to presuppose clusters with a given shape. Another advantage is that it does not require a predefined input number of clusters in the data that is analyzed like the method k-means. DBSCAN introduces the notion of noise, used to label atoms that are in low dense regions, which revealed to be an adequate feature in our case.

Focusing on the algorithm and in a simple manner, for each cluster identified, a point of that cluster is a core point if it has in its neighborhood (with a predefined radius *eps*) a predefined minimum number of points (*minPts*). DBSCAN classifies points as being:

- Core point: a point in the interior of the density based cluster;
- Border point: a point that belongs to the border of the density based cluster;
- Noise point: a point that is neither a core point nor border point.

```
main_dbscan:
Read the simulation configuration file
Derive input parameters used in simulation
Read the kMC simulation output VTK file passed as input to DBSCAN
// Apply the DBSCAN clustering algorithm to the data read from VTK file
nClusters = DBSCAN(atoms, nAtoms,eps,minPts)
if (nClusters > 0) then // DBSCAN found clusters
    Create an array of Arraylist necessary to store all clusters' atoms → cluster
    Organize the array of clustered points (atoms) in an array of ArrayList's (cluster),
        with the atoms of each cluster stored in a different position of the array
    Create an array of Arraylist's necessary to store all atoms' neighborhood → N
    Compute the neighborhood (inside a cluster) of each atom and store it in N
    Merge the clusters that are split in several parts in a single spatial region per cluster
    // Execute a small clusters analysis
    countNumberSmallClusters (cluster, nClusters)
    // Execute a stable precipitates analysis
    Remove the clusters that have a size smaller than a given threshold
      if (nPrecipitates > 0) then // There are stable precipitates
        Compute and store in an array the size of all precipitates → sizeCluster
        Print the size of all precipitates

        Compute and store in an array the radius of all precipitates → radiusCluster
        Print the radius of all precipitates

        Compute the average size among all precipitates → avgSize
        Print "Average size among all precipitates is " avgSize

        Compute the average radius among all precipitates → avgRadius
        Print "Average radius among all precipitates is " avgRadius

        Compute the percentage of Sc atoms in (Al3Sc) precipitates → pAl3Sc
        Compute the percentage of Sc atoms in Al solid solution → pAl3Sc

        Print "Percentage of Sc atoms in Al solid solution is " pAl3Sc
        Print "Percentage of Sc atoms in precipitates is " pAl3Sc

        Write a VTK file with result of clustering: (x,y,z,clusterId) for each atom

        endIf  // (end of) There are stable precipitates

      endIf  // (end of) DBSCAN found clusters
```

Figure 28 – Main function for the application of DBSCAN in pseudocode (part 1).

```
    Write (to file) a report about clustering analysis: some data about simulation,
        input VTK file, output VTK file, configuration file, number of (stable)
        precipitates identified, precipitates' size, average precipitates' size,
        precipitates' radius, average precipitates' radius, number of small clusters
        with the same size

    Free memory space

    end main_dbscan
```

Figure 29 – Main function for the application of DBSCAN in pseudocode (part 2).

Figure 28 and Figure 29 present the pseudocode for the ***main*** function that applies the DBSCAN clustering algorithm to the VTK files saved by the kMC simulation.

Some notes about the clustering analysis with DBSCAN. To store the atoms of each cluster it was used a data structure that varies dynamically, because the clusters are of variable and unknown size. The used data structure was inspired by the Java *ArrayList* class. After applying DBSCAN, the clusters that are split in several parts are merged in a single spatial region per cluster. This is required because we use PBC and aims to improve the 3D visualization of clusters.

To allow the comparison between kMC results and CNT, it was necessary to execute a small clusters analysis, beyond the stable precipitates analysis. Beyond the VTK file with precipitates, the analysis carried out by DBSCAN produces other results, such as, the size and radius of the precipitates, the average size and radius among all precipitates, the percentage of Sc atoms in precipitates and in Al solid solution, and the number of small clusters with the same size.

The clustering analysis is mainly implemented by the *DBSCAN* (Figure 30) and the *ExpandCluster* (Figure 31) functions. We will not go into more detail about the implementation of these functions, since they follow the structure proposed by [Ester1996].

```
DBSCAN (atoms[], nAtoms, eps, minPts) ≡

    Create an array to store a flag that indicates if an atom was yet visited or not → visited
    Initialize the entire array visited to FALSE

    Create an array of Arraylist necessary to store all atoms' neighborhood → N
    Compute the neighborhood of each atom on a radius of value eps and store it in N

    cid = 0    // current cluster ID
    pid = 0    // atom position on global array of atoms

    while (pid < nAtoms) do // cycle over all atoms

        if (visited[pid] = FALSE) then   // atom 'pid' was not yet visited
            visited[pid] = TRUE          // mark atom 'pid' as visited
            sizeN = size(N[pid])

            if (sizeN<minPts) then
              atoms[pid].classId = NOISE
            else
              resBool = ExpandCluster (atoms, nAtoms, visited, N, pid, cid, eps, minPts)
              if (resBool = TRUE) then
                    Increment cid
              endIf
            endIf
        endIf
        Increment pid
    endWhile

    Free memory space
    return cid
end DBSCAN
```

Figure 30 – *DBSCAN* clustering algorithm in pseudocode.

```
ExpandCluster (atoms[], nAtoms, visited[], N[], pid, cid, eps, minPts) ≡
    sizeN              = size(N[pid])
    sizeUnclustered    = sizeN
    i = 0
    while (i < sizeN) do // Count the unclustered neighbors of atom 'pid'
        nid  = position, on global array atoms, where element i of ArrayList N[pid] is stored
        if((atoms[nid].classId !=NOISE) AND (atoms[nid].classId !=UNCLASSIFIED)) then
            Decrement sizeUnclustered
        endIf
        Increment i
    endWhile

    // There are NOT enough unclustered neighbors to make 'pid' a core atom of a new cluster
    if (sizeUnclustered < minPts) then
        atoms[pid].classId = NOISE // Mark atom 'pid' as NOISE
        return FALSE

    // There are enough unclustered neighbors to make 'pid' a core atom of a new cluster
    else
        atoms[pid].classId = cid  // Add atom 'pid' to cluster 'cid'
        i = 0
        while (i < sizeN) do
            nid = element i of ArrayList N[pid]
            if (visited[nid] = FALSE) then // atom 'nid' is not yet visited
                visited[nid] = TRUE
                sizeNN = size(N[nid])
                if (sizeNN  >= minPts) then
                    nn = 0
                    for (nn < sizeNN) do
                        nnid  = element nn of ArrayList N[nid]
                        Add element 'nnid' to N[pid]
                        Increment sizeN
                        Increment nn
                    endWhile
                endIf
            endIf
            // Atom 'nid' is not yet member of any cluster
            if ((atoms[nid].classId=NOISE) OR (atoms[nid].classId=UNCLASSIFIED)) then
                atoms[nid].classId = cid
            endIf
            Increment i
        endWhile
    endIf  // (end of) There are enough unclustered neighbors ...
    return TRUE
end ExpandCluster
```

Figure 31 – Function *ExpandCluster* used by DBSCAN in pseudocode.

The implementation of the clustering analysis, which was applied to the snapshots saved during the kMC simulations, is composed of the files listed in Table 17. The complete C code is included in Appendix I. To compile the source files we used the following command:

gcc -O3 activationEnergy.c config.c utils.c fcclattice.c neighbors.c writeFiles.c readFiles.c ArrayList.c ArrayUtil.c dbscan1.c dbscan2.c clustering1.c main2.c -lm -o main2_O3

Table 17 – Source and header files for the C implementation of DBSCAN.

activationEnergy.c	ArrayList.c
ArrayList.h	ArrayUtil.c
ArrayUtil.h	clustering1.c
config.c	dbscan1.c
dbscan2.c	fcclattice.c
main_includes.h	main2.c
neighbors.c	readFiles.c
utils.c	writeFiles.c

Appendix D contains an example of the report produced after applying DBSCAN to the Simulation I final snapshot configuration (a VTK file). This report includes simulation parameters such as the radius value used to define an atom's neighborhood, the minimum number of neighbors to consider an atom as core and the minimum number of atoms to consider a cluster as a valid cluster. It also reports produced information such as the percentage of Sc atoms in the aluminum solid solution, the percentage of scandium atoms in Al_3Sc precipitates, the number of identified precipitates, the average size among all precipitates (in atoms) and the average radius among all precipitates (in Angstrom). Additionally, the report tables information relative to each precipitate individually: the ID, the size (in atoms) and the radius (in Angstrom).

Figure 32 illustrates a configuration with various Al_3SC precipitates identified by DBSCAN. To make the 3D visualization easier, only scandium atoms are displayed.

4.9 Classical nucleation theory

To validate the results obtained with the kMC simulations we may appeal to the classical nucleation theory (CNT).

4.9.1 Steady state nucleation rate and cluster concentration

CNT assumes that the precipitates evolution is stationary, with a size below a critical size n^*_{sc}. This means that CNT does not apply to precipitates with a size larger

than the critical size. The probability to find a precipitate with n_{Sc} Sc atoms, with $n_{Sc} \leq n^*_{Sc}$, is given by Equation 59 [Clouet2004a].

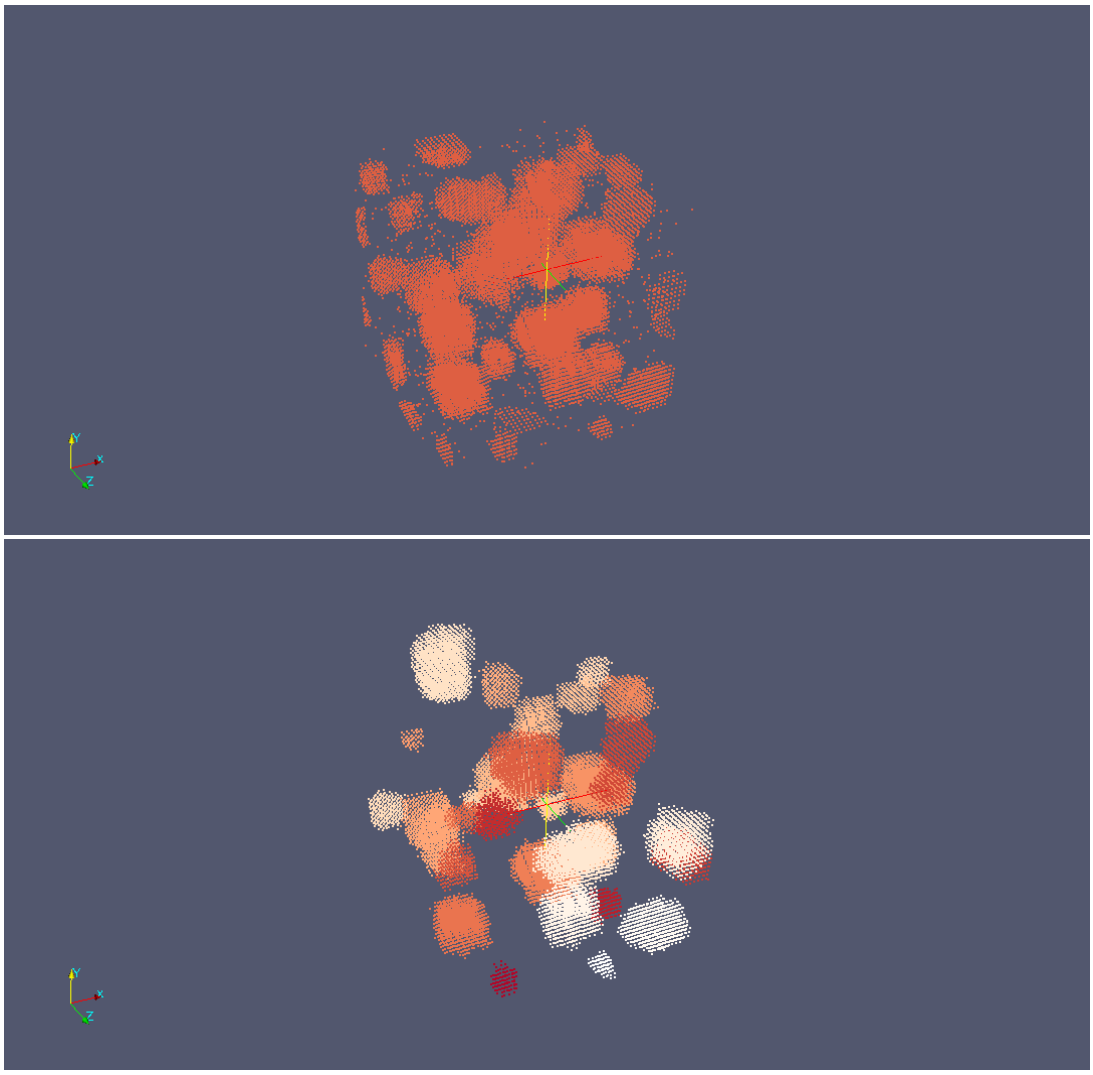


Figure 32 – Clustering of scandium atoms with DBSCAN. Top illustration shows all the scandium atoms, before application of DBSCAN. Bottom illustration shows the clustering of scandium atoms after applying DBSCAN.

$$Cn_{Sc} = exp\left(-\frac{\Delta Gn_{Sc}\left(X^0_{Sc}\right)}{kT}\right) \qquad \text{(Equation 59)}$$

where ΔGn_{Sc} is the formation free energy of the precipitate. Cn_{Sc} is also called cluster concentration. In terms of kMC, the cluster concentration is computed by Equation 60.

$$Cn_{Sc} = \#clusters * \frac{n_{Sc}}{N_{SC}} \qquad \text{(Equation 60)}$$

where N_{SC} is the number of scandium atoms in the lattice.

The main inputs necessary to compute the formation free energy are:

- The scandium equilibrium concentration, or solubility limit of scandium in aluminum, obtained by a third order Low Temperature Expansion (LTE) method is given by [Clouet2007]:

$$X_{Sc}^{eq} = exp\left[\frac{-6\omega_{AlSc}^{(2)}}{kT}\right] + 6 * exp\left[\frac{-10\omega_{AlSc}^{(2)}}{kT}\right] - 16 * exp\left[\frac{-12\omega_{AlSc}^{(2)}}{kT}\right] \quad \text{(Equation 61)}$$

  where $\omega_{AlSc}^{(2)}$ is the second nearest neighbors interaction between an Al and an Sc atoms.

- The scandium impurity diffusion coefficient is [Clouet2004a]:

$$D_{Sc} = 5.31 * 10^{-4} * exp(-1,79eV/kT) \quad \text{(Equation 62)}$$

All CNT input parameters are summarized in Table 18.

Table 18 – Classical Nucleation Theory input parameters.

Name	Value	Description
k	8,61733E-5 *eV*	Boltzman constant
a	4,05E-10	Lattice constant
T		Temperature
Ns		Number of nucleation sites
n_{Sc}^{*}	13	Critical size (minimum number of Sc atoms in a stable precipitate)
n_{Sc} ($n=4*n_{Sc}$)		Number of Sc (total) atoms in the precipitate
X_{Sc}^{0}		Nominal concentration
$\overline{\sigma}_{nSc}$		Interface free energy
X_{Sc}^{eq}		Sc equilibrium concentration or solubility limit of Sc in Al
D_{Sc}		Sc impurity diffusion coefficient

4.9.2 Precipitates formation free energy

Formation free energy $\Delta Gn_{Sc}(X^0_{Sc})$ is the energy necessary to create a $L1_2$ precipitate with n_{Sc} solute atoms (or $n=4*n_{Sc}$ total number of atoms) inside a solid solution with X^0_{Sc} nominal concentration of scandium atoms. Under the capillary approximation this energy includes 2 terms: (i) the volume contribution, which is a nucleation free energy $\Delta G^{nuc}(X^0_{Sc})$, and (ii) the surface contribution, which is the energy

needed to create an interface between the solid solution and the $L1_2$ precipitate. The interface contribution is a function of $a^2\bar{\sigma}$. The formation free energy can be expressed as [Clouet2005a]:

$$\Delta \boldsymbol{Gn_{Sc}}\left(\boldsymbol{X_{Sc}^0}\right) = \boldsymbol{4} * \boldsymbol{n_{Sc}} * \Delta \boldsymbol{G^{nuc}}\left(\boldsymbol{X_{Sc}^0}\right) + (\boldsymbol{36} * \boldsymbol{\pi})^{1/3} * \boldsymbol{n_{Sc}}^{2/3} * \boldsymbol{a^2} * \boldsymbol{\bar{\sigma}_{nSc}} \qquad \text{(Equation 63)}$$

4.9.3 Nucleation free energy

The nucleation free energy is expressed in terms of chemical potentials, but it can be simplified by an ideal model of the solid solution, corresponding to the Bragg-Williams approximation. Within this approximation, the ideal nucleation free energy expression is [Clouet2004a]:

$$\Delta \boldsymbol{G_{ideal}^{nuc}}\left(\boldsymbol{X_{Sc}^0}\right) = \frac{\boldsymbol{3}}{\boldsymbol{4}} \ln\left(\frac{\boldsymbol{1 - X_{Sc}^{eq}}}{\boldsymbol{1 - X_{Sc}^0}}\right) + \frac{\boldsymbol{1}}{\boldsymbol{4}} \ln\left(\frac{\boldsymbol{X_{Sc}^{eq}}}{\boldsymbol{X_{Sc}^0}}\right) \qquad \text{(Equation 64)}$$

The Bragg-Williams (BW) approximation only produces acceptable results with low supersaturated solid solutions (low nominal concentration). The poor behavior of BW approximation, in calculating the clusters size (Cn_{Sc}) and the steady-state nucleation rate (J^{st}), can be observed in figure 7 of [Clouet2004a]. With high supersaturations, nucleation free energy must be calculated using the Cluster Variation Method (CVM) or a direct calculation. But CVM raises a computational problem: it is not expressed by an analytical form. A possible solution, which produces results with the same quality as CVM and has an analytical expression, is the Low Temperature Expansion (LTE) method [Clouet2007] [Clouet2005a]. Considering only the third order of the LTE, the nucleation free energy assumes the following analytic expression [Clouet2007]:

$$\Delta \boldsymbol{G^{nuc}}\left(\boldsymbol{X_{Sc}^0}\right) = \boldsymbol{kT} * \left[\boldsymbol{q}\left(\boldsymbol{X_{Sc}^0}\right) - \boldsymbol{q}\left(\boldsymbol{X_{Sc}^{eq}}\right)\right] + \boldsymbol{3kT} * \exp\left(\frac{\boldsymbol{2} * \boldsymbol{\omega_{AlSc}^{(2)}}}{\boldsymbol{kT}}\right) * \left[\boldsymbol{q}\left(\boldsymbol{X_{Sc}^0}\right)^{\boldsymbol{2}} - \boldsymbol{q}\left(\boldsymbol{X_{Sc}^{eq}}\right)^{\boldsymbol{2}}\right] - \frac{\boldsymbol{1}}{\boldsymbol{4}}\boldsymbol{kT} * \left\{\ln\left[\boldsymbol{q}\left(\boldsymbol{X_{Sc}^0}\right)\right] - \ln\left[\boldsymbol{q}\left(\boldsymbol{X_{Sc}^{eq}}\right)\right]\right\} \qquad \text{(Equation 65)}$$

where $q(x)$ is the following function

$$\boldsymbol{q}(\boldsymbol{x}) = \frac{\boldsymbol{2x}}{\boldsymbol{1} + \sqrt{\boldsymbol{1} + \boldsymbol{4x} * \left[\boldsymbol{6} * \boldsymbol{exp}\left(\frac{\boldsymbol{2} * \boldsymbol{\omega_{AlSc}^{(2)}}}{\boldsymbol{kT}}\right) - \boldsymbol{19}\right]}} \qquad \text{(Equation 66)}$$

The expression of $\omega_{AlSc}^{(2)}$ is obtained from the solubility limit X_{Sc}^{eq} [Clouet2007]:

$$\boldsymbol{\omega_{AlSc}^{(2)} = -\frac{1}{6}kT * \ln\left(X_{Sc}^{eq}\right) + kT * \left(X_{Sc}^{eq\frac{2}{3}} - \frac{8}{3} * X_{Sc}^{eq}\right)} \qquad \textbf{(Equation 67)}$$

$\omega_{AlSc}^{(2)}$ is the second order energy and can also be expressed as a function of the second nearest-neighbor pair effective energies ($\varepsilon_{AlSc}^{(2)}$, $\varepsilon_{AlAl}^{(2)}$, $\varepsilon_{ScSc}^{(2)}$):

$$\boldsymbol{\omega_{AlSc}^{(2)} = \varepsilon_{AlSc}^{(2)} - \frac{1}{2}\varepsilon_{AlAl}^{(2)} - \frac{1}{2}\varepsilon_{ScSc}^{(2)}} \qquad \textbf{(Equation 68)}$$

Considering that *AlAl* and *ScSc* terms are null, the expression used to compute $\omega_{AlSc}^{(2)}$ was:

$$\boldsymbol{\omega_{AlSc}^{(2)} = \varepsilon_{AlSc}^{(2)} = 0.113 - 0.334 * 10^{-4} * T} \qquad \textbf{(Equation 69)}$$

4.9.4 Interface free energy ($\bar{\sigma}$)

The average (or isotropic) interface free energy ($\bar{\sigma}$) can be computed from the interface free energies for [100], [110] and [111] directions: σ_{100}, σ_{110}, and σ_{111}. The isotropic energy is the interface energy of a spherical precipitate that has the same volume than the real faceted one. The procedure that computes the isotropic energy uses a Wulff construction [Clouet2004a].

Computing isotropic interface free energy with Wulff construction

Al_3Sc precipitates present facets in [100], [110] and [111] directions. It is known that in pure phases, the interface energies on these 3 directions are related by the following manner:

$$\boldsymbol{\sigma_{100} = \frac{1}{\sqrt{2}}\sigma_{110} = \frac{1}{\sqrt{3}}\sigma_{111}} \qquad \textbf{(Equation 70)}$$

The surfaces of precipitate facets, oriented on each of the mentioned directions, are given by:

$$\boldsymbol{\mathcal{A}_{100} = 4 * (\sigma_{100} - \sqrt{2} * \sigma_{110})^2 - 2 * (\sigma_{100} - 2\sqrt{2} * \sigma_{110} + \sqrt{3} * \sigma_{111})^2} \qquad \textbf{(Equation 71)}$$

$$\boldsymbol{\mathcal{A}_{110} = 2\sqrt{2} * (-2\sigma_{100} + \sqrt{2} * \sigma_{110})^2 * (\sqrt{2} * \sigma_{110} - \sqrt{3} * \sigma_{111})^2} \qquad \textbf{(Equation 72)}$$

$$\mathcal{A}_{111} = \frac{3\sqrt{3}}{2} * (-\sigma_{100}^2 - 2 * \sigma_{110}^2 + \sigma_{111}^2)^2 + \frac{3}{2} * \sigma_{100} * (4\sqrt{6} * \sigma_{100} - 6 * \sigma_{111}) \quad \text{(Equation 73)}$$

For a spherical precipitate, with the same volume than the faceted one, its isotropic interface free energy is:

$$\bar{\sigma}_{nSc} = \sqrt[3]{\frac{1}{4\pi} * (6 * \sigma_{100} * \mathcal{A}_{100} + 12 * \sigma_{110} * \mathcal{A}_{110} + 8 * \sigma_{111} * \mathcal{A}_1} \quad \text{(Equation 74)}$$

Figure 33 presents the results of applying the previous formulation of the Wulff construction to obtain the isotropic interface free energy.

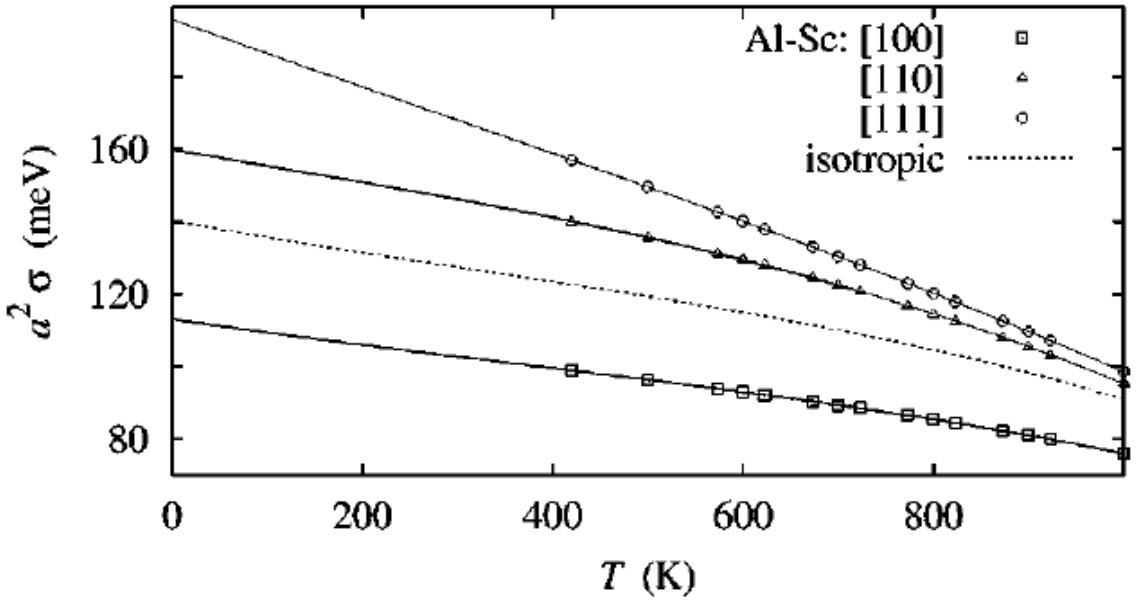


Figure 33 – Temperature versus the solid solution/Al_3Sc interface directions and the isotropic free energy.

The third order LTE provides an analytic expression to compute the isotropic interface free energy [Clouet2007]:

$$a^2\bar{\sigma}_{nSc} = (\frac{6}{\pi})^{1/3} * [\omega_{AlSc}^{(2)} - 2kT * \exp(\frac{-4\omega_{AlSc}^{(2)}}{kT}) - kT * \exp(\frac{-6\omega_{AlSc}^{(2)}}{kT})] \quad \text{(Equation 75)}$$

where *a* is the lattice constant.

4.9.5 Steady-state nucleation rate

Knowing the nucleation free energy $\Delta G^{nuc}(X_{Sc}^0)$, the isotropic interface free energy $a^2\bar{\sigma}_{nSc}$ and the impurity diffusion coefficient D_{Sc} , LTE allows us to compute the steady-state nucleation rate J^{st} for any combination of nominal concentration and temperature [Clouet2007]:

$$J^{st}(n_{Sc}, T) = -16N_s * \frac{\Delta G^{nuc}(X_{Sc}^0)}{\sqrt{kTa^2\bar{\sigma}_{nSc}}} * \frac{D_{Sc}}{a^2} * X_{Sc}^0 * exp\left(-\frac{\pi * (a^2\bar{\sigma}_{nSc})^3}{3kT * [\Delta G^{nuc}(X_{Sc}^0)]^2}\right) \quad \text{(Equation 76)}$$

where N_s is the number of lattice sites. To obtain a normalized J^{st}, we have to divide the value calculated with equation (Equation 76) by the simulated volume: $\frac{N_S a^3}{4}$.

The J^{st} defined by CNT is equivalent, in kMC, to the initial slope of the graphic that represents the number of precipitates as a function of the simulated time.

The implementation of the classical nucleation theory included the files held in

Table 19. The compilation command was:

gcc -O3 cnt_config.c cnt_functions1.c cnt_main.c -lm -o cnt_O3

Table 19 – Source and header files for the C implementation of CNT.

cnt_includes.h	cnt_main.c
cnt_config.c	cnt_functions1.c

Figure 34 presents the pseudocode for the main steps of the classical nucleation theory implementation in C language. This implementation followed the CNT equations introduced before. The complete C code is included in Appendix I.

4.10 Simulation configurations visualization

To permit the visualization of the lattice configurations generated by the kMC simulation and by the clustering analysis, these configurations are saved to files in a format that can be read and rendered by an available tool. The developed code allows us to save data in one of the following formats: **PDB**, **XYZ** and **VTK**.

main_CNT:

Read the simulation configuration file

Open output file

Allocate memory for the matrix that will store *deltaG_nuc[][]*

Allocate memory for the array that will store *D_Sc[]* values

Allocate memory for the array that will store *a2sigma[]* values

Allocate memory for the matrix that will store *Jst[][]* values

Allocate memory for the matrix that will store *C_nSc_X0 [][]* values

Compute data to draw the nucleation free energy graph → *deltaG_nuc(X0_Sc, T)*

Write the nucleation free energy data (*deltaG_nuc[][]*) to file

Compute data to draw the Sc impurity diffusion coefficient graph → *D_Sc(T)*

Write the Sc impurity diffusion coefficient data (*D_Sc[]*) to file

Compute data to draw the isotropic free energy graph → *a2sigma(T)*

Write the isotropic free energy data (*a2sigma[]*) to file

Compute data to draw the steady state nucleation rate graph → *Jst(X0_Sc,T)*

Write the steady state nucleation rate data (*Jst[][]*) to file

Compute data to draw the precipitate concentration graph for several X0_Sc: *C_nSc(X0_Sc,nSc)*

Write the precipitate concentration data (*C_nSc[][]*) to file

Free allocated memory and close open file

end main_CNT

Figure 34 – Pseudocode for the classical nucleation theory implementation.

4.10.1 Supported formats

PDB stands for Protein Data Bank. The Protein Data Bank can be understood as a worldwide repository of 3D structure data of biological molecules, namely proteins, nucleic acids as also for crystallographic structures (atoms, ions molecules). The PDB format is a text format that describes the position of atoms in a molecular structure. The coordinates are read from the ATOM records. Since PDB is relative complex format, whose must functionalities are nor relevant to the present work, it was implemented in a minimal version.

The **XYZ** file format is a generic and simple text format that can be used to write a set of 3D coordinates. Most visualization tools support this format.

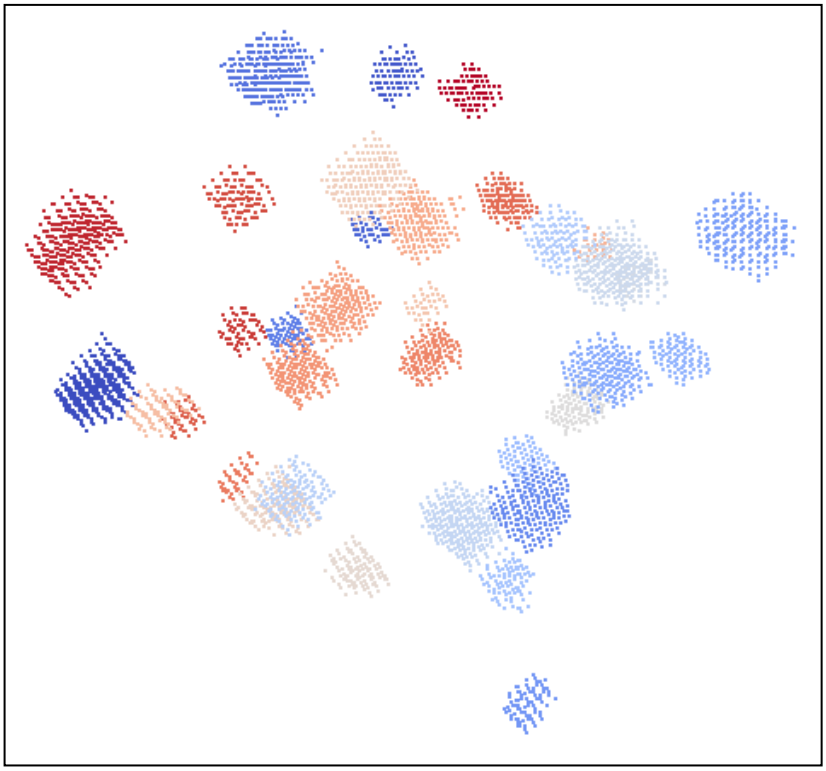


Figure 35 – An example of the visualization of a VTK file in ParaView.

VTK stands for Visualization Toolkit. The Visualization Toolkit is a software package that can be used to develop graphic-oriented applications. VTK also defines formats to store multidimensional data. The VTK file format is a text format that describes 3D graphic elements, such as points, vertices, lines and polygons. Among the three mentioned formats, our coding effort was centered in VTK format.

All these data formats can be visualized with the ParaView[3] tool, which is an open-source application adequate to the visualization and analysis of multidimensional data (Figure 35).

Figure 36 presents the pseudocode for the function that writes a lattice configuration to a VTK file. The configuration is specified by the coordinates and an attribute (atoms' type, atoms' precipitate identifier) for each atom of the selected type(s).

[3] www.paraview.org

```
writeVTKfileSimple:

Define the output file name based on the configuration parameter, the type of atoms
        (Al, Sc, Vacancy) to write to file and the number of the snapshot
Open the output file

Write to file the VTK format header

// Write the atoms section (atoms' coordinates) to file
n = 0
while (n < nAtoms) do
    if ( type of atom[n] was selected to be written to file) then
        Write the coordinates X, Y and Z of atom[n] to file
    endIf
    Increment n
endWhile

// Write the vertices section (atoms' order in VTK file) to file
n = 0
while (n < nSelectedAtoms) do
    Write a line with a pair "1 n" to file
    Increment n
endWhile

// Write the lookup table section (atoms' attribute) to file
Write the lookup table preamble to file
n = 0
while (n < nAtoms) do
    if ( type of atom[n] was selected to be written to file) then
        Write the type of atom[n] to file
    endIf
    Increment n
endWhile

Close the opened file

end writeVTKfileSimple
```

Figure 36 – Function that writes a lattice configuration to a VTK file, in pseudocode.

5. Results and Discussion

In this chapter, the results of the Monte Carlo simulations as well as the DBSCAN clustering analysis are presented and discussed. The simulations are organized according to the applied temperature. Simulations were conducted with 873K, 823K, 773K, 723K, 673K and 573K. The results focus on the time evolution of the number and size of the precipitates, as well as comparisons with the classical nucleation theory.

The simulations where run on the SeARCH cluster, which stands for "Services and Advanced Research Computing with HTC/HPC clusters". SeARCH has the vital goal of supporting R&D projects conducted at the University of Minho. Table 20 contains the technical specifications of the SeARCH cluster nodes where we run the Monte Carlo simulations.

Table 20 – Technical specifications of the SeARCH nodes used by the MC simulations.

Nodes	Processors	CPUs number	L2 Cache	Operating System
311–X nodes	Intel Xeon E5420	8	12 MB	Linux x86_64
201–X nodes	Intel Xeon 5130	4	4 MB	Linux x86_64
101–X nodes	Intel Xeon	4	2 MB	Linux x86_64

The results documented in this chapter include the number of precipitates, the precipitates mean size, the precipitates mean radius, the percentage of scandium atoms in the aluminum solution, the percentage of scandium atoms in precipitates, and the stable precipitates versus the number of lattice sites results. These results are presented in the form of graphics.

5.1 Series I of simulations

These results presented in this section refer to a series of simulations that underwent the conditions mentioned in Table 21. A series of five simulations, each one with a different Scandium percentage, were run with the same lattice size (50x50x50 unitary cells), the same temperature (873.15 K) and the same number of Monte Carlo steps ($5x10^{11}$). In each MC simulation 10 snapshots of the system configuration were saved to a VTK file. Each one of the 10 snapshots was analyzed with the DBSCAN algorithm to identify Al3Sc precipitates. DBSCAN analysis was conducted with the following parameters: *eps*=4.10, *minPts*=3 and *minClusterSize*=13, which means that a cluster of atoms is only considered a stable precipitate if it contains at least 13 scandium atoms.

Table 21 – Simulation I parameters.

Temperature [K]	873.15
Lattice size [FCC cells]	50x50x50
MCS	$5x10^{11}$

Sc %	Number Al atoms	Number Sc atoms
1	494999	5000
2	489999	10000
3	484999	15000
4	479999	20000
5	474999	25000

The total and partial (snapshots) simulated times are documented in Table 22. The simulated time evolution revealed steady with no compromising values. It is possible to observe that as the percentage of scandium increases the overall simulation time also increases.

Table 22 – Simulation I simulated time.

Simulated Time [s]					
	1% Sc	**2% Sc**	**3% Sc**	**4% Sc**	**5% Sc**
Snapshot I	5.326E-04	3.395E-03	4.680E-01	4.190E-02	1.600E+00
Snapshot II	1.049E-03	5.886E-03	1.372E+00	7.813E-02	2.987E+00
Snapshot III	1.549E-03	8.152E-03	1.791E+00	1.136E-01	4.198E+00
Snapshot IV	2.046E-03	1.034E-03	2.203E+00	1.463E-01	5.321E+00
Snapshot V	2.540E-03	1.243E-03	2.600E+00	1.782E-01	3.901E+00
Snapshot VI	3.028E-03	1.445E-02	3.015E+00	2.091E-01	1.398E+01
Snapshot VII	3.510E-03	1.632E-02	3.432E+00	2.714E-01	2.368E+01
Snapshot VIII	3.992E-03	1.822E-02	3.836E+00	3.014E-01	3.334E+01
Snapshot IX	4.471E-03	2.005E-02	4.232E+00	3.308E-01	4.300E+01
Snapshot X	4.945E-03	2.187E-02	4.587E+00	3.616E-01	5.239E+01
Total Time	4.945E-03	2.187E-02	4.587E+00	3.616E-01	5.239E+01

The computation time of the different simulations is presented in Table 23. A rough average of all simulations indicates that for the 50x50x50 lattice the simulation will take 12 days. We can also conclude that the percentage of scandium does not influence the computation time. This explained by the fact that simulation has only one vacancy and the main computation effort is not dependent on scandium percentage. The DBSCAN analysis time, which is very much lower than the simulation duration, increases with the scandium percentage.

Table 23 – Simulation I computation time.

Scandium percentage	Computation time
1 %	12 d: 1 h: 27 m: 48 s
2 %	12 d: 1 h: 55 m: 06 s
3 %	12 d: 1 h: 22 m: 58 s
4 %	12 d: 2 h: 01 m: 07 s
5 %	12 d: 2 h: 42 m: 41 s

Figure 37 illustrates the evolution of the number of precipitates over the duration of the simulations. The number of precipitates decreases over time. The explanation for the number of precipitates in Figure 37 to begin at a high value and then to decrease, is that the nucleation of precipitates is a very fast phenomenon. Accordingly, the initial rapid growth of precipitates is not observed with a sampling of 10 snapshots per $5x10^{11}$ MCS. As the number of precipitates decreases the average size of the precipitates increases, as demonstrated by Figure 39. Figure 39 displays the average size of precipitates, measured in terms of scandium atoms. Figure 40 illustrates the evolution of the scandium in the solid aluminum solid solution and in contrast, Figure 41 shows the evolution of scandium atoms in precipitate structures. The observation of these graphs allows one to say that the initial rapid increase in the number of precipitates decreases drastically the presence of scandium atoms in the aluminum solid solution and consequently increases the percentage of scandium in Al3Sc precipitates. Figure 42 shows the evolution of the number of stable precipitates normalized by the number of lattice sites. The figure demonstrates a rapid initial increase in the number of precipitates and then the tendency of decreasing as time evolves.

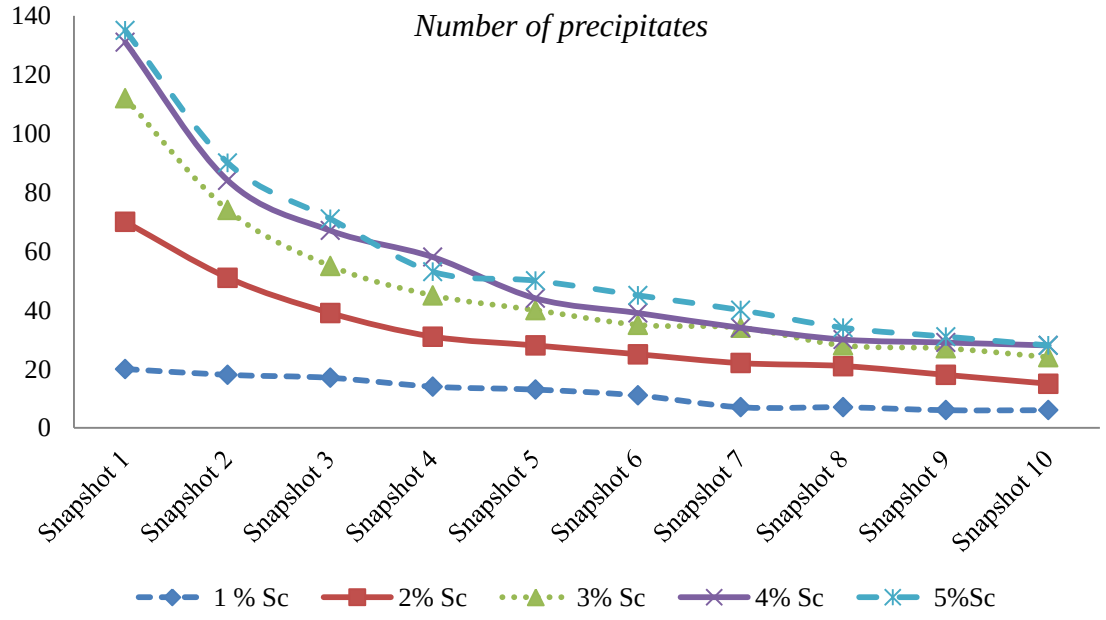


Figure 37 – Simulation I number of precipitates.

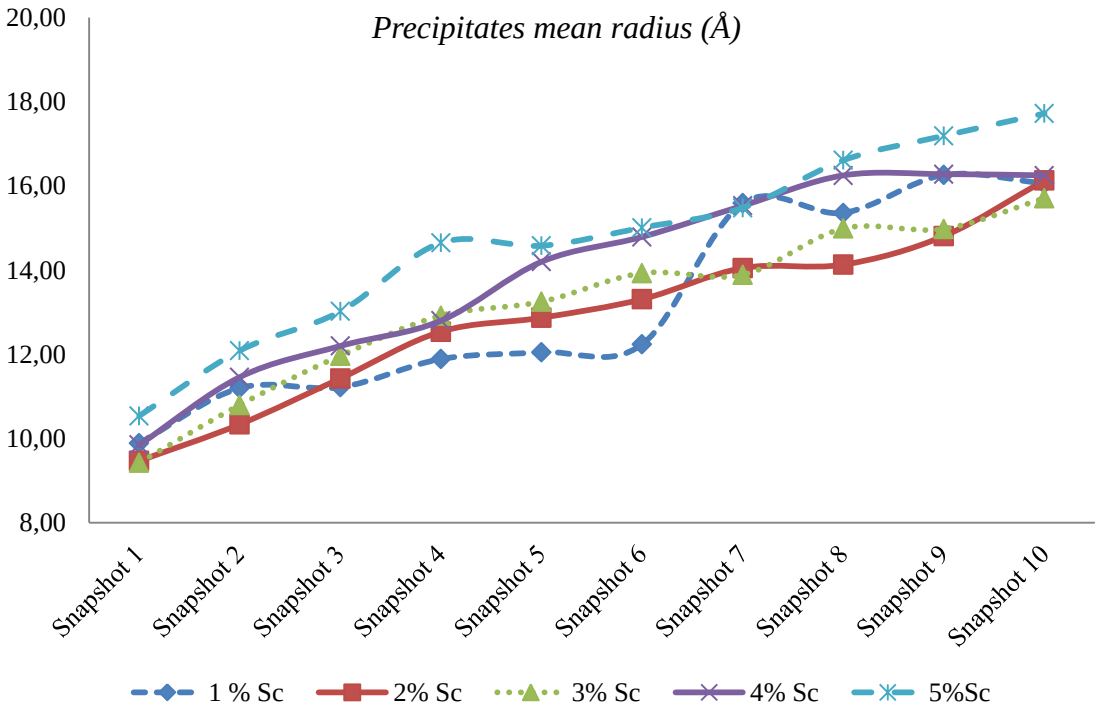


Figure 38 – Simulation I precipitates mean radius results.

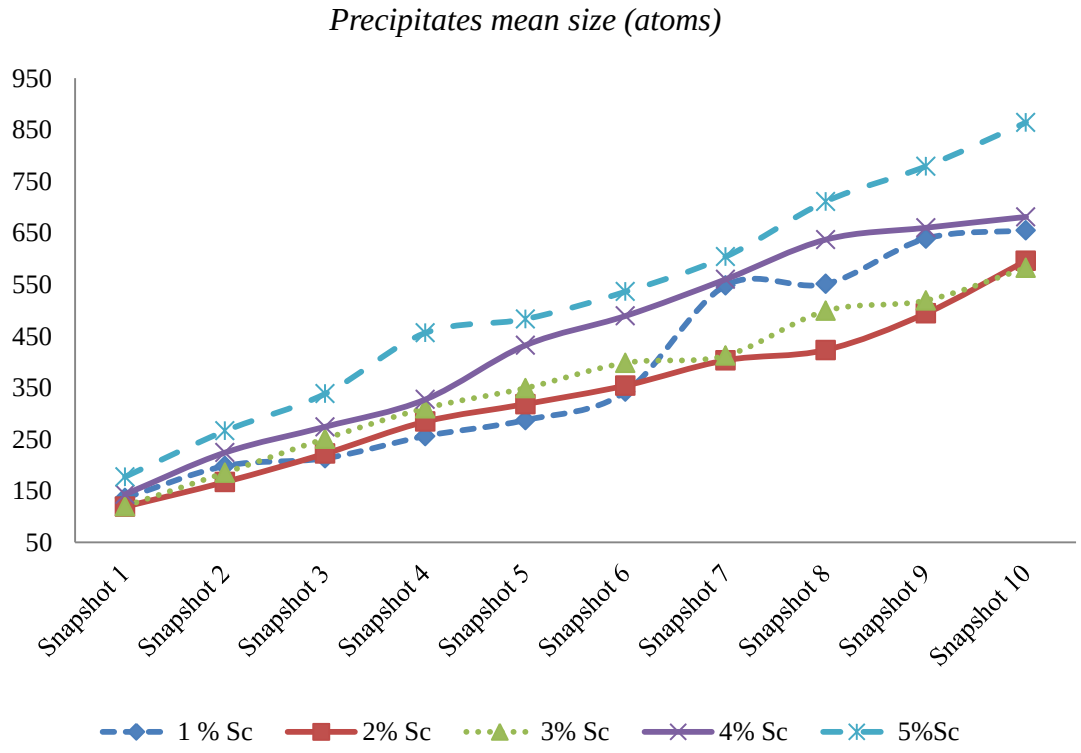


Figure 39 – Simulation I precipitates mean size.

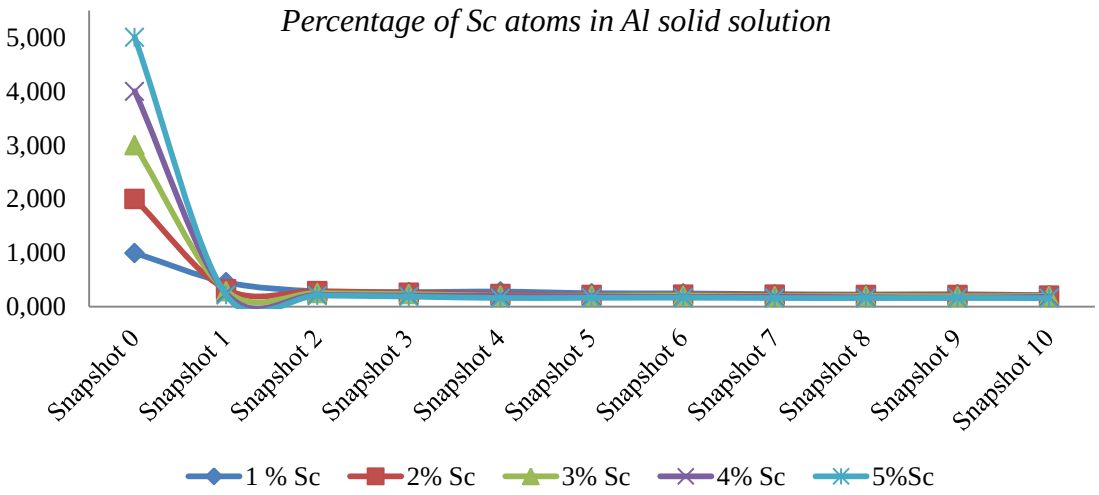


Figure 40 – Simulation I percentage of Sc in Al solid solution.

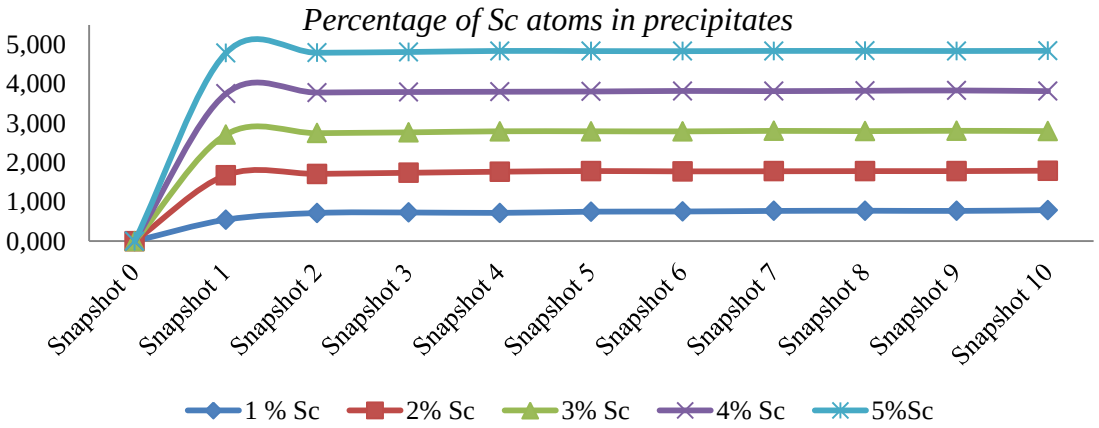


Figure 41 – Simulation I percentage of Sc atoms in precipitates.

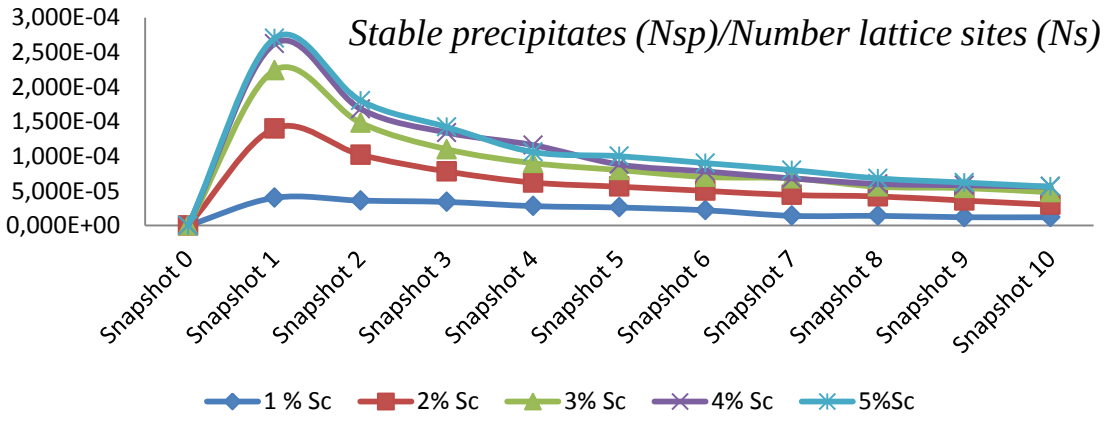


Figure 42 – Simulation I stable precipitates normalized by the number of lattice sites.

Figure 43 until Figure 47 present the last snapshot of each simulation of this series. Appendix E contains all the 10 snapshots, before and after applying the DBSCAN clustering algorithm, in the case of 1 % of scandium simulation. The sequence of images makes it clear that atoms group in Al_3Sc precipitates.

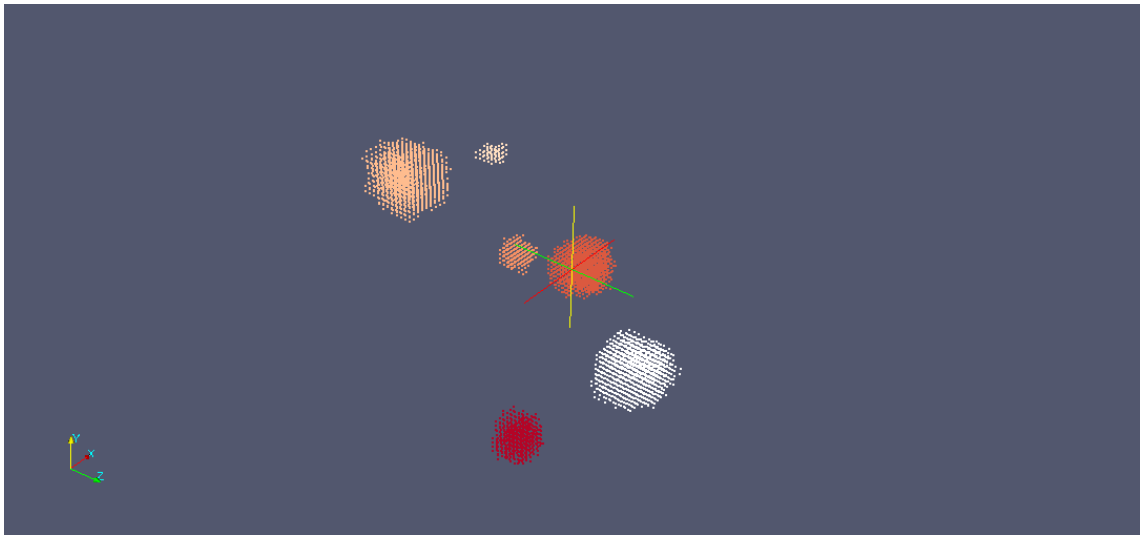


Figure 43 – Snapshot 10 of a 50x50x50 lattice with 1% Sc.

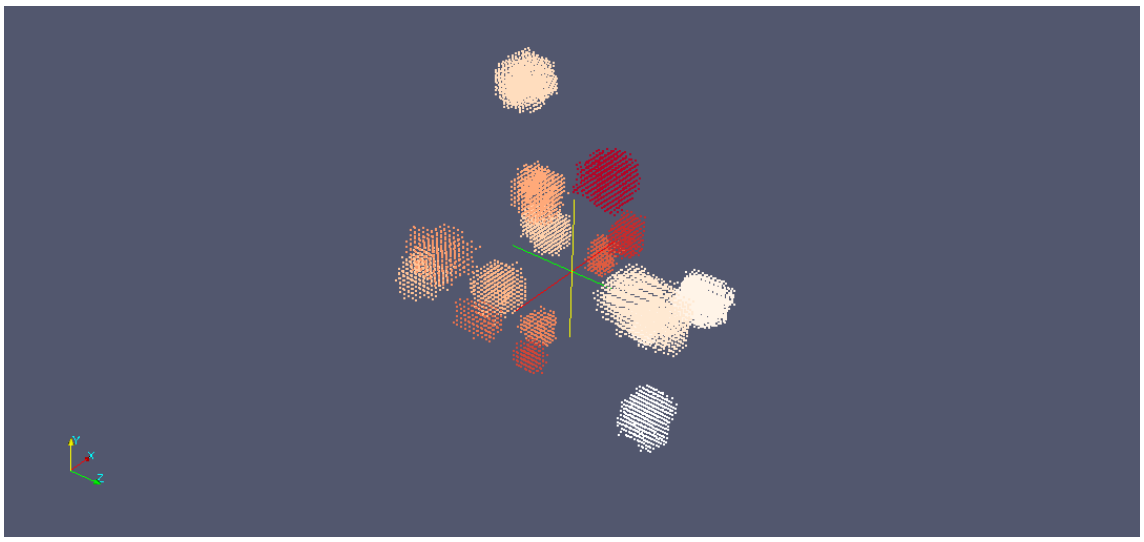


Figure 44 - Snapshot 10 of a 50x50x50 lattice with 2% Sc.

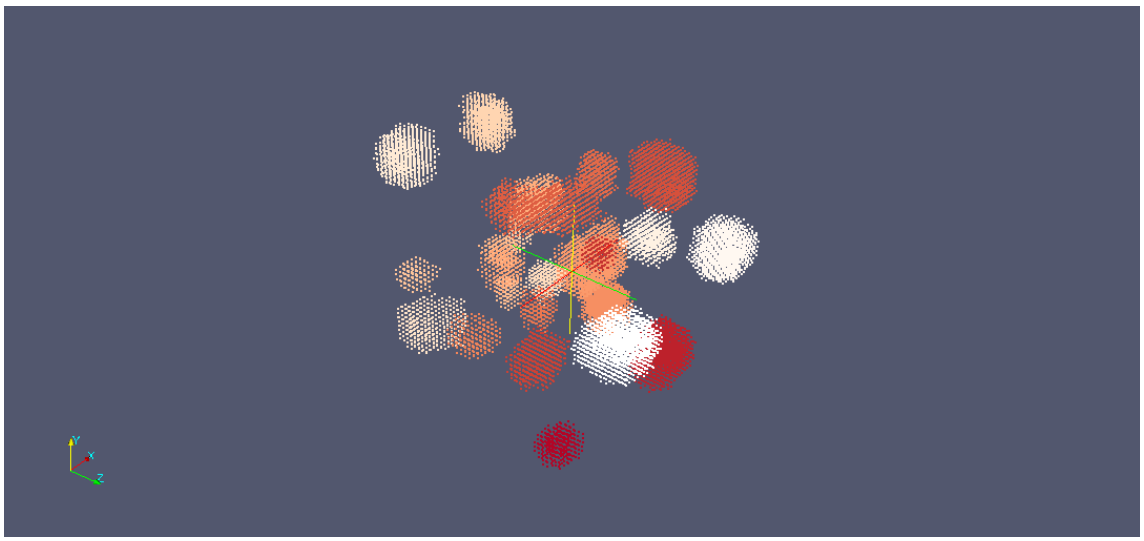


Figure 45 - Snapshot 10 of a 50x50x50 lattice with 3% Sc.

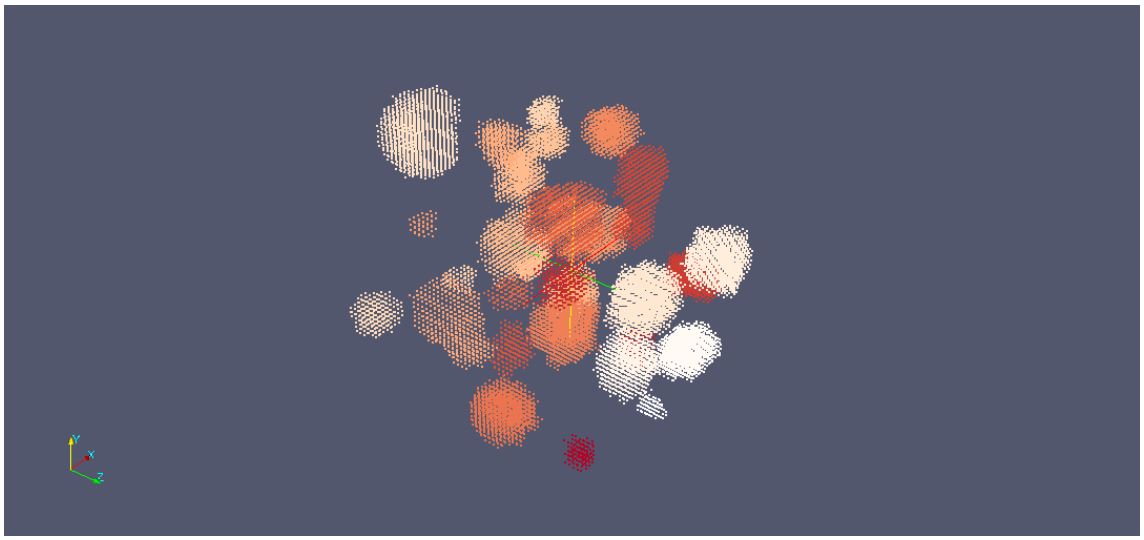


Figure 46 - Snapshot 10 of a 50x50x50 lattice with 4% Sc.

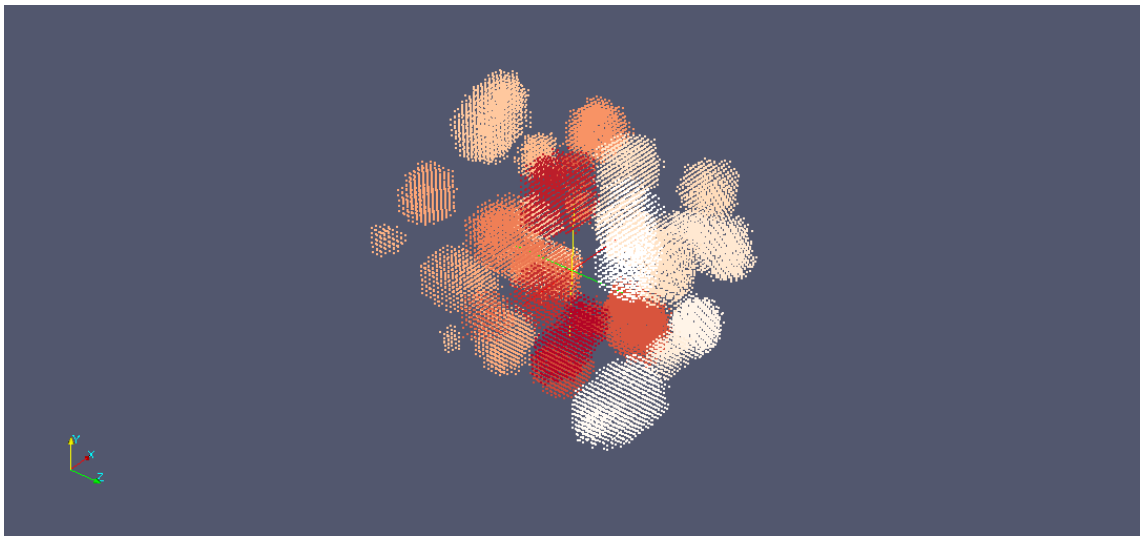


Figure 47 - Snapshot 10 of a 50x50x50 lattice with 5% Sc.

5.2 Series II of simulations

The series II of simulations underwent the conditions that are mentioned in Table 24. As series I, series II held five different simulations in which the scandium percentage ranged between 1 and 5 percent. The simulations were held with the same temperature as simulations from series I (873.15K), but with twice the number of Monte Carlo steps and a lattice with a size eight times larger. Consequently the computation time becomes twice as large.

Table 24 – Simulation II parameters.

Temperature [K]	873.15
Lattice size [FCC cells]	100x100x100
MCS	$1x10^{12}$

Sc %	Number Al atoms	Number Sc atoms
1	3959999	40000
2	3919999	80000
3	3879999	120000
4	3839999	160000
5	3799999	200000

Table 25 summarizes the simulated time for each of the ten snapshots and Table 26 the computation time for this sequence of simulations. Not surprisingly the total computation time is much higher than in series I and averages eighteen days.

Comparing the results with those obtained in series I of simulations it is possible to refer that the mean radius value is not drastically different between the two series. And yes the tendency of all the graphs agree between the two series of simulations.

Table 25 – Simulation II simulated time.

Simulated Time [s]					
	1% Sc	**2% Sc**	**3% Sc**	**4% Sc**	**5% Sc**
Snapshot I	1.105E-03	6.075E-02	2.790E-03	3.764E-03	2.865E-01
Snapshot II	2.171E-03	1.343E-01	5.076E-03	6.846E-03	5.948E-01
Snapshot III	3.183E-03	2.079E-01	7.197E-03	9.650E-03	8.942E-01
Snapshot IV	4.126E-03	3.844E-01	9.223E-03	1.232E-02	1.176E+00
Snapshot V	4.988E-03	3.543E-01	1.118E-02	1.487E-02	1.448E+00
Snapshot VI	5.767E-03	4.220E-01	1.307E-02	1.735E-02	1.716E+00
Snapshot VII	6.499E-03	4.868E-01	1.489E-02	1.984E-02	2.125E+00
Snapshot VIII	7.195E-03	5.555E-01	1.666E-02	2.218E-02	2.409E+00
Snapshot IX	7.867E-03	6.218E-01	1.841E-02	2.245E-02	2.644E+00
Snapshot X	8.519E-03	7.651E-01	2.016E-02	2.678E-02	2.872E+00
Total Time	8.519E-03	7.651E-01	2.016E-02	2.678E-02	2.872E+00

Table 26 - Simulation II computation time.

Scandium percentage	Time
1 %	18d:1h:54m:26s
2 %	18d:2h:44m:17s
3 %	18d:3h:58m:17s
4 %	18d:5h:32m:02s
5 %	18d:6h:43m:53s

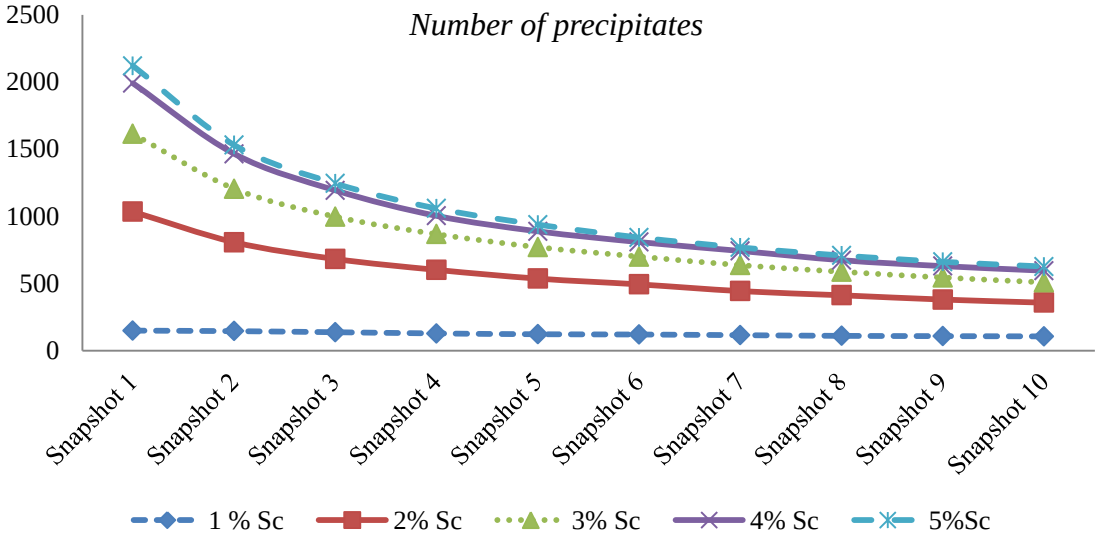


Figure 48 – Simulation II number of precipitates.

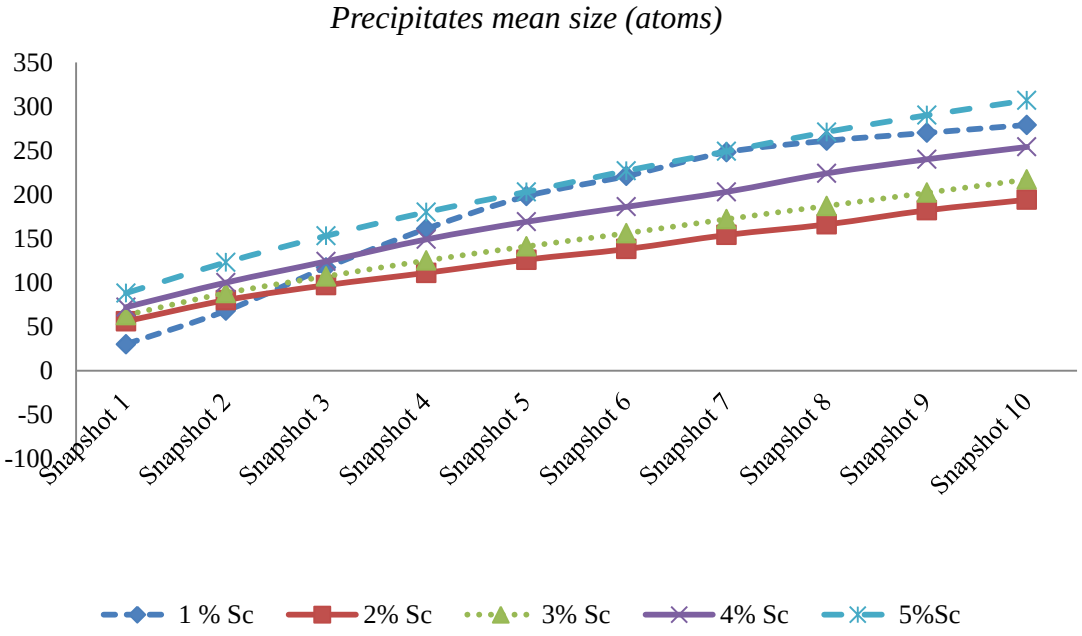


Figure 49 - Simulation II precipitates mean size.

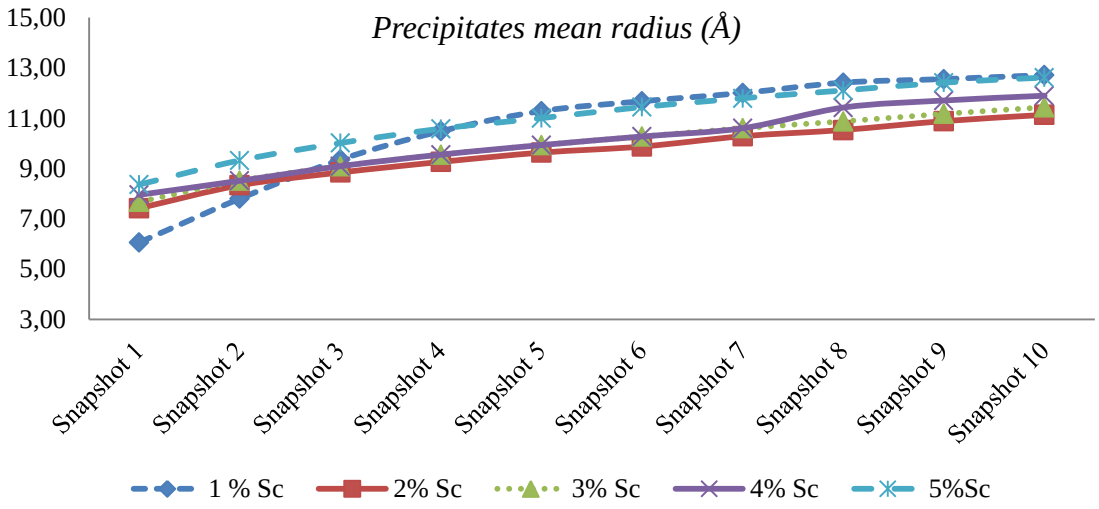


Figure 50 – Simulation II precipitates mean radius.

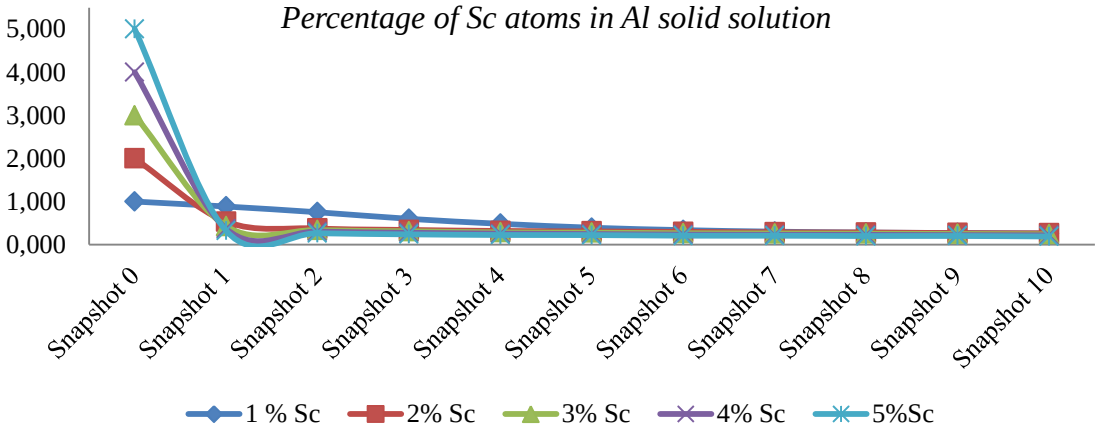


Figure 51 – Simulation II percentage of Sc atoms in Al solid solution.

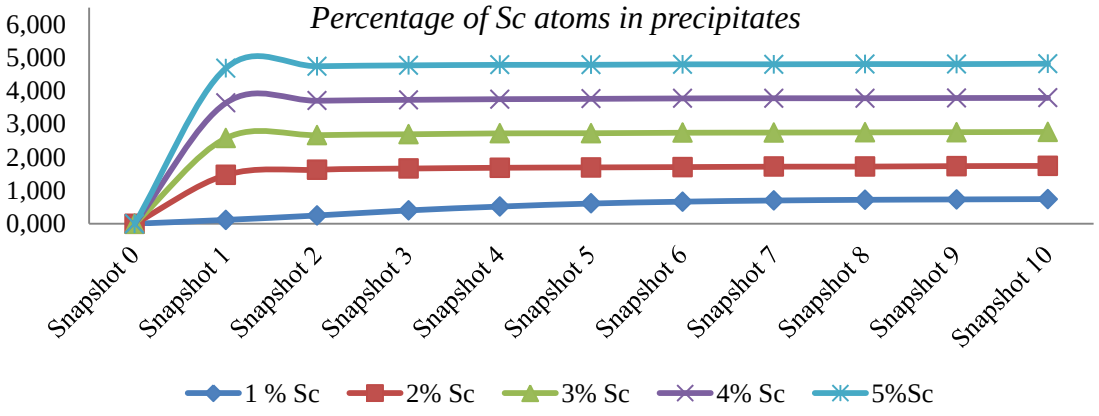


Figure 52 - Simulation II percentage of Sc atoms in precipitates.

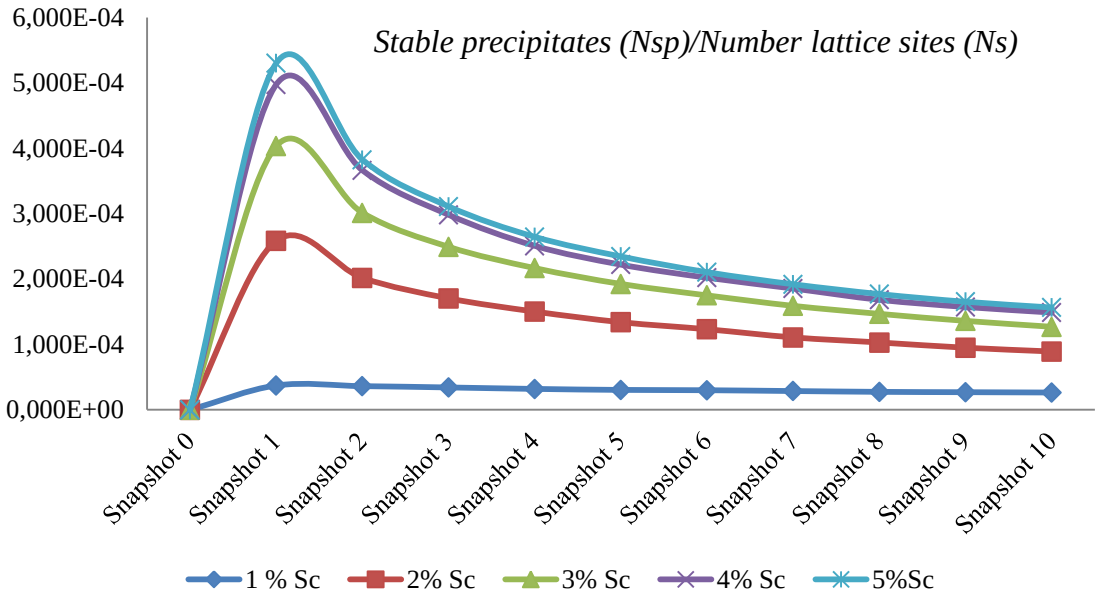


Figure 53 - Simulation II stable precipitates normalized by the number of lattice sites.

5.3 Series III of simulations

This series of simulation was run with a temperature of 823.15K. The applied parameters are listed in Table 27. The simulation underwent successfully although the simulated times, in the case of 5% scandium, completely deviate from the expected pattern (Table 28). In this case the simulated time is excessively high in comparison to the previous series of simulations. This means that the strategy we used to obtain the real MC time failed in this case, probably due to some very low jumping frequencies that when are inverted result in very high time steps.

Table 27 – Simulation III parameters (1st part).

Temperature [K]	823.15
Lattice size [FCC cells]	50x50x50
MCS	$5x10^{11}$

Sc %	Number Al atoms	Number Sc atoms
1	494999	5000
2	489999	10000
3	484999	15000
4	479999	20000
5	474999	25000

Table 28 – Simulation III simulated time (1st part).

	Simulation Time [s]				
	1% Sc	**2% Sc**	**3% Sc**	**4% Sc**	**5% Sc**
Snapshot I	1.545E-02	2.549E-02	2.870E+00	4.664E-02	5.764E+01
Snapshot II	2.676E-02	4.720E-02	5.629E+00	8.312E-02	1m:5.782E+01
Snapshot III	3.662E-02	6.721E-02	8.220E+00	1.148E-01	2m:5.653E+01
Snapshot IV	4.643E-02	8.630E-02	1.082E+00	1.457E-01	3m:4.599E+01
Snapshot V	5.617E-02	1.040E-01	1.345E+01	1.750E-01	4m:3.360E+01
Snapshot VI	6.547E-02	1.215E-01	1.563E+01	2.032E-01	5m:2.027E+01
Snapshot VII	7.479E-02	1.397E-01	1.778E+01	2.303E-01	6m:5.235E+00
Snapshot VIII	8.378E-02	1.571E-01	1.978E+01	2.566E-01	6m:4.847E+01
Snapshot IX	9.286E-02	1.738E-01	2.208E+01	2.833E-01	7m:5.352E+01
Snapshot X	1.018E-01	1.902E-01	2.421E+01	3.091E-01	8m:3.612E+01
Total Time	1.018E-01	1.902E-01	2.421E+01	3.091E-01	8m:3.612E+01

Table 29 - Simulation III computation time (1st part).

Scandium percentage	Time
1 %	12d:0h:43m:12s
2 %	12d:0h:16m:58s
3 %	12d:1h:59m:10s
4 %	12d:2h:37m:52s
5 %	12d:2h:56m:19s

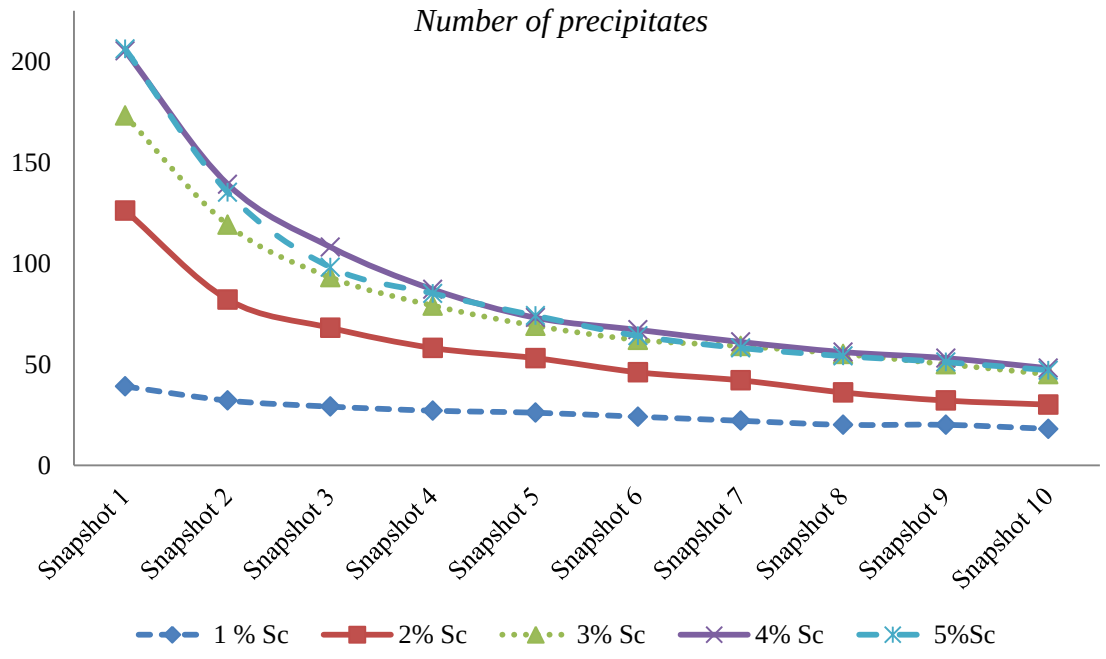


Figure 54 – Simulation III number of precipitates (1st part).

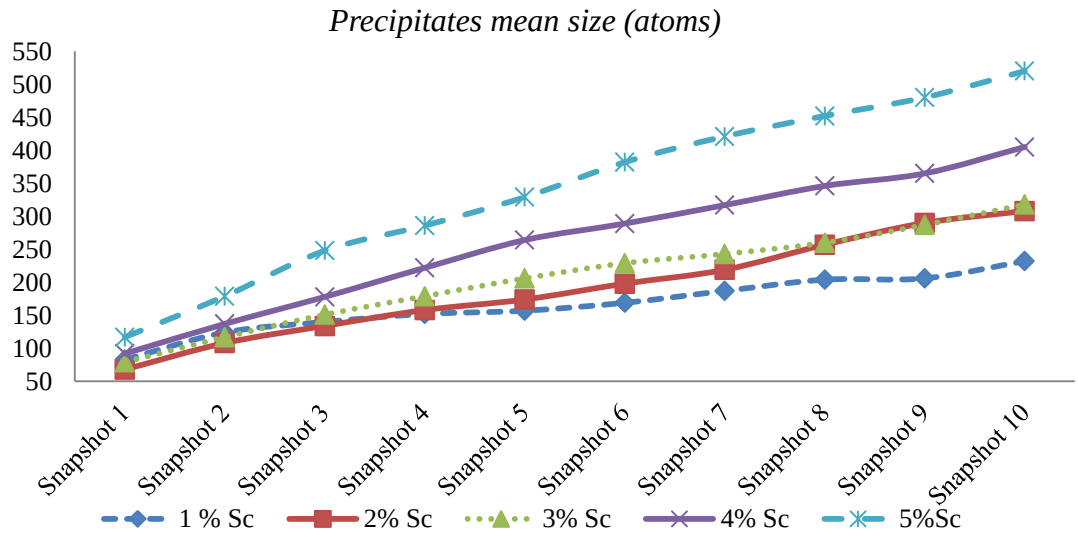


Figure 55 – Simulation III precipitates mean size (1st part).

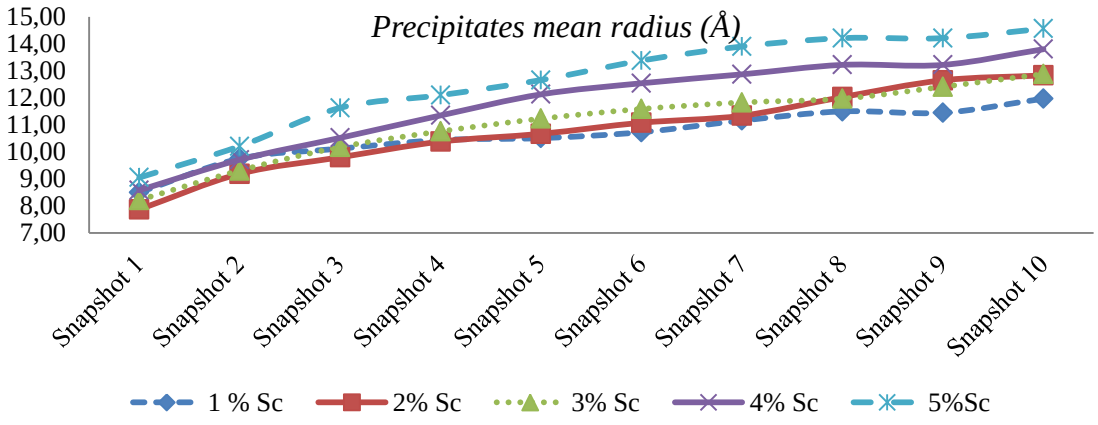


Figure 56 – Simulation III precipitates mean radius (1st part).

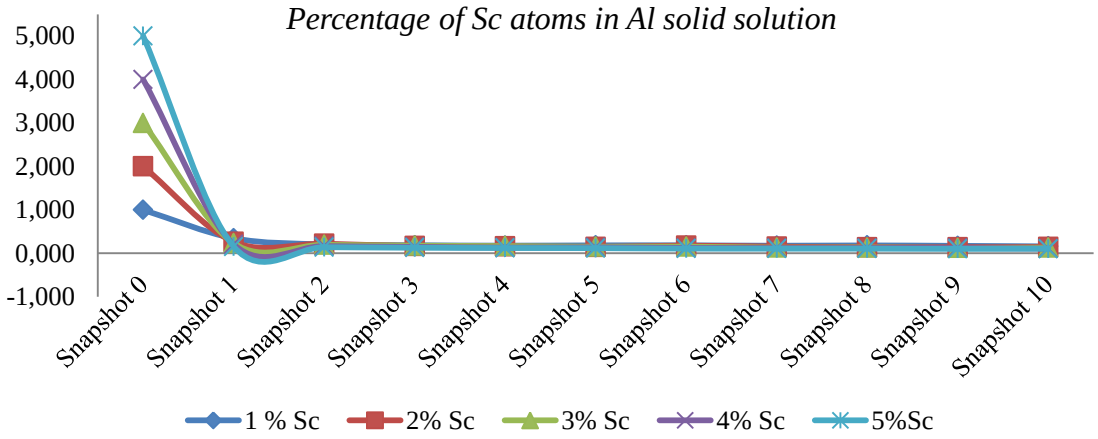


Figure 57 – Simulation III percentage of Sc atoms in Al solid solution (1st part).

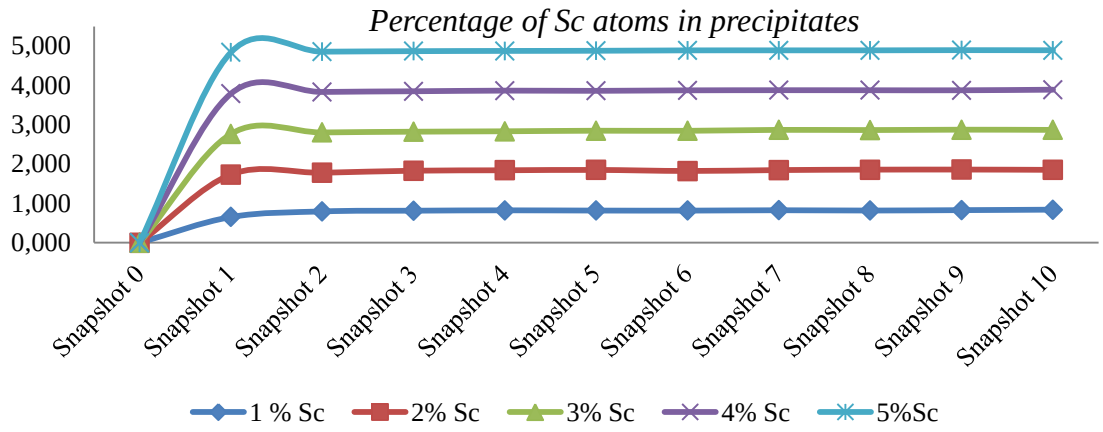


Figure 58 – Simulation III percentage of Sc atoms in precipitates (1st part).

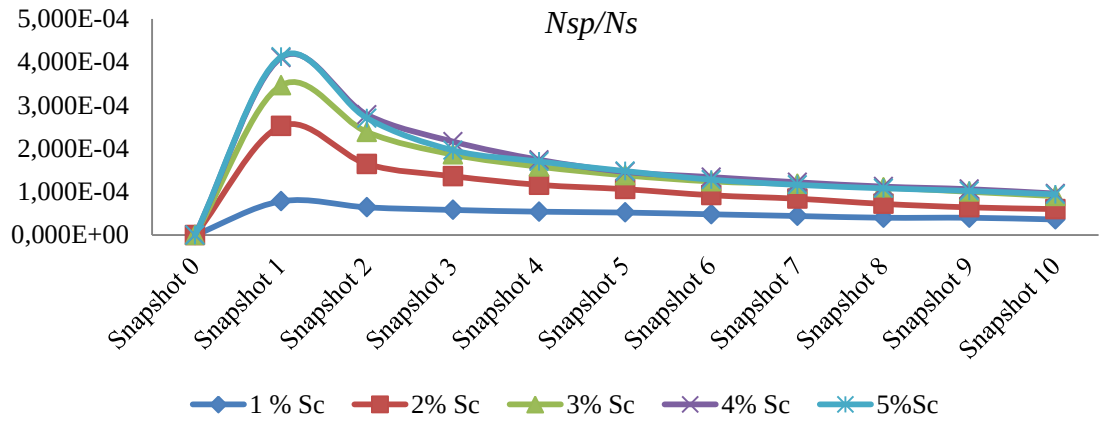


Figure 59 – Simulation III precipitates normalized by the number of lattice sites (1st part).

Next we present the results from simulations carried out with percentages of 0.25, 0.50, 0.75 and 1.25. These simulations are very relevant because they correspond to percentages that will allow us to compare our results with related work.

Table 30 – Simulation III input conditions (2ndpart).

Temperature [K]	823.15
Lattice size [FCC cells]	50x50x50
MCS	$5x10^{11}$

Sc %	Number Al atoms	Number Sc atoms
0.25	498749	1250
0.50	497499	2500
0.75	496249	3750
1.25	493749	6250

Table 31 – Simulation III simulated (2nd part).

Simulation Time [s]				
	0.25% Sc	**0.50% Sc**	**0.75% Sc**	**1.25% Sc**
Snapshot I	1.697E-02	2.695E-02	2.629E-02	2.815E-01
Snapshot II	3.396E-02	5.602E-02	5.680E-02	4.565E-01
Snapshot III	5.090E-02	8.565E-02	8.692E-02	6.057E-01
Snapshot IV	6.786E-02	1.153E-01	1.163E-01	7.495E-01
Snapshot V	8.481E-02	1.447E-01	1.454E-01	8.826E-01
Snapshot VI	1.018E-01	1.730E-01	1.739E-01	1.013E+00
Snapshot VII	1.188E-01	2.014E-01	2.019E-01	1.139E+00
Snapshot VIII	1.357E-01	2.290E-01	2.301E-01	1.263E+00
Snapshot IX	1.527E-01	2.564E-01	2.582E-01	1.383E+00
Snapshot X	1.696E-01	2.834E-01	2.863E-01	1.499E+00
Total Time	1.696E-01	2.834E-01	2.863E-01	1.499E+00

Table 32 – Simulation III computation time (2nd part).

Scandium percentage	Time
0.25 %	8d:21h:47m:47s
0.50 %	8d:22h:28m:33s
0.75 %	8d:20h:54m:48s
1.25 %	8d:22h:25m:40s

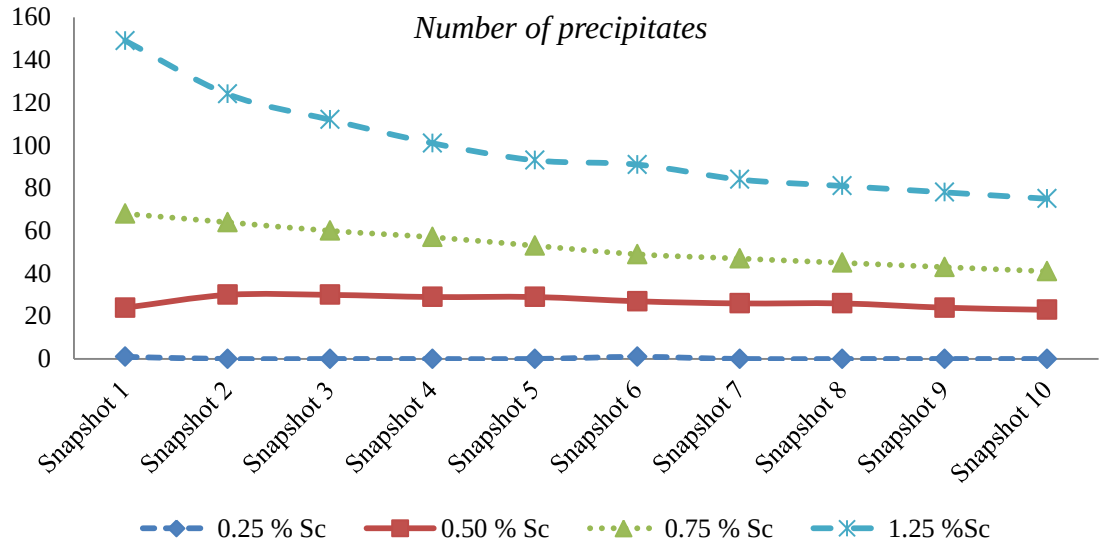


Figure 60 – Simulation III number of precipitates (2nd part).

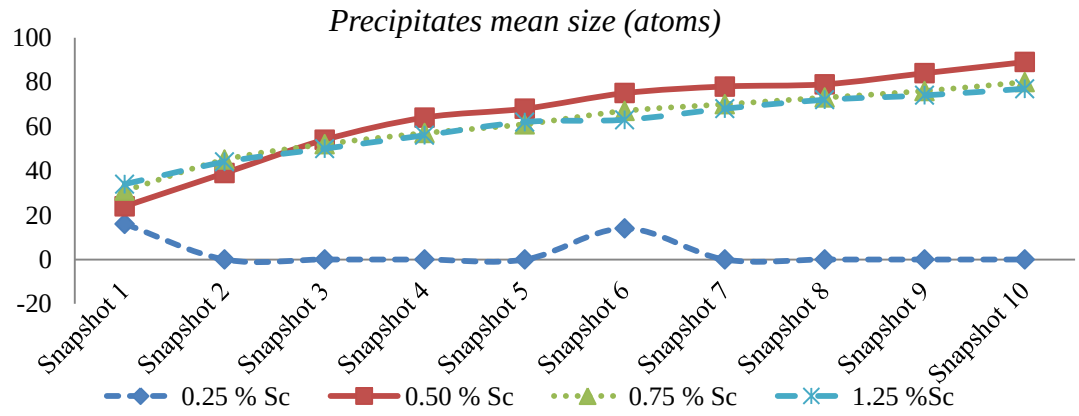


Figure 61 – Simulation III precipitates mean size (2nd part).

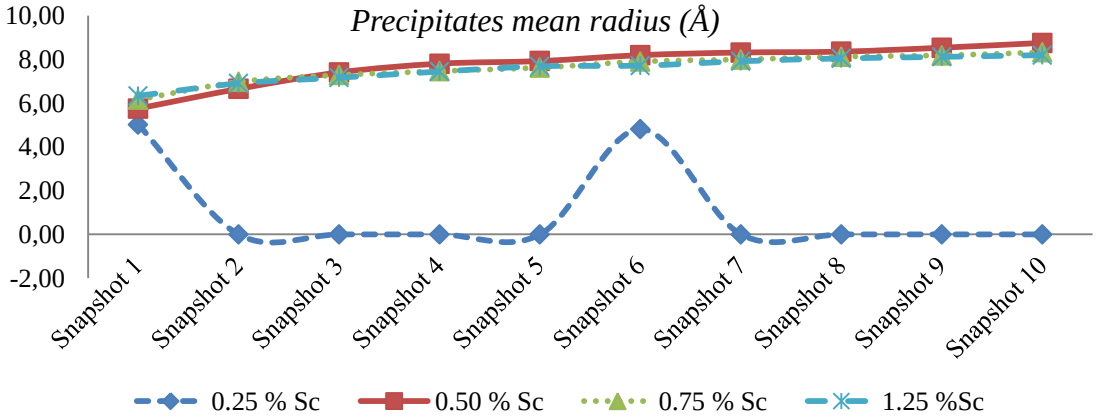


Figure 62 – Simulation III precipitates mean radius (2nd part).

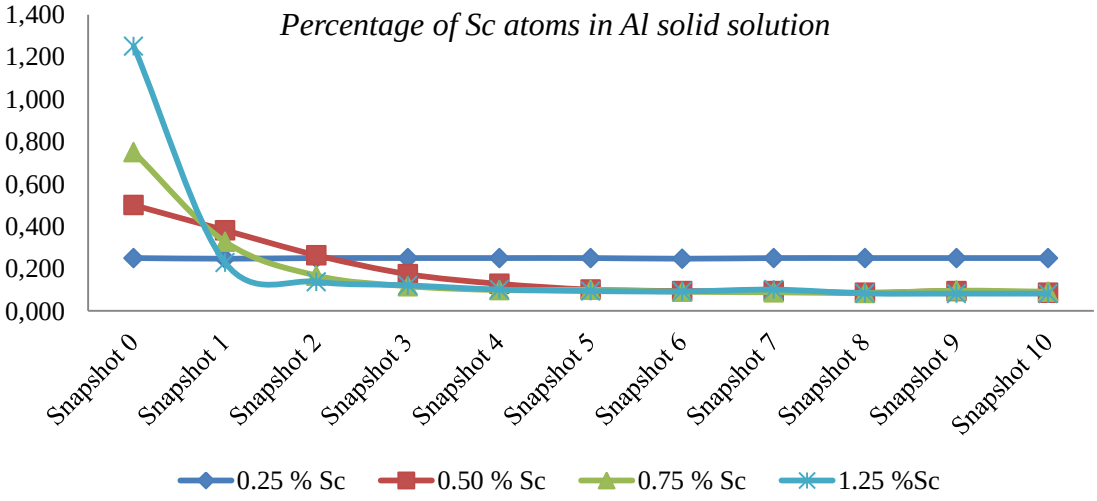


Figure 63 – Simulation III percentage of Sc atoms in Al solution (2nd part).

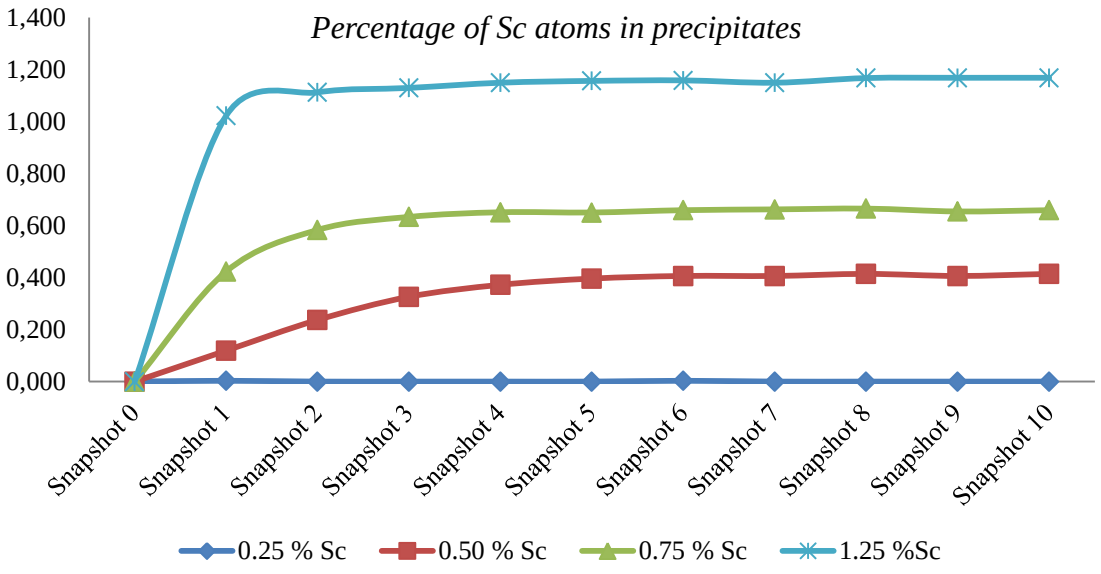


Figure 64 – Simulation III percentage of Sc atoms in precipitates (2nd part).

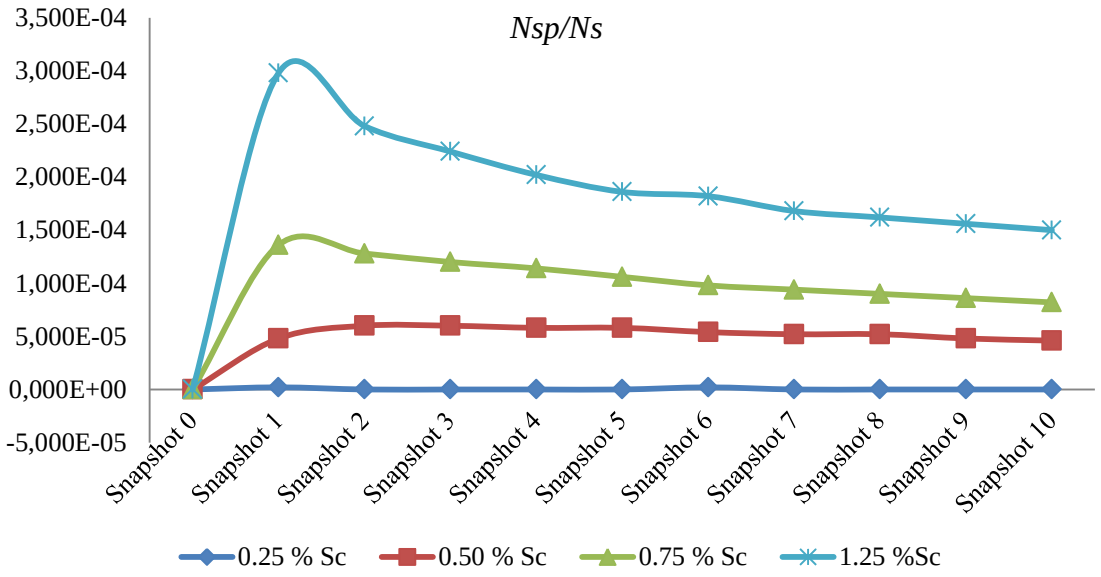


Figure 65 – Simulation III stable precipitates normalized by the number of lattice sites (2nd part).

5.4 Series IV of simulations

This series of simulations was run with a temperature of 773.15K. The applied parameters are listed in Table 33. The simulation underwent successfully although the simulated times, in the cases of 4 and 5 percent scandium, is excessively high and completely deviate from the expected pattern (Table 34). An explanation for this behavior was given in previous series of simulations.

Table 33 – Simulation IV parameters (1st part).

Temperature [K]	773.15
Lattice size [FCC cells]	50x50x50
MCS	$5x10^{11}$

Sc %	Number Al atoms	Number Sc atoms
1	494999	5000
2	489999	10000
3	484999	15000
4	479999	20000
5	474999	25000

Table 34 – Simulation IV simulated time (1st part).

	Simulation Time [s]				
	1% Sc	**2% Sc**	**3% Sc**	**4% Sc**	**5% Sc**
Snapshot I	5.189E-02	6.053E-02	2.017E-01	14m:2.177E+01	7m:4.874E+01
Snapshot II	8.279E-02	1.012E-01	3.480E-01	24m:4.405E+01	15m:1.916E+01
Snapshot III	1.091E-01	1.380E-01	4.776E-01	34m:1.965E+01	22m:2.687E+01
Snapshot IV	1.338E-01	1.722E-01	6.049E-01	43m:4.937E+01	29m:2.827E+01
Snapshot V	1.578E-01	2.048E-01	7.299E-01	52m:5.662E+01	36m:2.007E+01
Snapshot VI	1.811E-01	2.363E-01	8.446E-01	1h:1m:3.639E+01	43m:3.196E+00
Snapshot VII	2.037E-01	2.665E-01	9.609E-01	1h:10m:7.157E+01	49m:3.768E+01
Snapshot VIII	2.257E-01	2.961E-01	1.071E-01	1h:18m:2.917E+01	56m:1.162E+01
Snapshot IX	2.480E-01	3.252E-01	1.179E+00	1h:26m:4.062E+01	1h:2m:1.064E+01
Snapshot X	2.700E-01	3.531E-01	1.285E+00	1h:34m:4.168E+01	1h:8m:2.269E+01
Total Time	2.700E-01	3.531E-01	1.285E+00	1h:34m:4.168E+01	1h:8m:2.269E+01

Table 35 - Simulation IV computation time (1st part).

Scandium percentage	Time
1 %	08d:21h:51m:13s
2 %	08d:19h:54m:06s
3 %	08d:21h:09m:44s
4 %	08d:21h:24m:21s
5 %	12d:03h:45m:45s

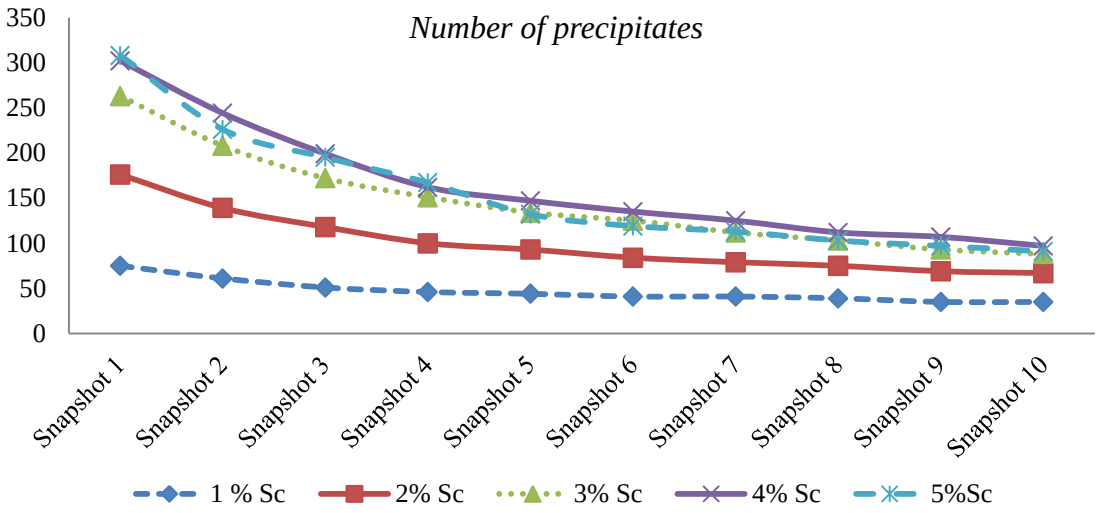


Figure 66 – Simulation IV number of precipitates (1st part).

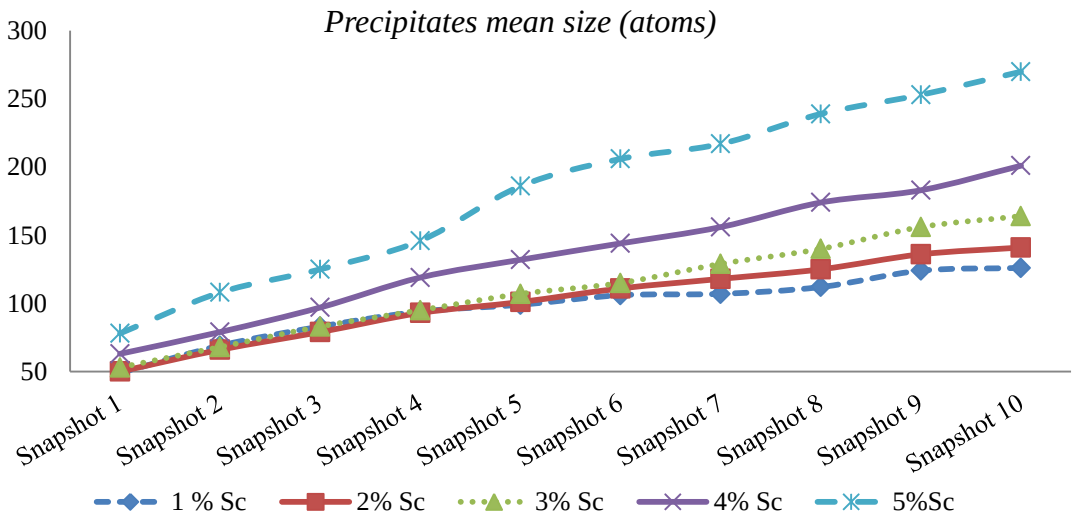


Figure 67 – Simulation IV precipitates mean size (1st part).

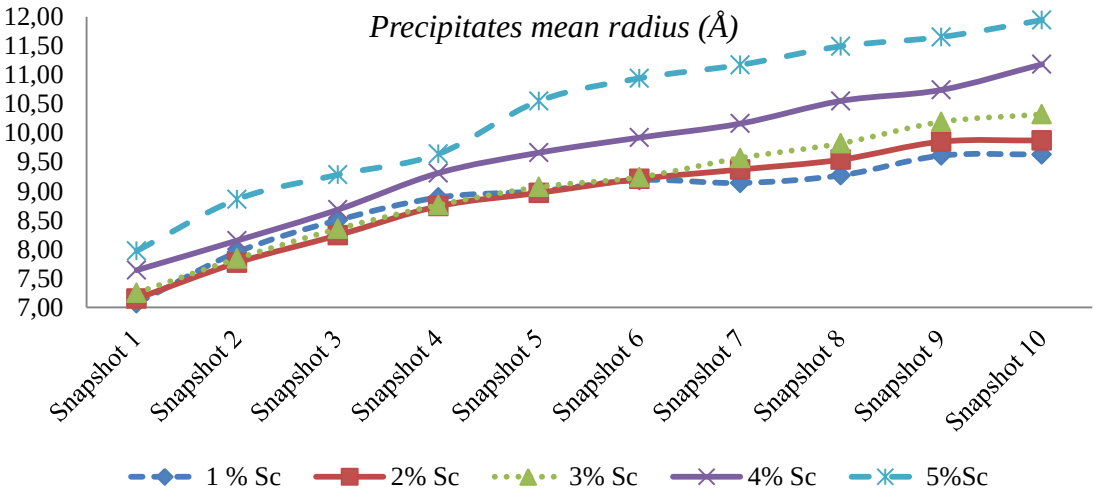


Figure 68 – Simulation IV precipitates mean radius (1st part).

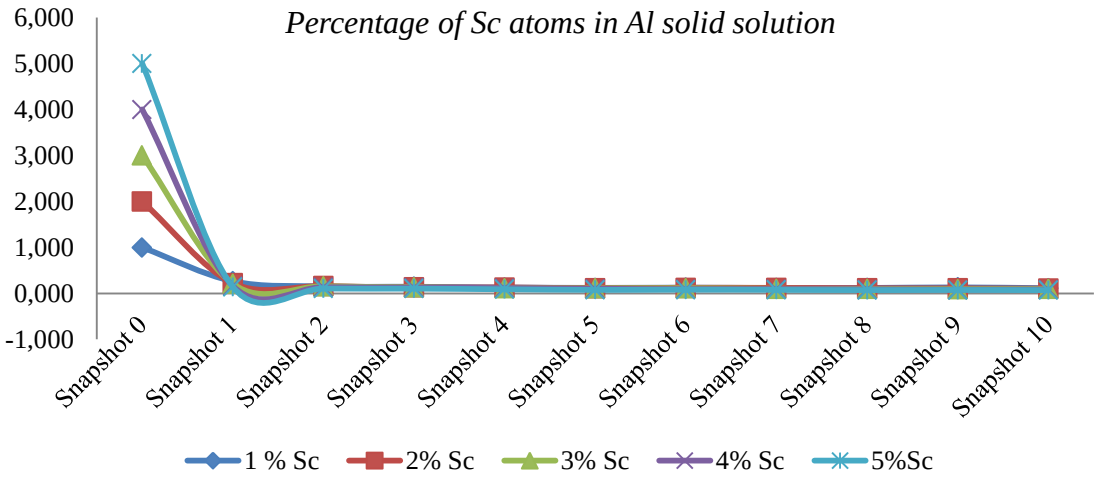


Figure 69 – Simulation IV percentage of Sc atoms in Al solution (1st part).

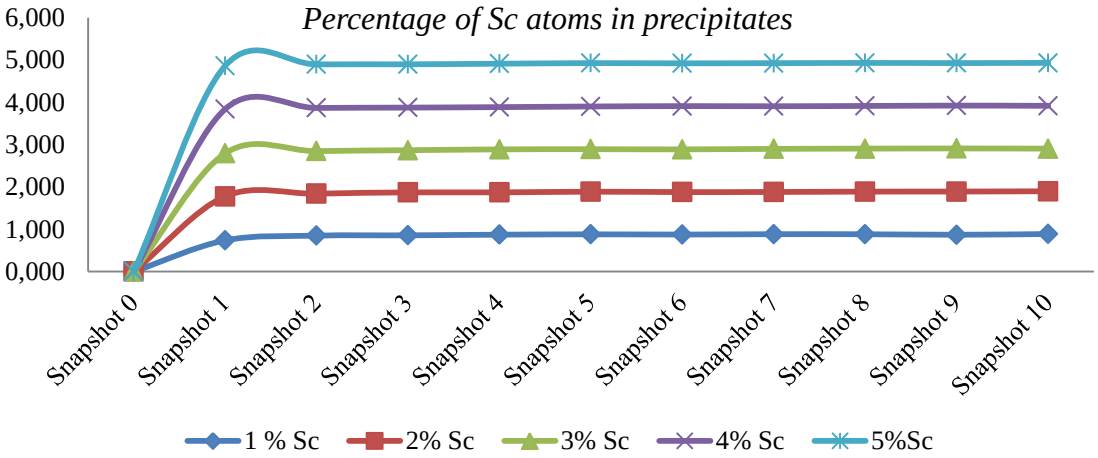


Figure 70 – Simulation IV percentage of Sc atoms in precipitates (1st part).

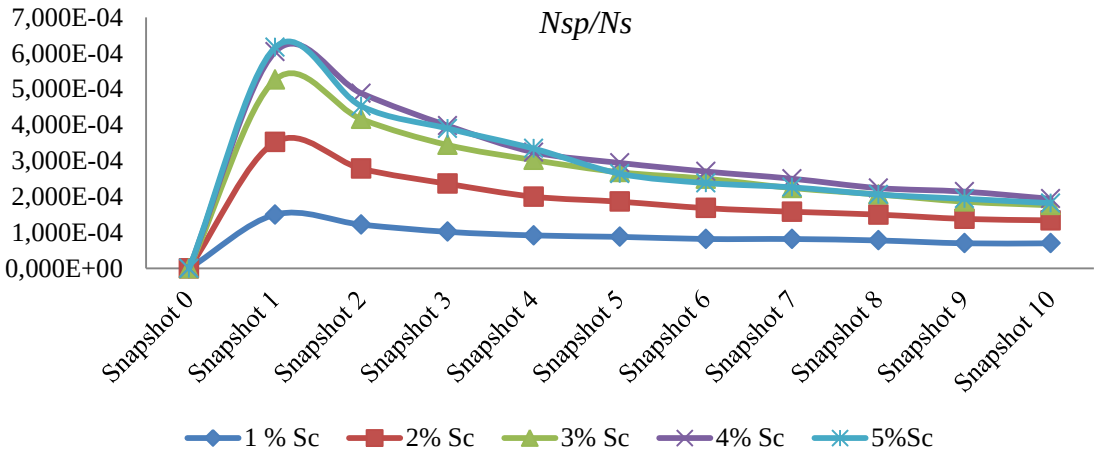


Figure 71 – Simulation IV stable precipitates normalized by the number of lattice sites (1st part).

Next we present the results from simulations carried out with percentages of 0.25, 0.50, 0.75 and 1.25. These simulations will allow us to compare our results with related work. It is interesting to notice that the simulated time of the 1% simulation (0.27s in Table 34) lies between the values of the 0.75% (0.257s in Table 37) and 1.25% (0.655s in Table 37).

Table 36 – Simulation IV parameters (2nd part).

Temperature [K]	773.15
Lattice size [FCC cells]	50x50x50
MCS	$5x10^{11}$

Sc %	Number Al atoms	Number Sc atoms
0.25	498749	1250
0.50	497499	2500
0.75	496249	3750
1.25	493749	6250

Table 37 – Simulation IV simulated time (2nd part).

Simulation Time [s]				
	0.25% Sc	**0.50% Sc**	**0.75% Sc**	**1.25% Sc**
Snapshot I	1.900E-02	1.865E-02	2.332E-02	9.201E-02
Snapshot II	3.817E-02	3.906E-02	4.971E-02	1.682E-01
Snapshot III	5.736E-02	6.112E-02	7.672E-02	2.374E-01
Snapshot IV	7.667E-02	8.336E-02	1.032E-01	3.046E-01
Snapshot V	9.601E-02	1.057E-01	1.293E-01	3.663E-01
Snapshot VI	1.154E-01	1.282E-01	1.555E-01	4.263E-01
Snapshot VII	1.348E-01	1.498E-01	1.811E-01	4.843E-01
Snapshot VIII	1.542E-01	1.716E-01	2.068E-01	5.409E-01
Snapshot IX	1.737E-01	1.939E-01	2.319E-01	5.978E-01
Snapshot X	1.933E-01	2.163E-01	2.571E-01	6.549E-01
Total Time	1.933E-01	2.163E-01	2.571E-01	6.549E-01

Table 38 – Simulation IV computation time (2nd part).

Scandium percentage	**Time**
0.25 %	8d:22h:00m:40s
0.50 %	8d:22h:59m:24s
0.75 %	8d:22h:19m:40s
1.25 %	8d:21h:10m:32s

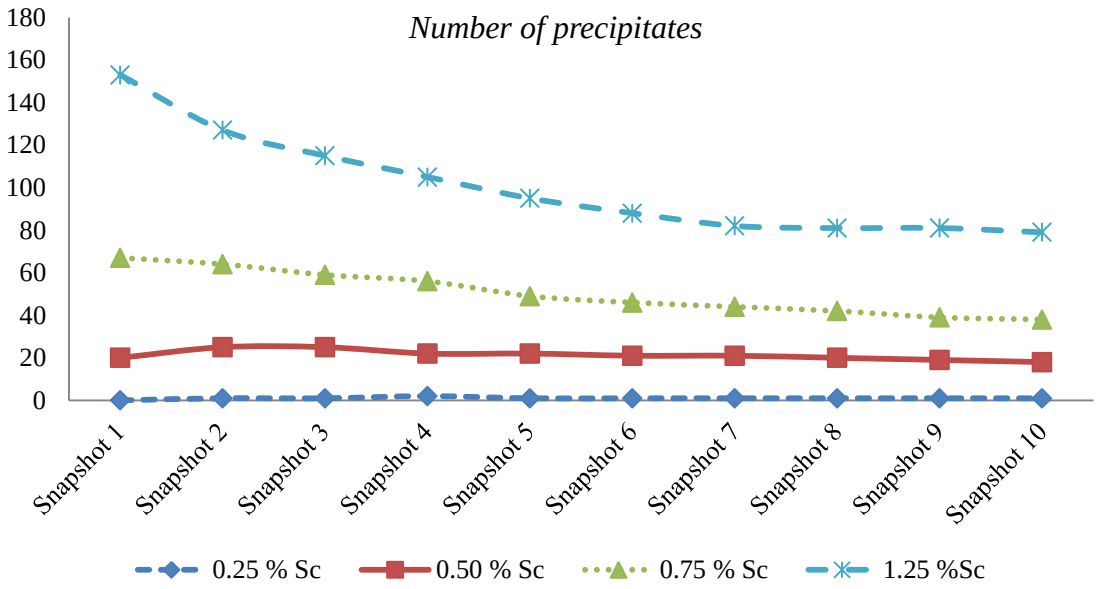


Figure 72 – Simulation IV number of precipitates (2nd part).

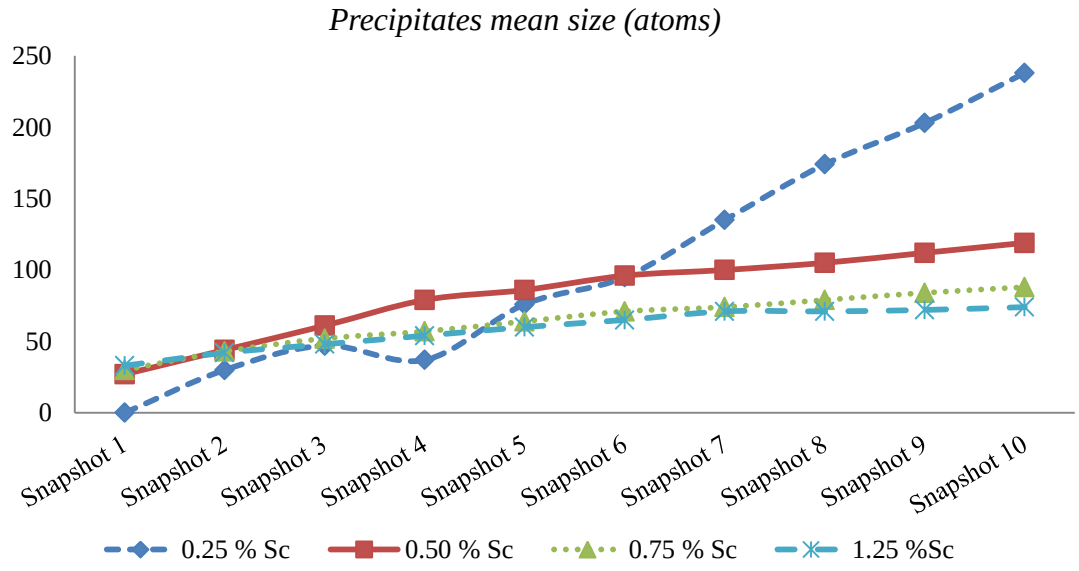


Figure 73 – Simulation IV precipitates mean size (2nd part).

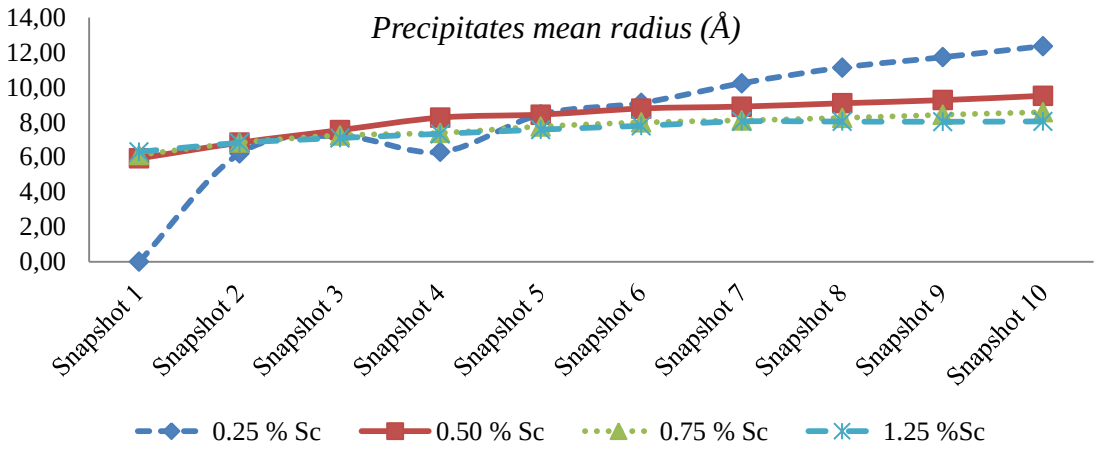


Figure 74 – Simulation IV precipitates mean radius (2nd part).

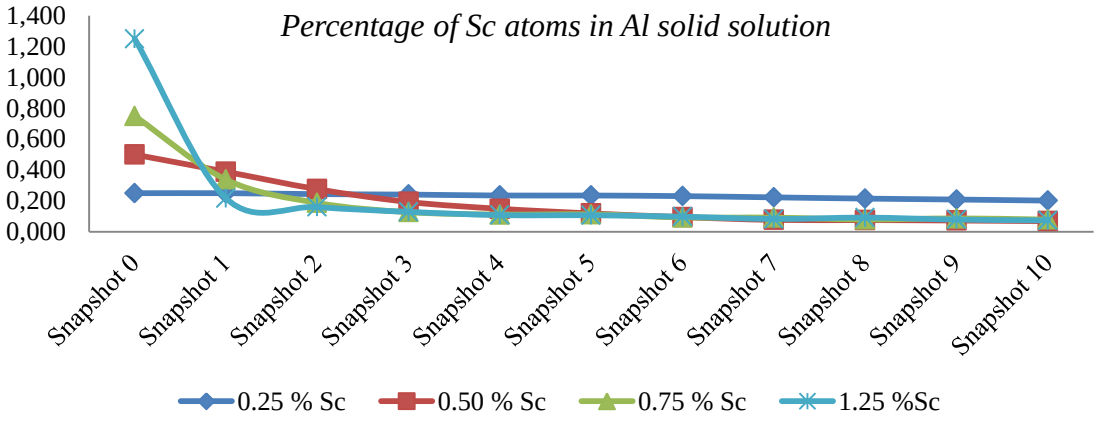


Figure 75 – Simulation IV percentage of Sc atoms in Al solution (2nd part).

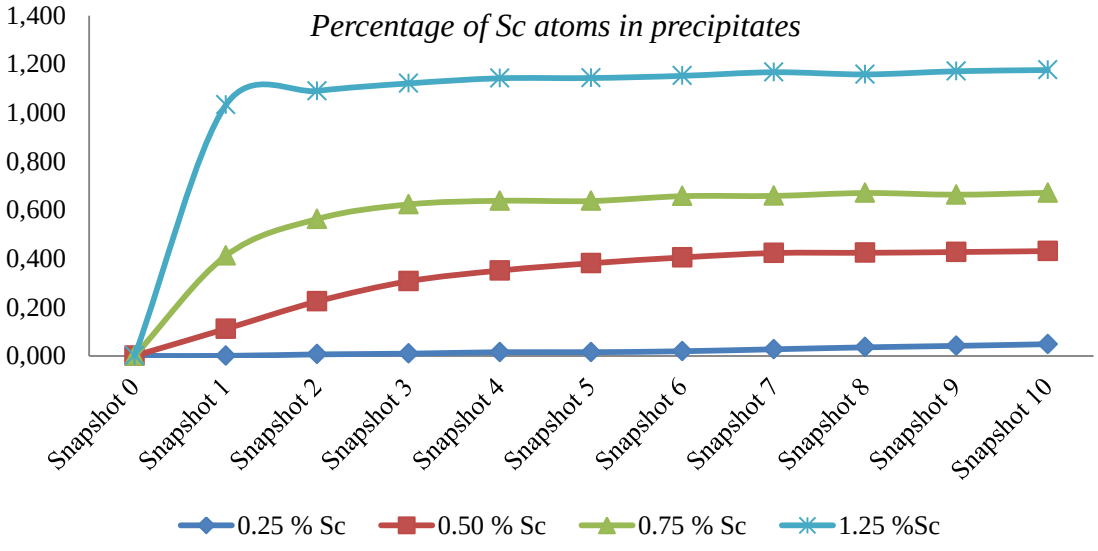


Figure 76 – Simulation IV percentage of Sc atoms in precipitates (2nd part).

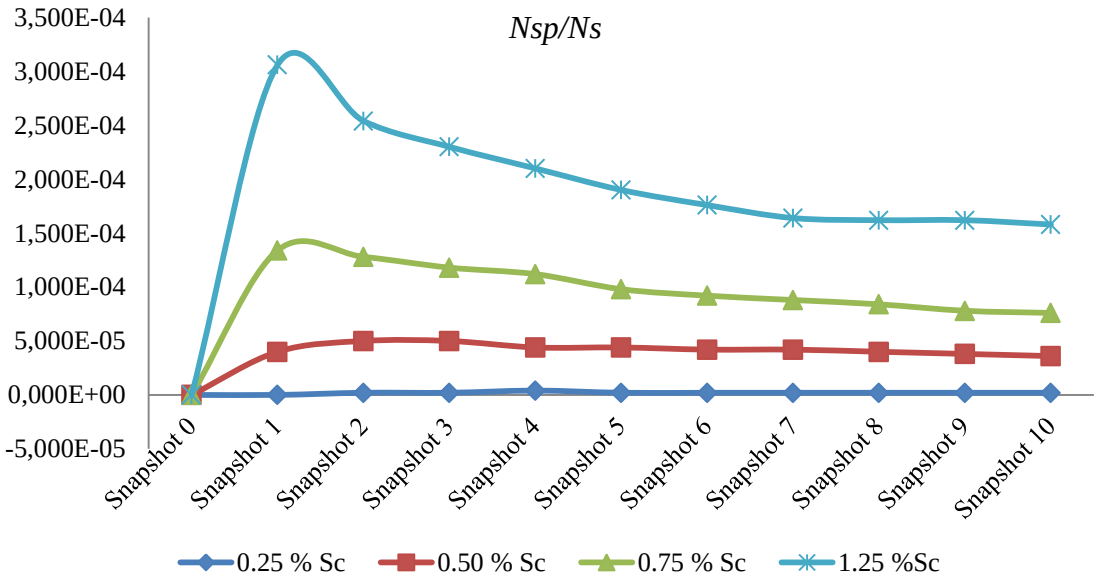


Figure 77 – Simulation IV precipitates normalized by the number of lattice sites (2nd part).

5.5 Series V of simulations

Table 39 contains the parameters applied in the series V of simulation. These simulations are very much identical as previous series, differing only in the applied temperature, which now is 723.15K. The simulations underwent successfully although the simulated time, in the cases of 4 and 5 percent scandium, presents the same problem described in series III and IV.

Table 39 – Simulation V parameters (1st part).

Temperature [K]	723.15
Lattice size [FCC cells]	50x50x50
MCS	$5x10^{11}$

Sc %	Number Al atoms	Number Sc atoms
1	494999	5000
2	489999	10000
3	484999	15000
4	479999	20000
5	474999	25000

Table 40 – Simulation V simulated time (1st part).

	Simulated Time [s]				
	1% Sc	2% Sc	3% Sc	4% Sc	5% Sc
Snapshot I	3.957E-01	3.385E-01	4.025E-01	12m:2.936E+01	2h:33m:4.792E+01
Snapshot II	6.775E-01	5.692E-01	6.952E-01	24m:5.365E+01	4h:52m:2.421E+01
Snapshot III	9.000E-01	7.740E-01	9.566E-01	35m:5.498E+01	7h:2m:5.480E+01
Snapshot IV	1.130E-01	9.657E-01	1.198E+00	47m:2.365E+01	9h:6m:4.649E+01
Snapshot V	1.357E+00	1.146E+00	1.424E+00	57m:3.917E+01	11h:7m:4.124E+01
Snapshot VI	1.565E+00	1.319E+00	1.641E+00	1h:7m:8.038E+00	13h:5m:3.218E+01
Snapshot VII	1.768E+00	1.485E+00	1.849E+00	1h:16m:4.298E+01	15h:0m:1.695E+00
Snapshot VIII	1.967E+00	1.643E+00	2.050E+00	1h:26m:1.152E+01	16h:53m:3.266E+01
Snapshot IX	2.159E+00	1.797E+00	2.243E+00	1h:34m:3.829E+01	18h:45m:4.377E-01
Snapshot X	2.352E+00	1.951E+00	2.430E+00	1h:43m:3.413E+01	1d:0h:5m:4.872E+01
Total Time	2.352E+00	1.951E+00	2.430E+00	1h:43m:3.413E+01	1d:0h:4.872E+01

Table 41 - Simulation V computation time (1st part).

Scandium percentage	Time
1 %	8d:18h:58m:44s
2 %	8d:20h:43m:08s
3 %	8d:22h:02m:53s
4 %	8d:22h:28m:59s
5 %	9d:01h:33m:11s

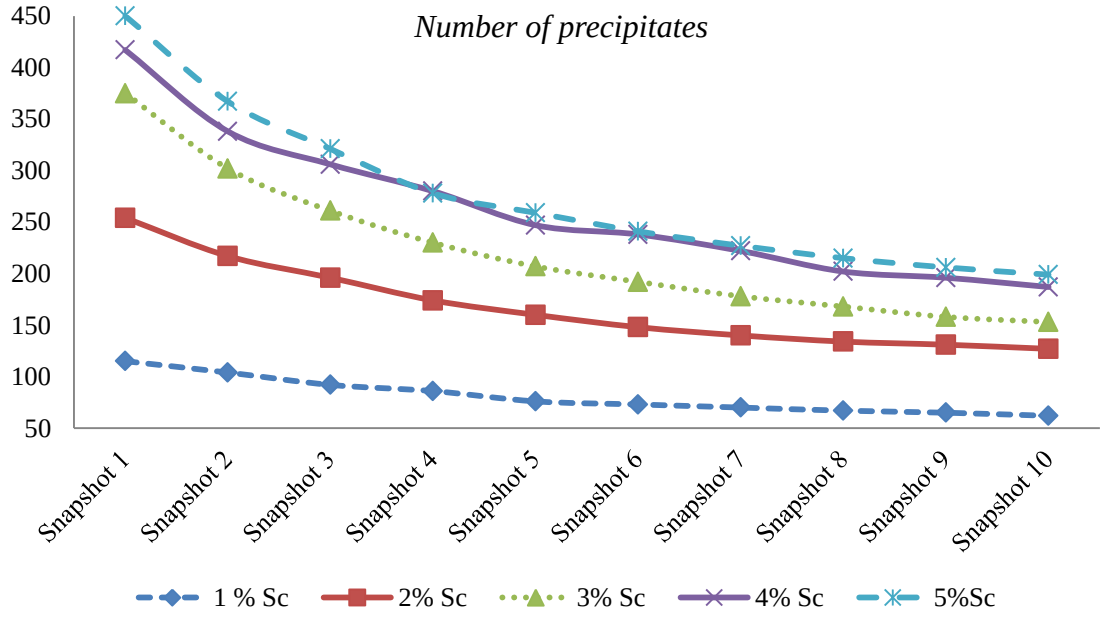


Figure 78 – Simulation V number of precipitates (1st part).

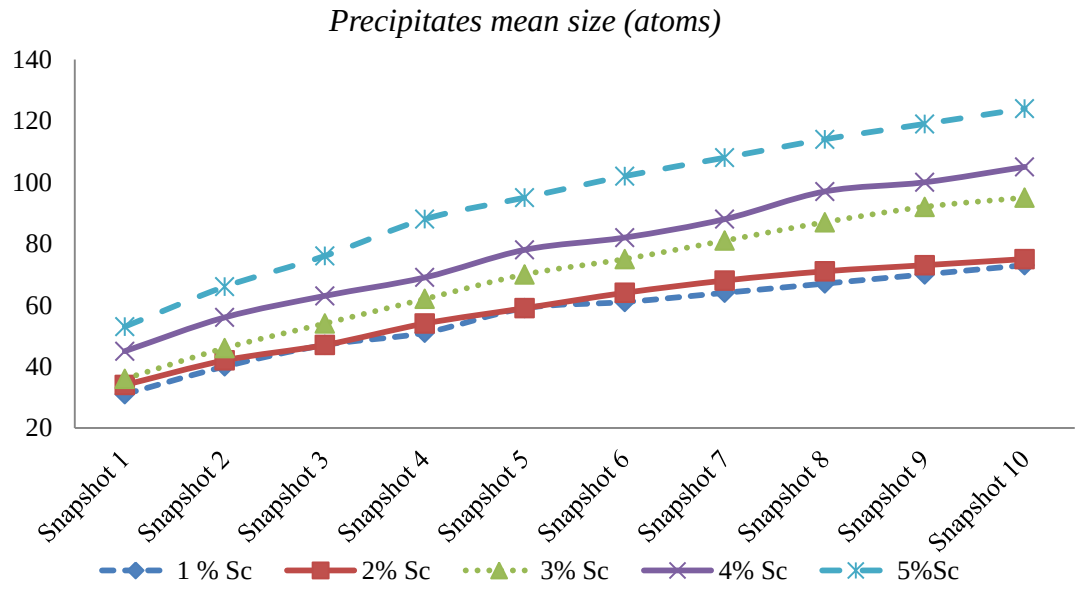


Figure 79 – Simulation V precipitates mean size (1st part).

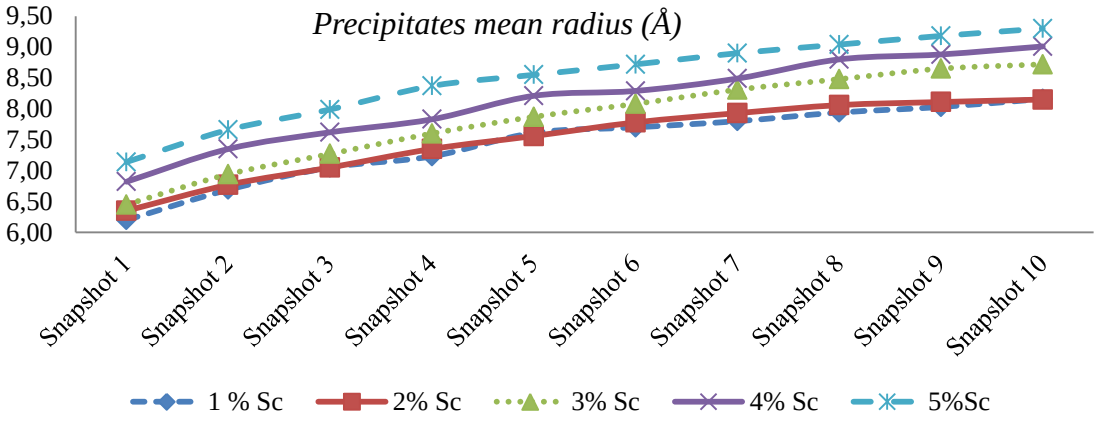


Figure 80 – Simulation V precipitates mean radius (1st part).

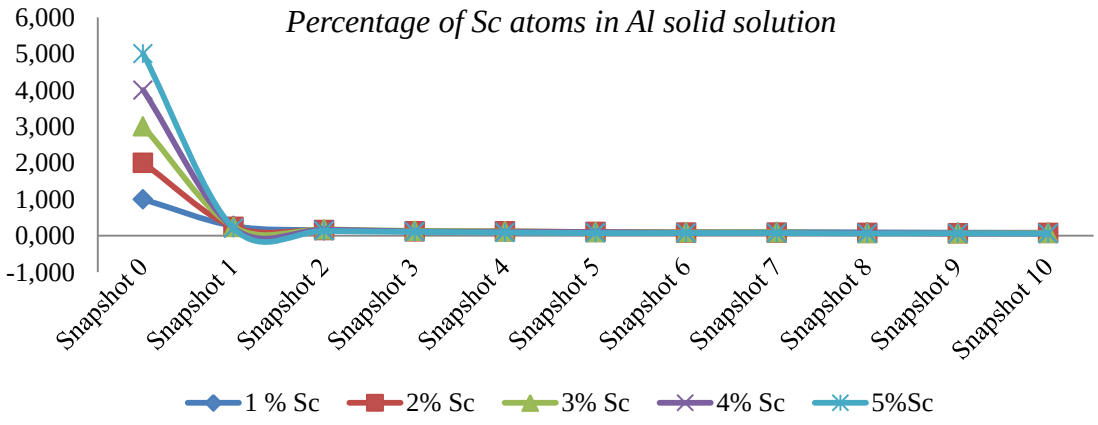


Figure 81 – Simulation V percentage of Sc atoms in Al solution (1st part).

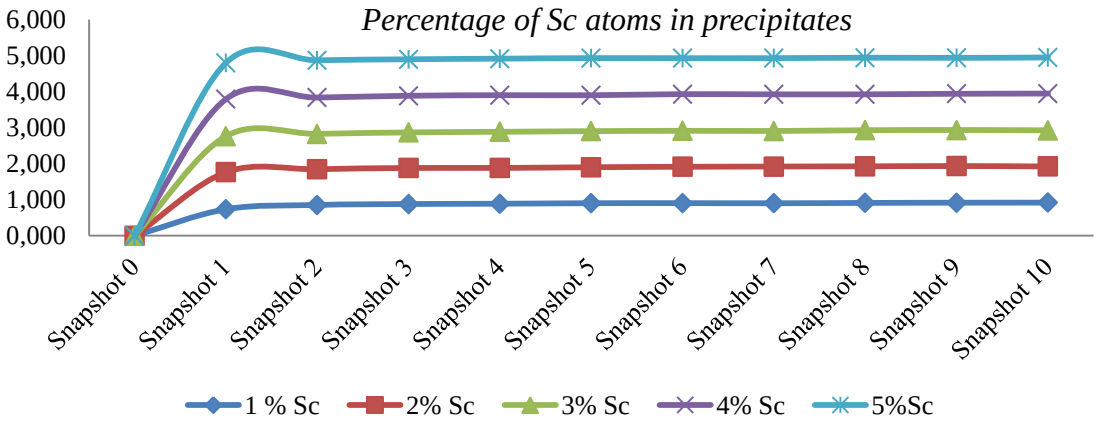


Figure 82 – Simulation V percentage of Sc atoms in precipitates (1st part).

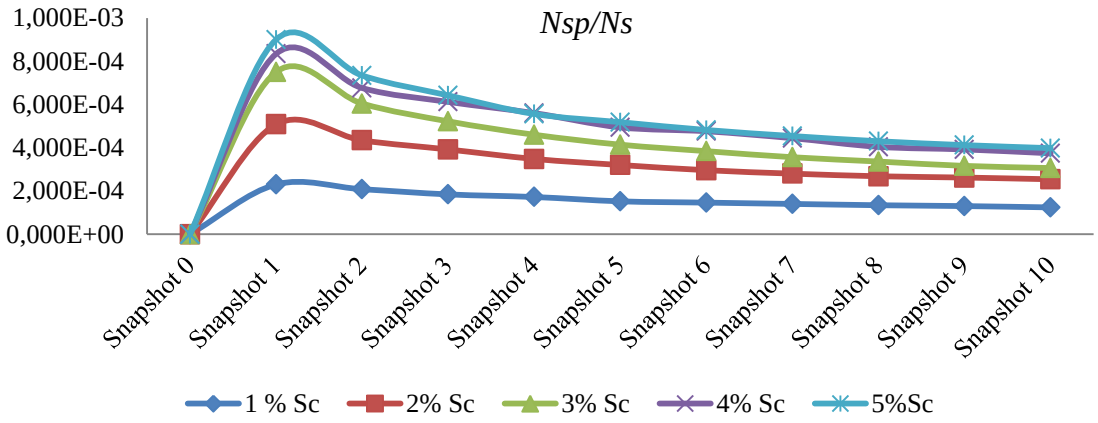


Figure 83 – Simulation V precipitates normalized by the number of lattice sites (1st part).

Additionally, in this series we also performed simulations with lower scandium percentages: 0.25, 0.50, 0.75 and 1.25. These simulations also allow us to compare our results with related work. The obtained results are presented next.

Table 42 – Simulation V parameters (2nd part).

Temperature [K]	723.15
Lattice size [FCC cells]	50x50x50
MCS	$5x10^{11}$

Sc %	Number Al atoms	Number Sc atoms
0.25	498749	1250
0.50	497499	2500
0.75	496249	3750
1.25	493749	6250

Table 43 – Simulation V simulated time (2nd part).

Simulated Time [s]				
	0.25% Sc	**0.50% Sc**	**0.75% Sc**	**1.25% Sc**
Snapshot I	1.712E-02	3.349E-02	5.312E-02	9.201E-02
Snapshot II	3.426E-02	6.848E-02	1.038E-01	1.682E-01
Snapshot III	5.137E-02	1.034E-01	1.505E-01	2.374E-01
Snapshot IV	6.852E-02	1.373E-01	1.942E-01	3.046E-01
Snapshot V	8.568E-02	1.693E-01	2.360E-01	3.663E-01
Snapshot VI	1.028E-01	2.009E-01	2.770E-01	4.264E-01
Snapshot VII	1.199E-01	2.316E-01	3.174E-01	4.843E-01
Snapshot VIII	1.373E-01	2.619E-01	3.565E-01	5.409E-01
Snapshot IX	1.547E-01	2.922E-01	3.947E-01	5.978E-01
Snapshot X	1.723E-01	3.223E-01	4.321E-01	6.549E-01
Total Time	1.723E-01	3.223E-01	4.321E-01	6.549E-01

Table 44 – Simulation V computation time (2nd part).

Scandium percentage	Time
0.25 %	8d:21h:12m:36s
0.50 %	8d:21h:02m:40s
0.75 %	8d:22h:40m:11s
1.25 %	8d:22h:21m:04s

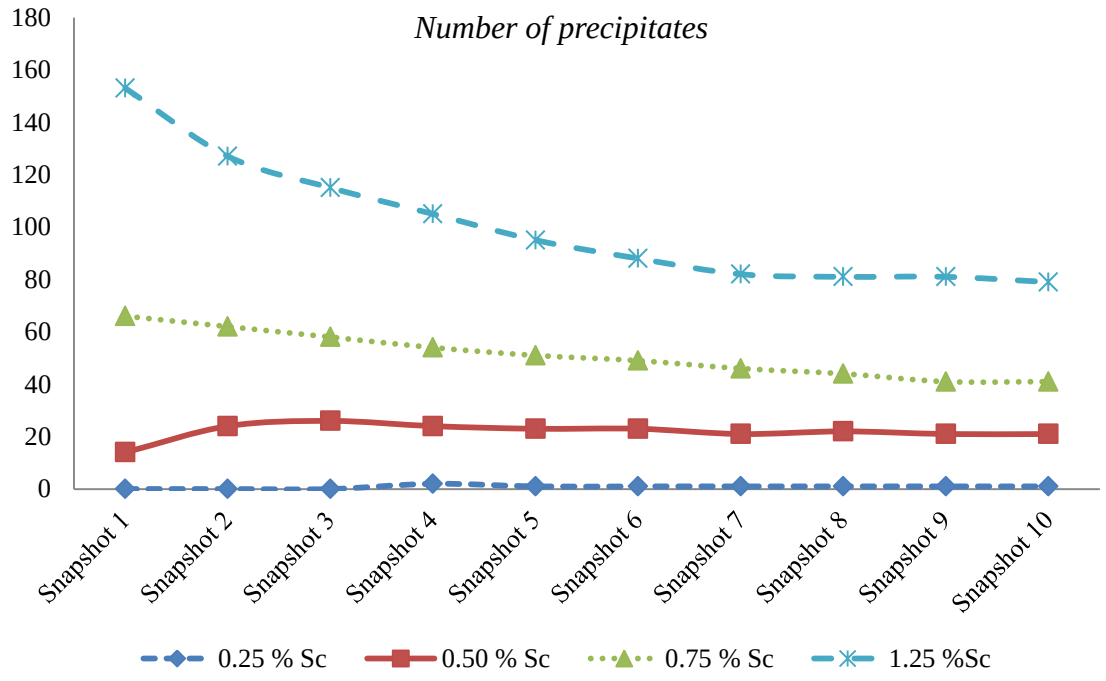


Figure 84 – Simulation V number of precipitates (2nd part).

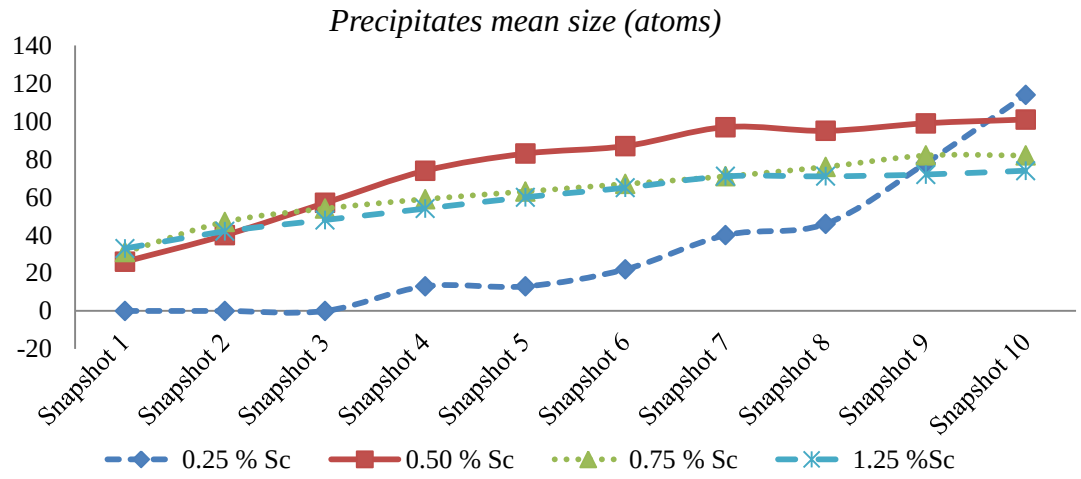


Figure 85 – Simulation V precipitates mean size (2nd part).

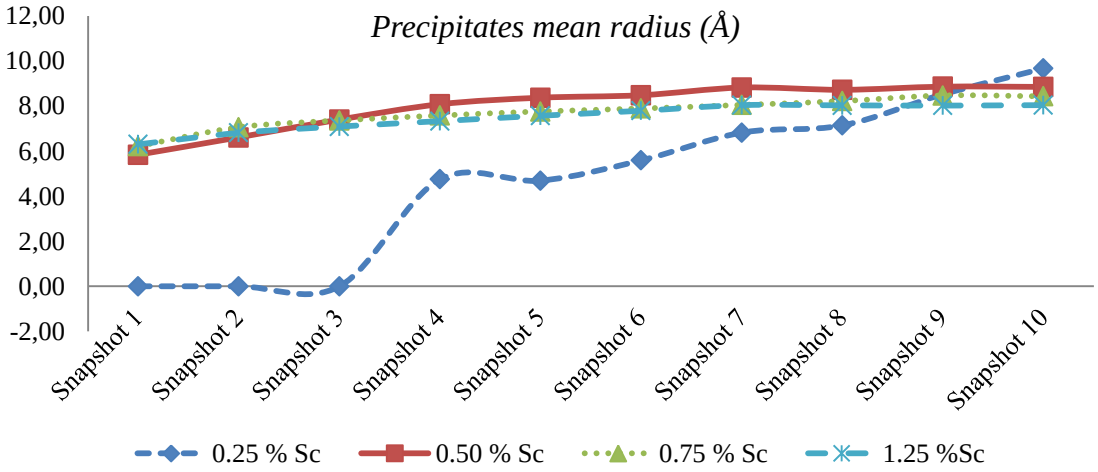


Figure 86 – Simulation V precipitates mean radius (2nd part).

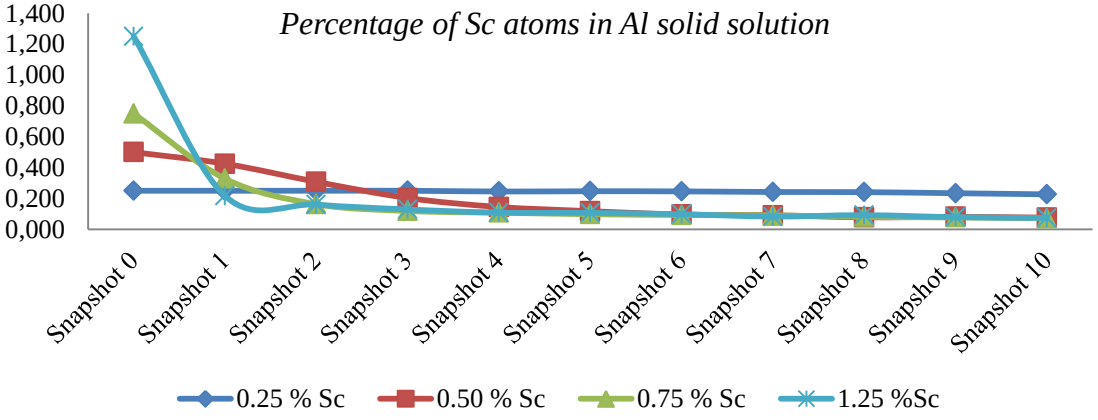


Figure 87 – Simulation V percentage of Sc atoms in Al solid solution (2nd part).

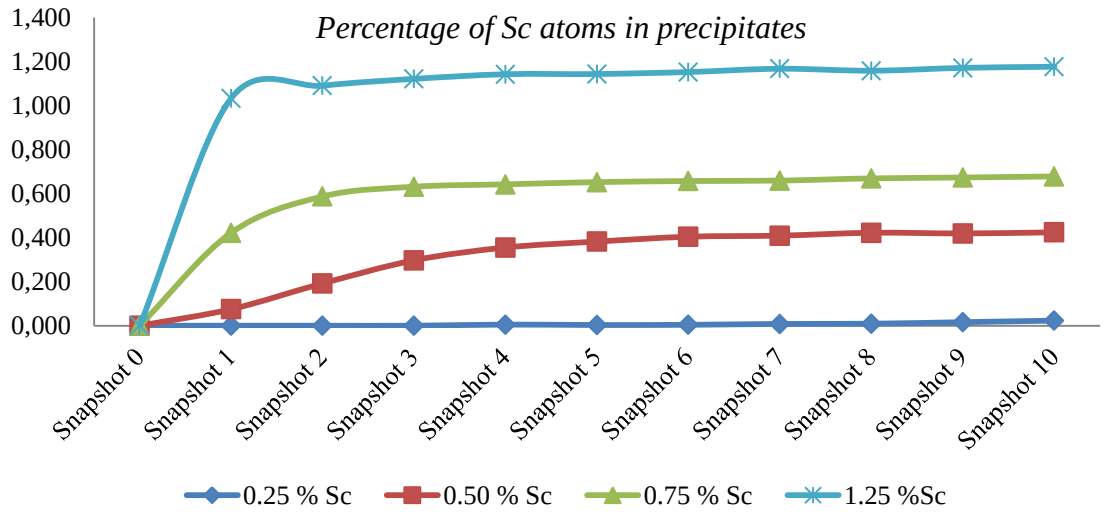


Figure 88 – Simulation V percentage of Sc atoms in precipitates (2nd part).

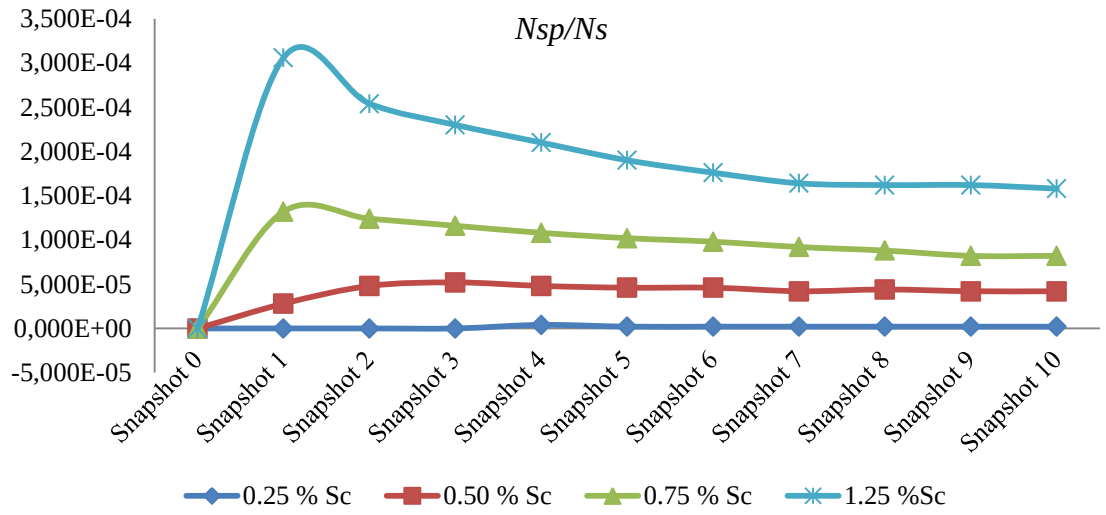


Figure 89 – Simulation V precipitates normalized by the number of lattice sites (2nd part).

5.6 Series VI of simulations

This section presents the results obtained with the simulations carried out with a temperature of 673.15 K. Table 45 contains the applied parameters, Table 46 the simulated time for each scandium percentage and Table 47 the computation time. Other results are documented in Figure 90 through Figure 95.

Table 45 – Simulation VI parameters.

Temperature [K]	673.15
Lattice size [FCC cells]	50x50x50
MCS	$5x10^{11}$

Sc %	Number Al atoms	Number Sc atoms
1	494999	5000
2	489999	10000
3	484999	15000
4	479999	20000
5	474999	25000

Table 46 – Simulation VI simulated time.

Simulated Time [s]					
	1% Sc	**2% Sc**	**3% Sc**	**4% Sc**	**5% Sc**
Snapshot I	4.815E-01	1.085E+00	2.532E+01	1m:9.140E+00	3.443E+00
Snapshot II	9.249E-01	1.912E+00	4.933E+01	2m:1.414E+01	6.079E+00
Snapshot III	1.318E+00	2.640E+00	1m:1.035E+01	3m:1.179E+01	8.470E+00
Snapshot IV	1.685E+00	3.315E+00	1m:2.984E+01	4m:5.616E+00	1.073E+01
Snapshot V	2.039E+00	3.951E+00	1m:4.974E+01	4m:5.824E+01	1.289E+01
Snapshot VI	2.380E+00	4.565E+00	2m:9.463E+00	5m:4.927E+01	1.498E+01
Snapshot VII	2.713E+00	5.155E+00	2m:2.972E+01	6m:3.915E+01	1.701E+01
Snapshot VIII	3.038E+00	5.738E+00	2m:5.018E+01	7m:2.831E+01	1.901E+01
Snapshot IX	3.364E+00	6.293E+00	3m:1.024E+01	8m:1.569E+01	2.097E+01
Snapshot X	3.673E+00	6.826E+00	3m:2.845E+01	9m:2.503E+00	2.288E+01
Total Time	3.673E+00	6.826E+00	3m:2.845E+01	9m:2.503E+00	2.288E+01

Table 47 - Simulation VI computation time.

Scandium percentage	Time
1 %	8d:22h:03m:28s
2 %	8d:23h:23m:20s
3 %	9d:00h:34m:47s
4 %	9d:10h:53m:48s
5 %	8d:23h:56m:35s

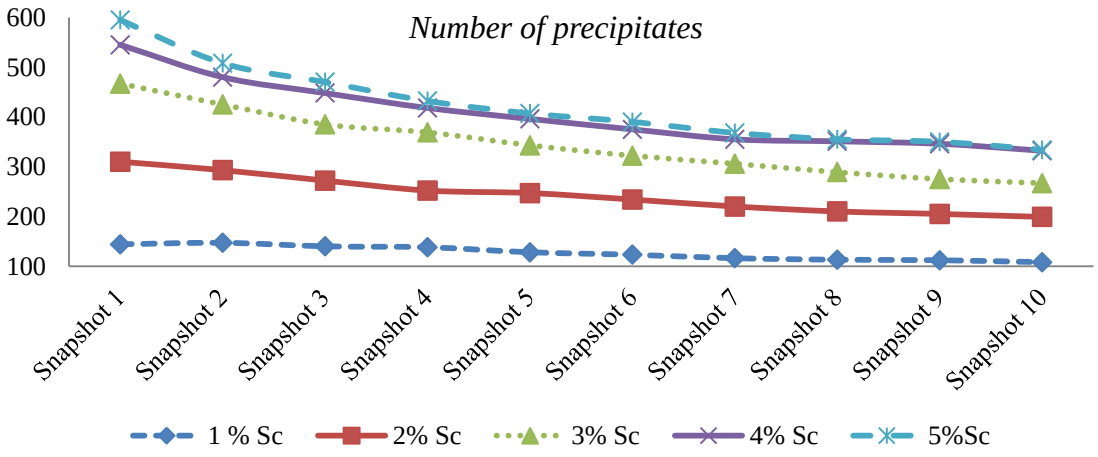


Figure 90 – Simulation VI number of precipitates results.

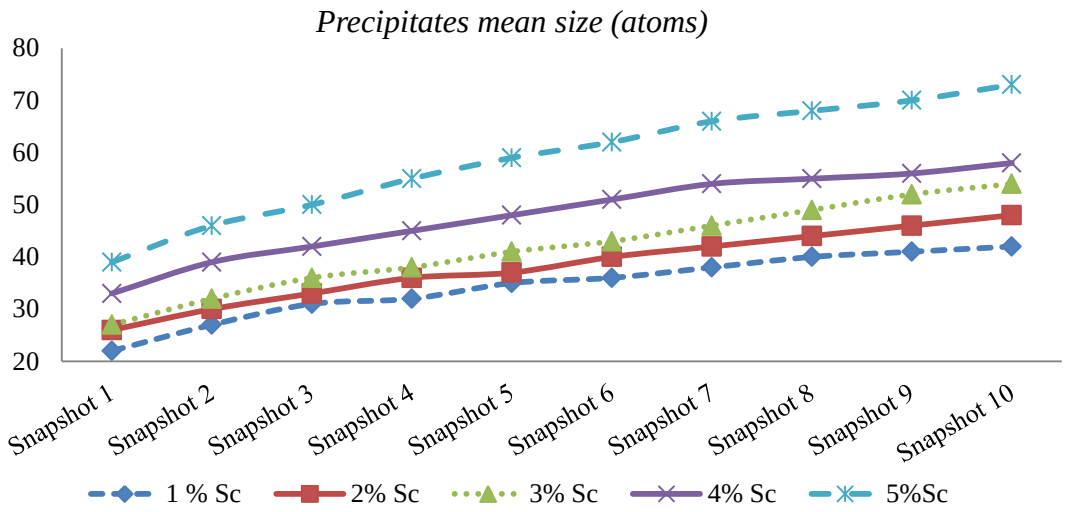


Figure 91 – Simulation VI precipitates mean size results.

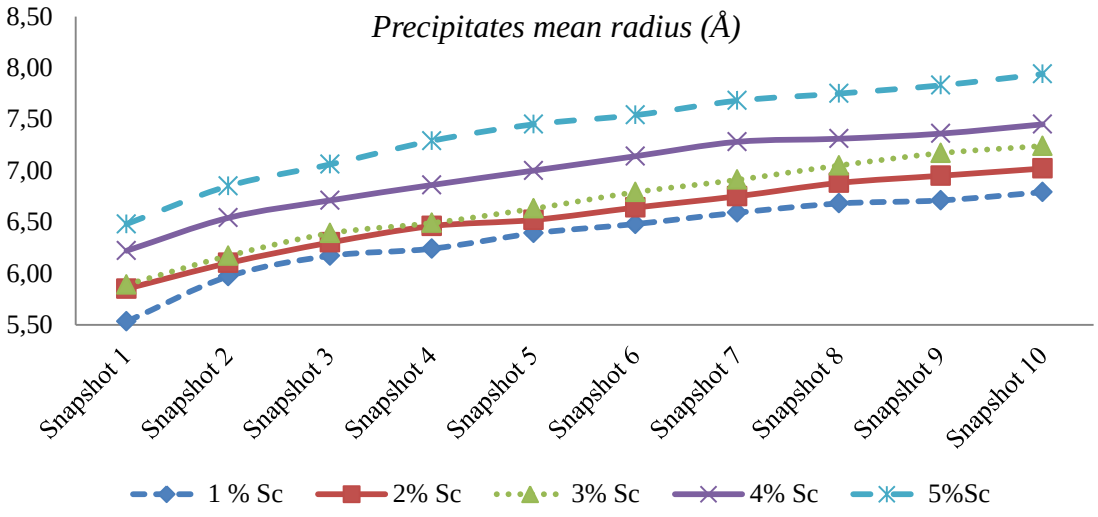


Figure 92 – Simulation VI precipitates mean radius results.

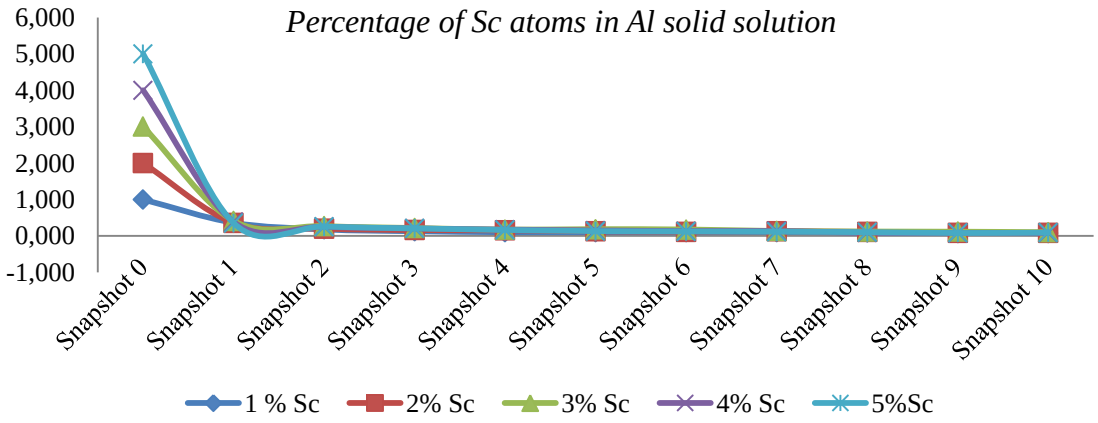


Figure 93 – Simulation VI percentage of Sc atoms in Al solution results.

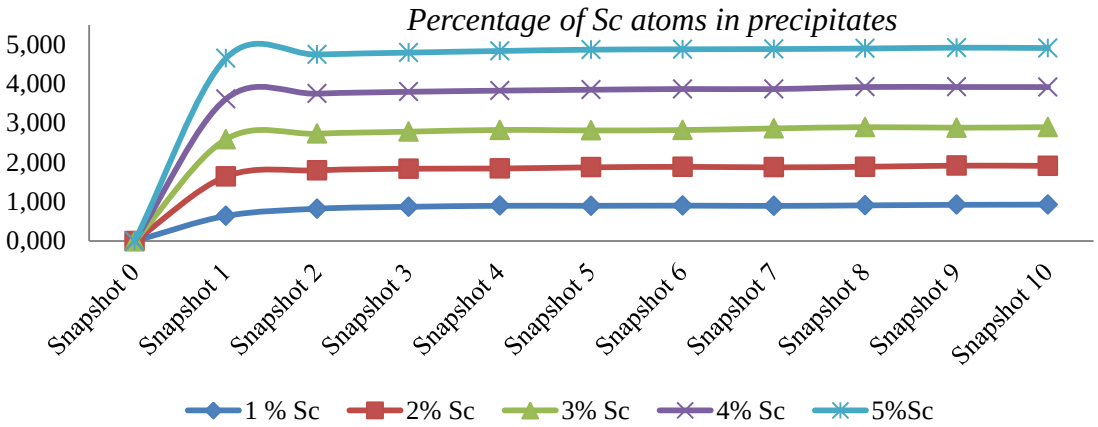


Figure 94 – Simulation VI percentage of Sc atoms in precipitates results.

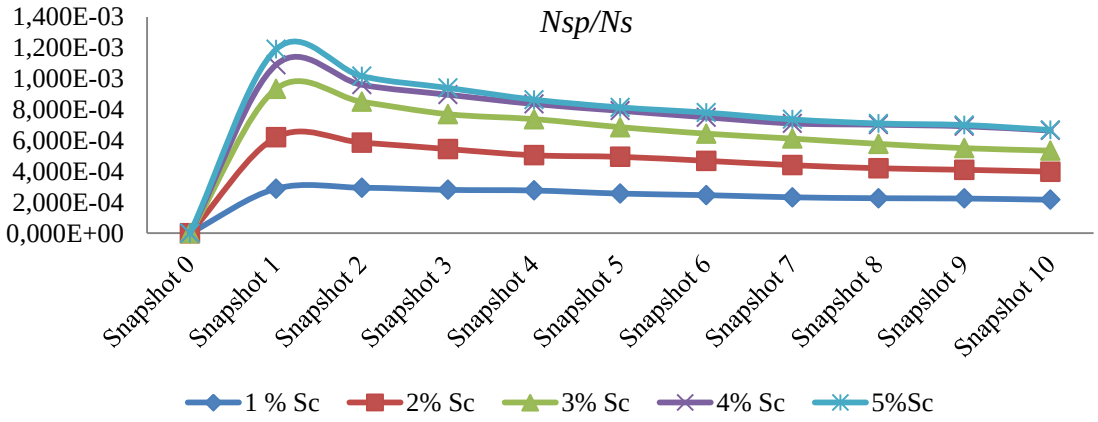


Figure 95 – Simulation VI precipitates normalized by the number of lattice sites.

5.7 Series VII of simulations

The last series of simulations was carried out with a temperature of 573.15 K. Table 48 contains the applied parameters, Table 49 the simulated time for each scandium percentage and Table 50 the computation time. As can be observed from Table 49, the strategy for calculating the MC real time proved ineffective once more, generating excessively high times, especially in the simulation with 4% scandium. The other results are documented in Figure 96 through Figure 101.

Table 48 – Simulation VII input conditions.

Temperature [K]	573.15
Lattice size [FCC cells]	50x50x50
MCS	$5x10^{11}$

Sc %	Number Al atoms	Number Sc atoms
1	494999	5000
2	489999	10000
3	484999	15000
4	479999	20000
5	474999	25000

Table 49 – Simulation VII simulated time.

	Simulated Time [s]				
	1% Sc	**2% Sc**	**3% Sc**	**4% Sc**	**5% Sc**
Snapshot I	0m:09s	0m:54s	04m:07s	017d:11h:39m:19s	03m:44s
Snapshot II	0m:22s	1m:49s	07m:26s	043d:03h:51m:18s	06m:56s
Snapshot III	0m:37s	2m:44s	10m:28s	068d:10h:40m:35s	09m:59s
Snapshot IV	0m:54s	3m:38s	13m:22s	096d:09h:11m:51s	12m:55s
Snapshot V	1m:10s	4m:31s	16m:11s	128d:10h:41m:19s	15m:44s
Snapshot VI	1m:28s	5m:24s	18m:55s	162d:11h:47m:28s	18m:29s
Snapshot VII	1m:46s	6m:16s	21m:34s	200d:00h:11m:37s	21m:13s
Snapshot VIII	2m:45s	7m:06s	24m:11s	237d:20h:46m:54s	23m:54s
Snapshot IX	2m:23s	7m:56s	26m:45s	274d:21h:48m:28s	26m:32s
Snapshot X	2m:41s	8m:45s	29m:17s	312d:05h:15m:48s	29m:81s
Total Time	2m:41s	8m:45s	29m:17s	312d:5h:15m:48s	29m:81s

Table 50 - Simulation VII computation time.

Scandium percentage	**Time**
1 %	12d:06h:11m:20s
2 %	12d:04h:58m:18s
3 %	12d:02h:56m:03s
4 %	12d:02h:48m:35s
5 %	12d:06h:04m:27s

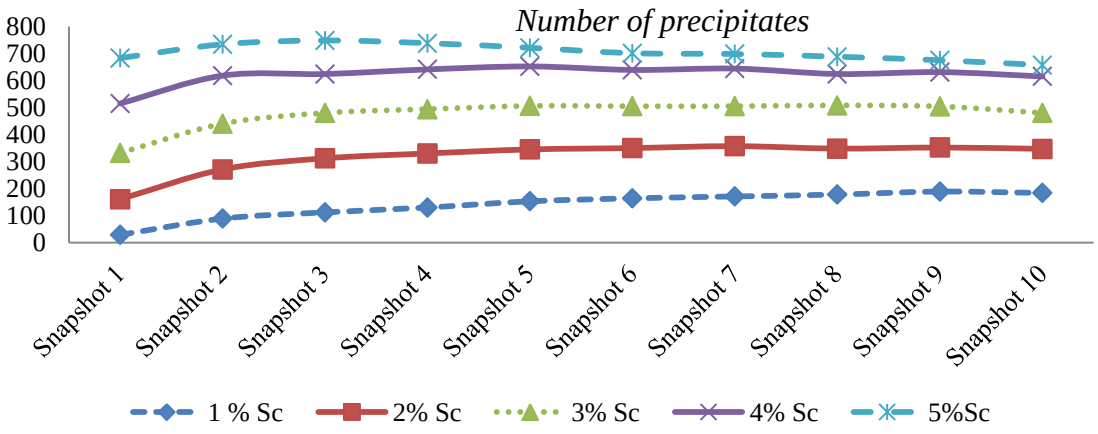


Figure 96 – Simulation VII number of precipitates results.

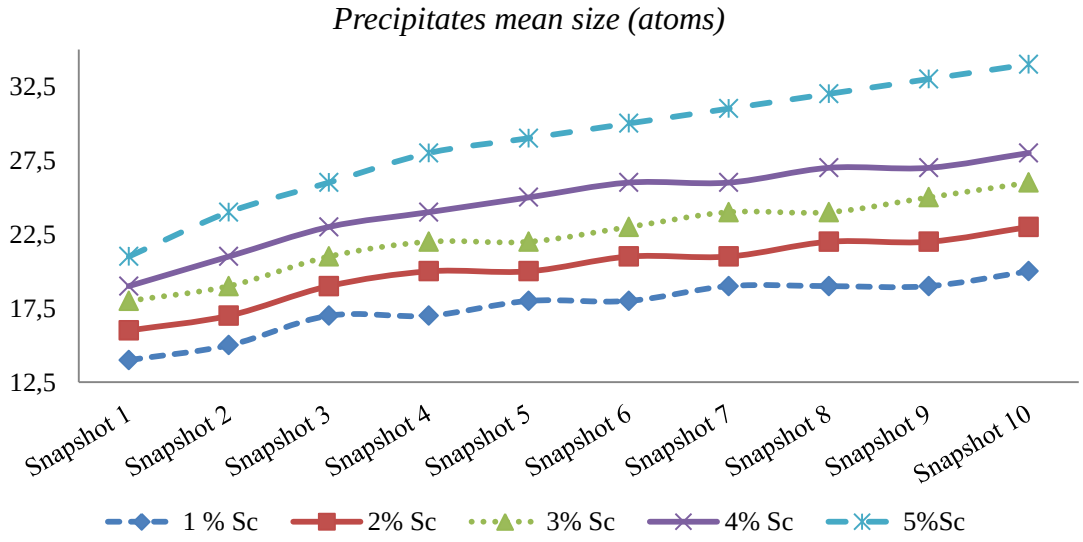


Figure 97 – Simulation VII precipitates mean size results.

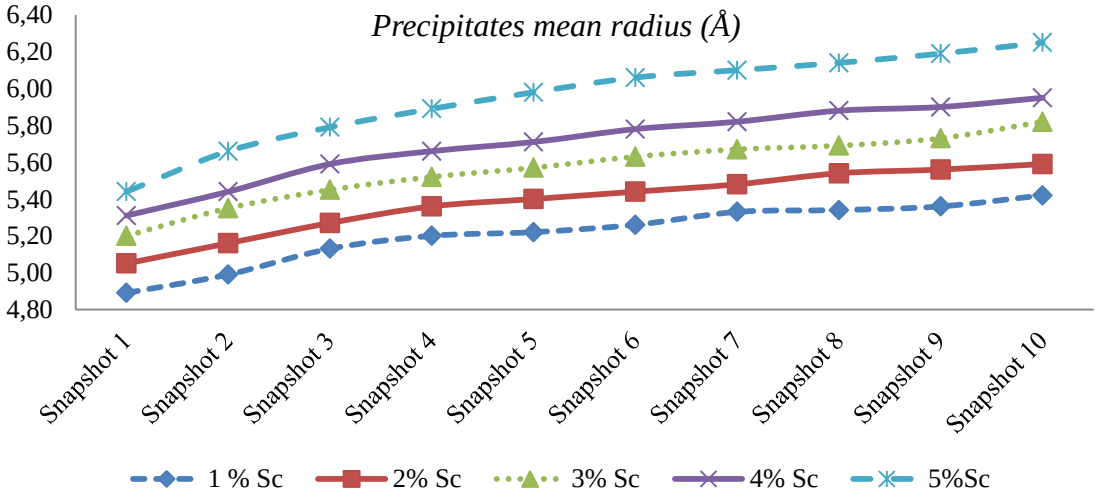


Figure 98 – Simulation VII precipitates mean radius results.

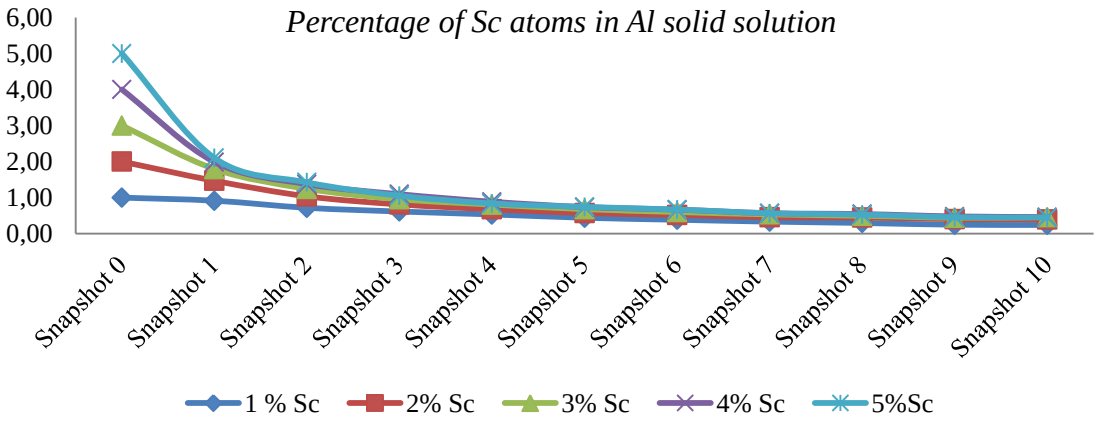


Figure 99 – Simulation VII percentage of Sc atoms in Al solid solution results.

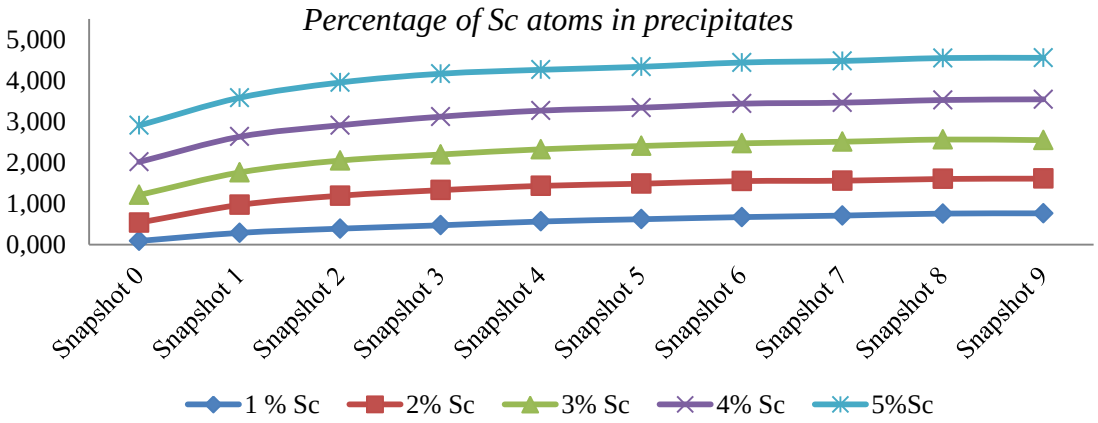


Figure 100 – Simulation VII percentage of Sc atoms in precipitates results.

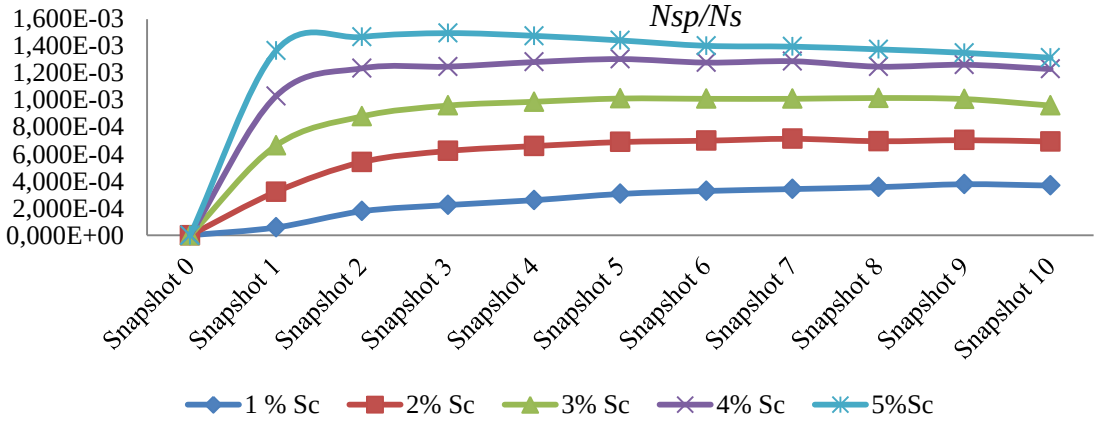


Figure 101 – Simulation VII stable precipitates normalized by the number of lattice sites.

5.8 Classical Nucleation Theory Comparative

Two parameters were used in this work to compare the Monte Carlo simulations results with the classical nucleation theory (CNT) calculations. Those parameters are the steady state nucleation rate (J^{st}) and the cluster concentration (C_{nSc}).

5.8.1 Steady State Nucleation Rate (J^{st})

Figure 102 demonstrates the steady state nucleation rate using the classical nucleation theory described earlier for various temperatures and various scandium nominal concentrations (X^0_{Sc}).

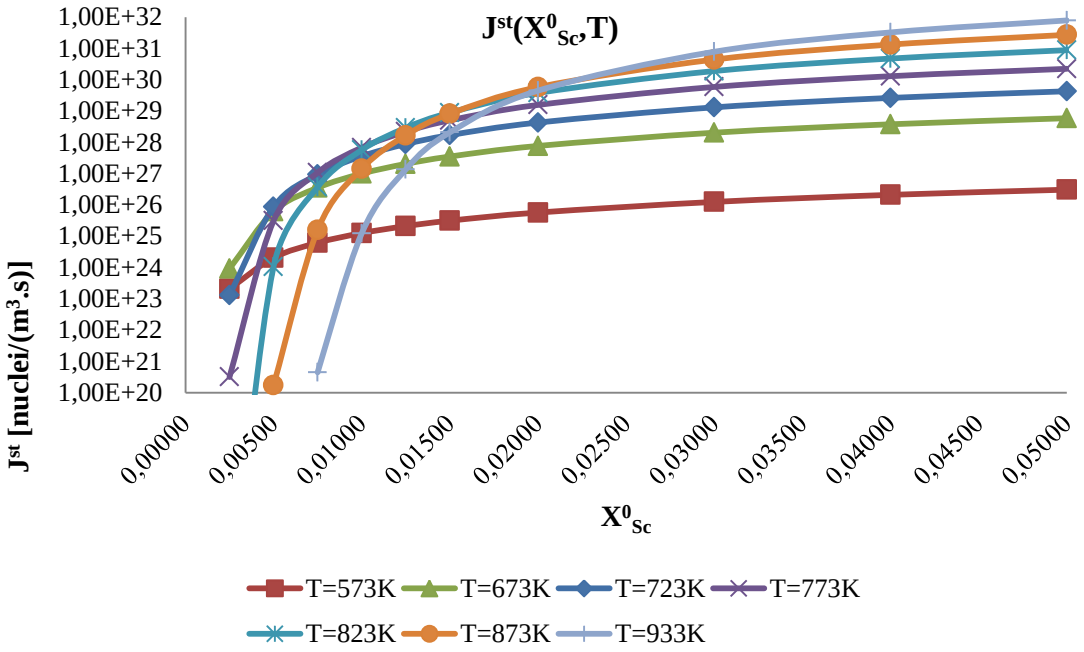


Figure 102 – Steady state nucleation rate by the classical nucleation theory.

Figure 103 and Figure 104 compare the kinetic Monte Carlo simulation with CNT for a temperature of 723.15K and 773.15K, respectively. The Monte Carlo values for J^{st} were approximated by the initial gradient of the time evolution of Nsp/Ns, where Nsp is the number of stable precipitates and Ns is the number of lattice sites. Appendix F contains all analyzed gradients. To capture the initial phase of precipitates nucleation shorter simulations were performed but with a higher sampling frequency (number of snapshots).

It is possible to observe that the J^{st} values calculated by kMC, for 0.5 and 0.75 scandium percentages, are quite bigger than those obtained by CNT. For the other scandium percentages, kMC results are in accordance with CNT.

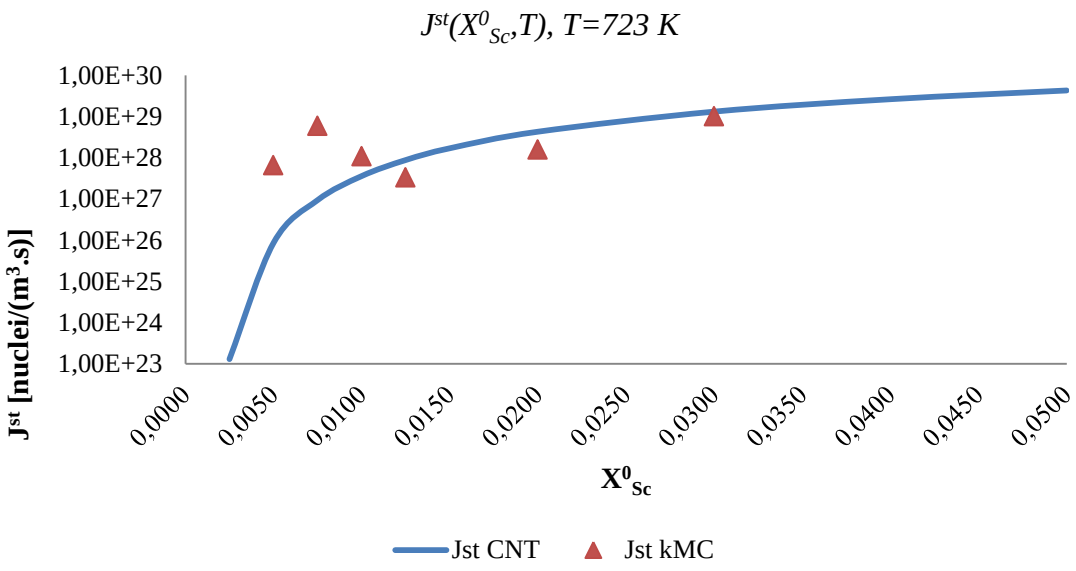


Figure 103 – Comparison between CNT and kMC for the temperature of 723K.

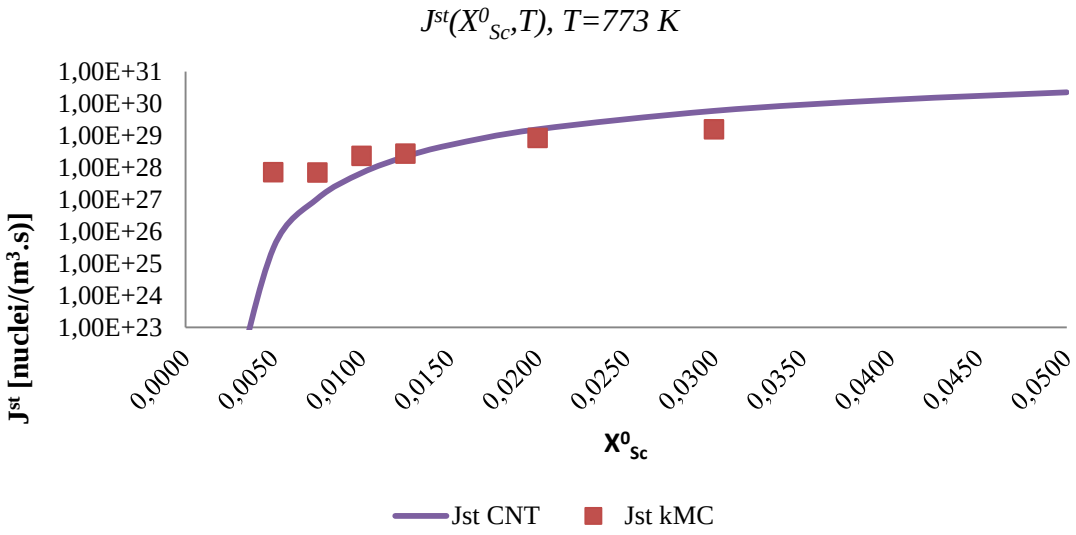


Figure 104 – Comparison between CNT and kMC for the temperature of 773K.

5.8.2 Cluster Concentration (C_{nSc})

Figure 105 exhibits the cluster size distribution by applying the classical nucleation theory, for various cluster sizes (nSc) and various scandium nominal concentrations (X^{o}_{Sc}).

Figure 106 compares the cluster concentration predictions obtained by CNT and kMC. We can conclude that cluster concentration obtained by kMC is in accordance with CNT for small cluster sizes (up to 7/8 Sc atoms). For larger cluster sizes and higher Sc concentrations there are significant differences between both methods predictions.

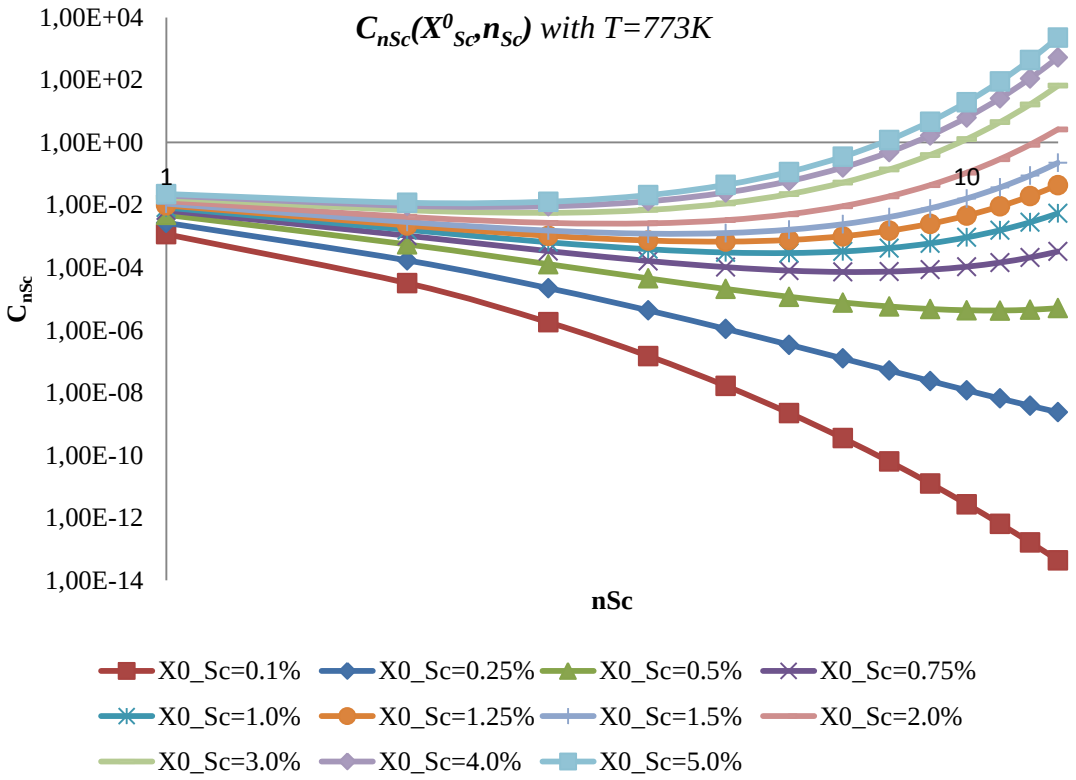


Figure 105 – Cluster size distribution by the classical nucleation theory.

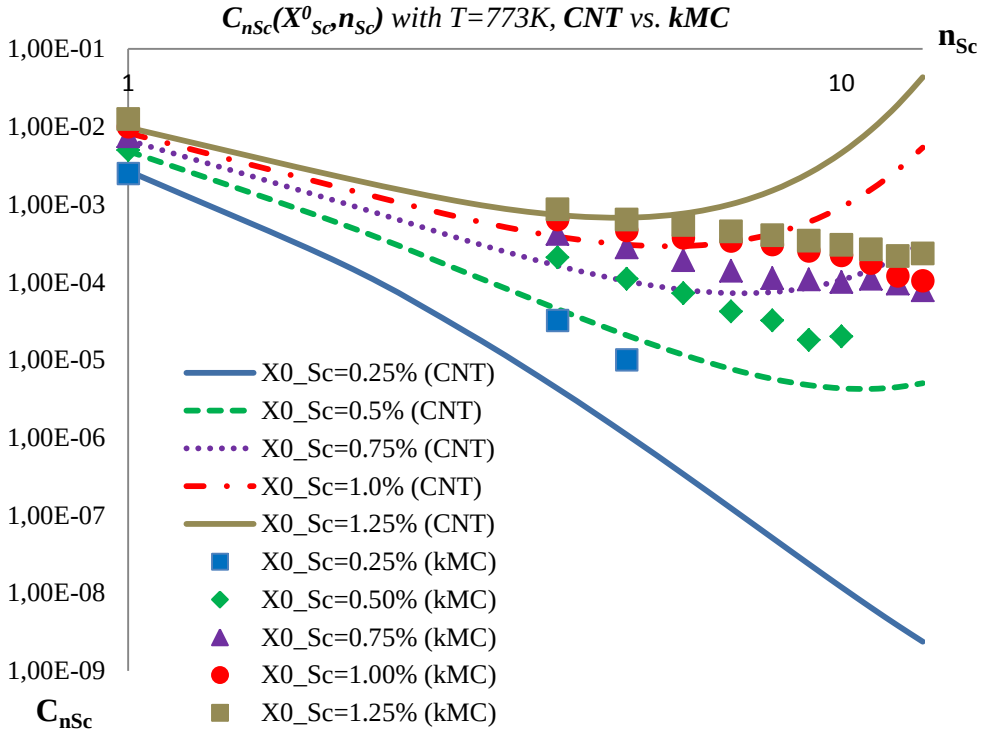


Figure 106 – Cluster size distribution comparison: kMC vs. CNT.

5.9 Comparative with Related Work

In this section we will focus in comparing the results obtained from the present work with the results of related published literature.

The work by Emmanuel Clouet deals only with $L1_2$ precipitates [Clouet2004a]. Precipitates are clusters bigger than a pre-established critical size, and he uses as critical size $n_{Sc}^{*} = 13$. Figure 107, from Emmanuel Clouet work, resumes his results for the scandium concentration in the aluminum solid solution (x_{Sc}), the precipitates average size ($\langle n_{Sp} \rangle_{Sp}$), and the number of stable precipitates normalized by the number of lattice sites (N_{sp}/N_s).

The results achieved in the present thesis are in good agreement with those reported by Emmanuel, namely:

- The precipitates average size ($\langle n_{Sp} \rangle_{Sp}$) increases as time evolves;
- The number of stable precipitates increases in the initial phase and generally stabilizes after the initial increase;
- The scandium concentration (x_{Sc}) in the aluminum solid solution reduces over the simulation time and after some time it remains constant at a residual value;
- The results obtained by the comparison of kMC simulations with CNT reveal that the steady state nucleation rate predicted by the CNT is in good agreement with our results from kMC simulations (Figure 103 and Figure 104) as well as the observations by Emmanuel Clouet (see Figure 108);
- With regard to the cluster concentration, the orders of magnitude are quite similar, as the values we obtained range between the values of 10^{-3} and 10^{-05}. Very much similar to the values presented in the work by Emmanuel Clouet, as can be seen in Figure 109.

From the work by E. Marquis and D. Seidman it is possible to conclude that, in the range of studied temperatures and aging time, the mean precipitates radius increases almost linearly over time (Figure 110). The present thesis confirms this trend, for all temperatures and scandium concentrations, except for the “difficult” simulations with 0.25% Sc. To achieve acceptable results with 0.25% Sc, we should have used very long simulations.

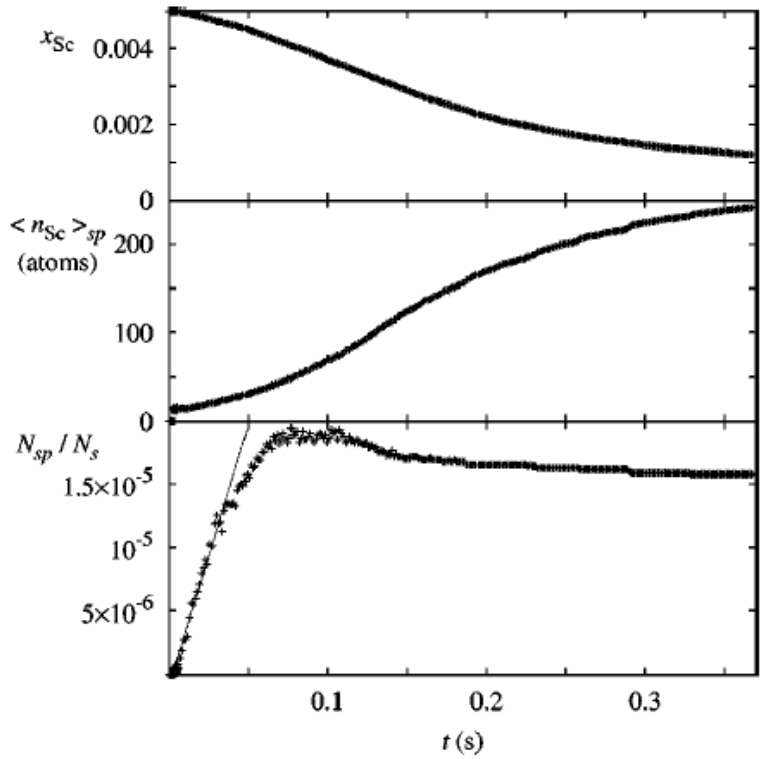


Figure 107 – Results from Emmanuel Clouet published paper [Clouet2004a].

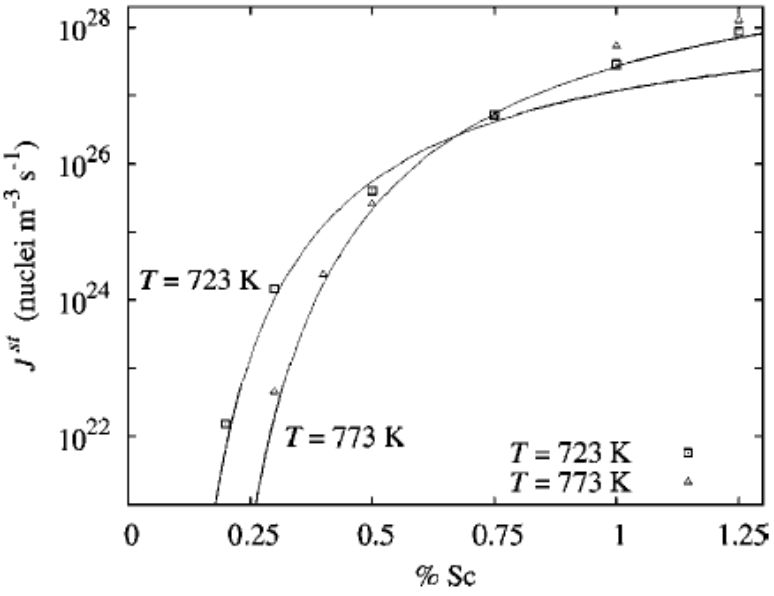


Figure 108 – Results from Emmanuel Clouet published paper [Clouet2004a].

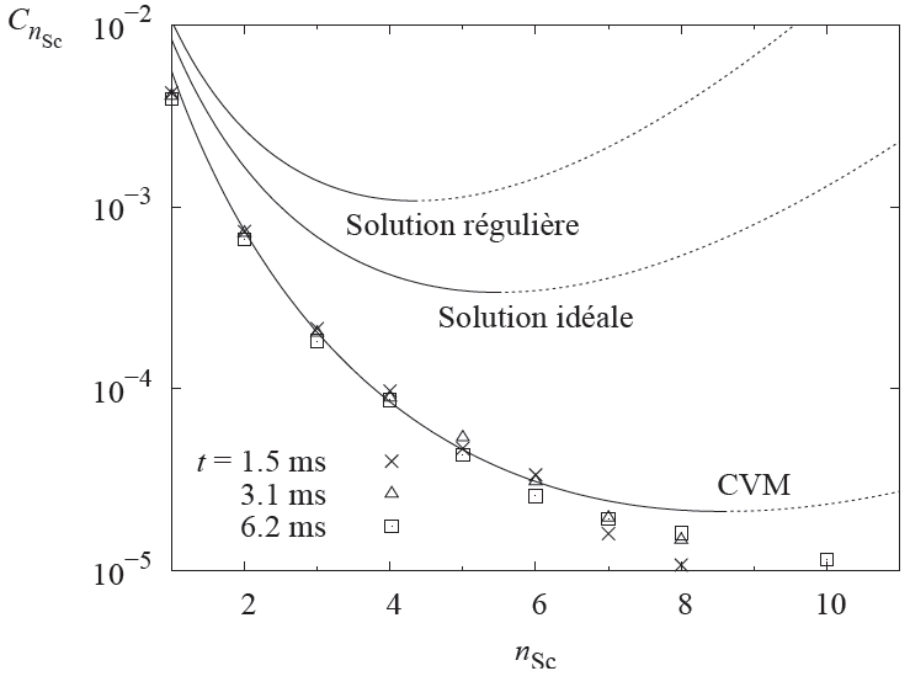


Figure 109 - Results from Emmanuel Clouet PhD thesis [Clouet2004b].

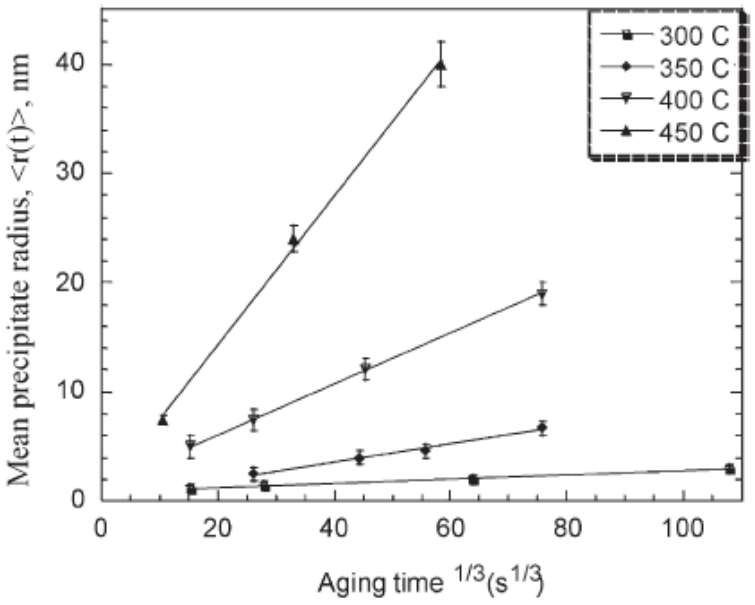


Figure 110 – Results from E. A. Marquis [Marquis2001].

In the literature it is mentioned that precipitates tend to have a Great Rhombicuboctahedron morphology (Figure 111). This means six {100} facets, twelve {110} facets, and eight {111} facets, summing a total of twenty six facets. Figure 112 shows four examples of precipitates observed in the performed simulations at four different temperatures. The precipitates Rhombicuboctahedron morphology tends to be more spherical when the temperature increases from 573 K to 873 K. This tendency is not fully perceptible in Figure 112, being easily visualized in Paraview. Anyway, it is clear that the precipitate observed at 873 K (D) is more spheric that the fully cubic precipitate observed at 573 K (A).

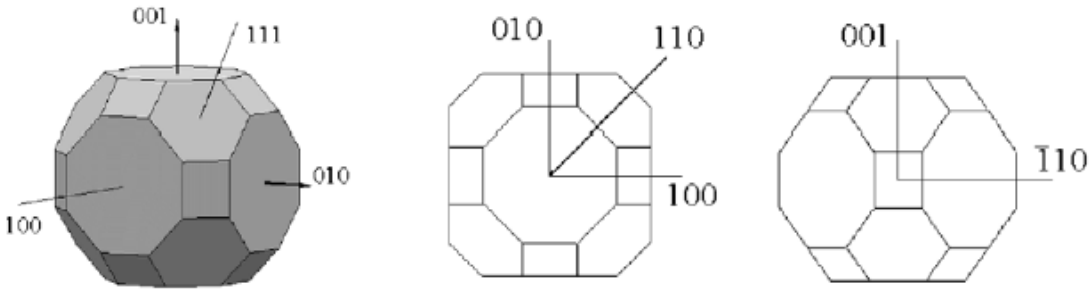
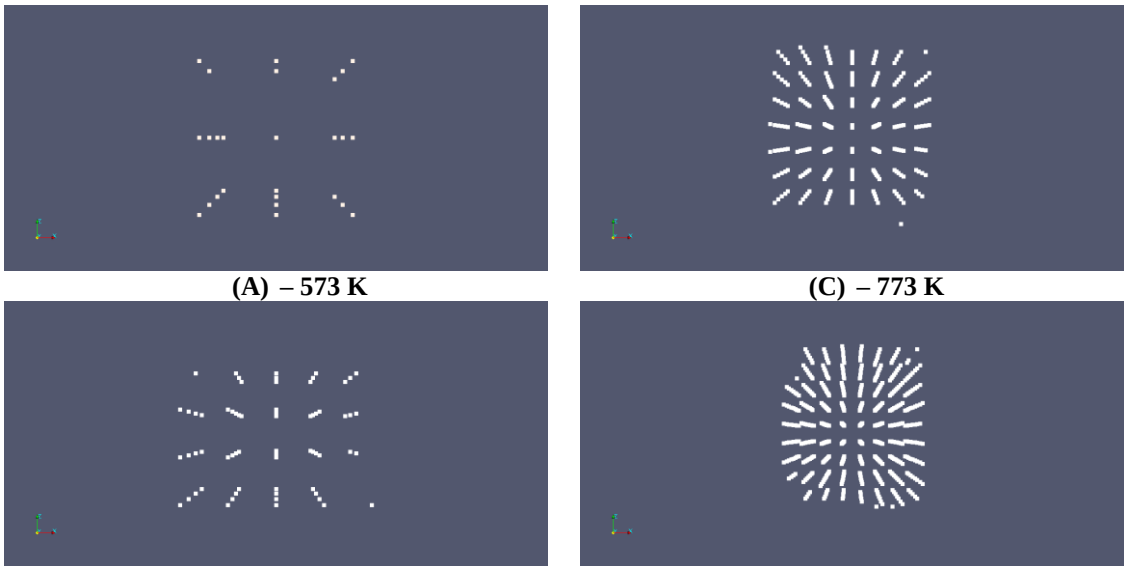


Figure 111 – Ideal Al_3Sc precipitate morphology [Marquis2001].



(A) – 573 K **(C) – 773 K**

(B) – 673 K **(D) – 873 K**

Figure 112 – Al_3Sc precipitates observed at four different temperatures.

6. Conclusions and Future Work

The first and most important achievement of the present thesis was the successful simulation of the Al_3Sc precipitation on a supersaturated aluminum solid solution, with kMC. This proves that the equations that were used correctly model the physical phenomenon of Al_3Sc precipitation. The results obtained by the kMC simulation were further improved by the application of DBSCAN, which proved to be a simple and very effective clustering algorithm. DBSCAN was a valorous aid to identify the Al_3Sc precipitates, eliminating the unclustered Sc atoms and improving visualization. By simulating with various scandium percentages, as well as temperatures, the capacity of clustering Al_3Sc precipitates maintains accurate.

The computation time mainly depends on the number of Monte Carlo steps. Simulations duration is also influenced by the technical specifications of the machines where the simulations were run. On a `compute-311-X` node of the SeARCH cluster a simulation with $5x10^{11}$ took around 8 days, and 12 days on a less performing `compute-201-X` node. Computation time does not depend significantly on the scandium percentage, the lattice size or any other parameter of the simulations. However, to achieve similar precipitation results with different scandium percentages, for lower percentages, it is necessary a higher number of MCS. As so, to achieve adequate results with percentages of scandium in order of 0.25%, it is well necessarily a much bigger number of Monte Carlo steps then $5x10^{11}$. On the other hand, proceeding with simulations in the range of $5x10^{12}$ Monte Carlo steps was not possible in the final stages of this thesis (basically a time issue).

For high supersaturations, 4 and 5% of scandium, the simulated time tend to be overestimated compared with lower supersaturations. The explanation for this situation lies on the strategy we used to obtain the real MC time, which fails in these cases, due to some very low jumping frequencies that when are inverted result in very high time steps.

Analyzing the outputs of simulations we concluded that the number of stable precipitates strongly increases in the initial phase. After this phase, as the simulations evolve over time, the number of precipitates does reduce as predicted by the theory of nucleation. Consequently the surviving precipitates do increase in size (number of atoms) and consequently in radius. The mean precipitates radius increases almost linearly over time.

The scandium concentration in the aluminum solid solution reduces over the simulation time and after some time it remains constant at a residual value. The residual value decreases with the temperature increase, due to an increased vacancy diffusion. For temperatures above 723 K, this tendency inverts, probably due to an increased facility for outer atoms leave precipitates at high temperatures. Naturally, the scandium

concentration in precipitates increases in the initial phase of simulation and remains constant after that.

The number of precipitates normalized by the number of lattice sites increases rapidly on the initial phase of the simulation and then decreases slightly during the rest of the simulation. To capture this initial phase we had to run shorter simulations with a higher sampling frequency.

To analyze the effect of the lattice size on the simulations results, simulations were carried out with two sizes: 50x50x50 and 100x100x100. The number of Monte Carlo steps was $5x10^{11}$ and 10^{12}, respectively. Considering that a 100x100x100 is eight times bigger than 50x50x50, it was used a higher simulation time with the former lattice. The results confirmed the expectations: a higher lattice needs a higher time to achieve similar precipitation results. For example, comparing simulations performed with the two different lattice sizes, but both with 1% of scandium, a temperature of 873K, and $5x10^{11}$ MCS, the conclusions are: the simulation with 50x50x50 reaches a configuration with (i) a smaller normalized number of precipitates ($1.2x10^{-5}$ *vs.* $3.05x10^{-5}$), (ii) much less precipitates (6 *vs.* 122), (iii) a larger size (655 *vs.* 198) and (iv) a larger radius (16.06 *vs.* 11.28). As the normalized number of precipitates is the best reference for the comparison, we concluded that the smaller lattice (50x50x50) is more effective to study precipitation in a reasonable computation time. This justifies why we adopted the 50x050x50 lattice in most of our work.

Temperature has a profound influence on the evolution of the precipitation simulation. As the classical nucleation states and the graphics do prove, the steady state nucleation rate rises with the rise of the temperature. The simulations undertaken do show this tendency.

As the Sc concentration rises, so does the initial number of precipitates. As previously mentioned, to clearly observe precipitation with low supersaturations (for ex. 0.25% Sc), a huge number of Monte Carlo steps would be necessary.

Another interesting finding is related to the dependency between the morphology of the precipitates and the temperature. As mentioned in [Marquis2001], we also observed that the precipitates morphology becomes Great Rhombicuboctahedron with the temperature increase. At 573 K, precipitates are small and almost cubic, while at 873 K they are much bigger and tend to be spheric.

As said before, the DBSCAN algorithm reveals adequate in the role of identifying, visualizing and measuring (size, radius, and shape) the precipitates embedded in the Monte Carlo output data. It is well true that it was not undertaken a profound study using different DBSCAN parameters other than *eps*=4.1, *minPts*=3 and *minClusterSize*=13. These parameter values supported the obtained results from the Monte Carlo simulations. To compare the cluster concentration predictions obtained by CNT and kMC, the kMC simulations achieved much better results using *eps*=6.1 instead of 4.1.

Comparing kMC and CNT leads to the following observations. The J^{st} values calculated by kMC, for 0.5 and 0.75 scandium percentages, are quite bigger than those obtained by CNT and the other scandium percentages, kMC results are in accordance with CNT. For the cluster concentration obtained by kMC, it is in accordance with CNT for small cluster sizes (up to 7/8 Sc atoms). For larger cluster sizes and higher Sc concentrations there are significant differences between both methods predictions. DBSCAN tuning revealed to have a major impact in the small clusters analysis required by the cluster concentration calculation with kMC data.

The results achieved in the present thesis are very much in good agreement with those reported by Emmanuel Clouet. It is possible to point out that the precipitates average size increase over time follows the same tendency; the scandium concentration in the aluminum solid solution reduction during the simulation follows the same tendency; the comparison between kMC and CNT are very much similar. Although the present thesis is very much inspired in the work by Emmanuel Clouet, we went deeper in the evaluation of the influence of all the parameters involved in simulation: lattice size, temperature, scandium concentration, number of MC steps, and the technique used in cluster identification and measuring. We also tried also strategies to accelerate the simulation.

The migration from the Matlab version to a C language version was vital for the success of the Monte Carlo simulation and subsequently the DBSCAN analysis. With the implementation in C language, it was possible to obtain significant speedups, as well as flexibility. The possibility to use an open source platforms revealed vital. The attempt to implement the kMC method with several threads of execution, through OpenMP, brought no advantages compared to the optimized and sequential version of KMC. This is due to the sequential nature of the problem of precipitation, which has a single point of evolution at each stage. DBCAN was not a challenging problem from the computational point view, only for high scandium concentrations, the analysis time became a concern. But even in these cases, the clustering analysis took only around 1 hour. Contrary to the implementation of kMC, in the implementation of DBSCAN calculating *a priori* the distance between all pairs of atoms proved prohibitive in terms of space occupied in memory for the highest scandium concentration (5%). In kMC, compute *a priori* all the neighborhoods brought speedup to the simulation.

Some features of ParaView made it an interesting choice for visualization and even analysis: it supports the three formats (VTK, PDB, XYZ) we used as output of kMC, it is open source and based on a popular framework (VTK), supports parallelism to handle huge files and complex tasks, handles smoothly complex 3D visualization, and has a "small feature" that revealed vital to our work: it synchronizes, in terms of spatial coordinates, several opened windows; this is crucial to compare several snapshots of the same simulation. The potential of ParaView is quite a hand full, and that for this thesis, was not fully explored, focusing mainly on the visualization aspects.

Future Work

We will point out some improvements of the work done in the present thesis and future directions of investigation.

The method used to convert MC steps to simulated time must be improved in order to solve the overestimation detected in several simulations with high supersaturations.

During the most intense phase of kMC simulations in the SeARCH cluster, some long simulations were aborted before its scheduled completion, having lost several days of simulation. To minimize this problem, it would be necessary to implement support for resuming a simulation from data saved at a particular snapshot. Although this functionality have been thought, comparing its implementation effort with the resulting benefits, it was decided this functionality was not a priority to achieve the goals of the thesis. After completing the thesis, it should be noted that this functionality should be implemented for a safer use of the application.

Another improvement of the realized work could be the replacement of DBSCAN by another clustering algorithm to obtain a better approximation of the cluster concentration and as so to compare kMC with CNT.

A possible field for future research would be the exploration of parallelization techniques for the kMC simulation. Due to the sequential nature of the precipitation problem, a hypothesis would be to use multiple vacancies and use parallel simulations, each with a sub-lattice. Examples of algorithms that follow this strategy are the optimistic synchronous relaxation (OSR) and the semi-rigorous synchronous sublattice (SL) [Nandipati2009]. These approaches will have to deal with two critical issues: correct the excessive vacancy concentration and synchronize the parallel instances of the asynchronous kMC simulation.

Another fundamental future research topic would be to extend Monte Carlo method to simulate also ternary alloys, as for example an aluminum-magnesium-scandium alloy (Al-Mg-Sc), an aluminum-scandium-silicon alloy (Al-Sc-Si) or an aluminum-scandium-zirconium alloy (Al-Sc-Zr), besides the binary alloy aluminum-scandium (Al-Sc). We focused on the elements aluminum and scandium, but it is also interesting to simulate other elements in the binary or ternary form. If such case, it would be necessary to upgrade the DBSCAN analyzer in order to include the third element of the ternary alloy.

Finally, another interesting issue for future work would be to compare the kMC results not just with the classical nucleation theory but also with other simulation methods, such as cluster dynamics [Clouet2005b] [Lae2004].

Bibliography

[Ahmad2003] Z. Ahmad | The Properties and Application of Scandium-Reinforced Aluminum | JOM | 2003

[Amar2006] Jacques G. Amar | The Monte Carlo Method in science and Engineering | Computing in Science & Engineering | IEEE | 2006

[Ashby1998] Michael F. Ashby and David R. H. Jones. | Engineering Materials 2. An introduction to microstructure, processing and design. Second Edition. | Butterworth-Heinemann | 1998

[Barralis2010] Jean Barralis, Gerard Maeder | Prontuário de Metalurgia | Fundação Calouste Gulbenkian | 2ª Ed. | 2010

[Binkele2003] P. Binkele & S. Schmauder | An atomistic Monte Carlo simulation of precipitation in a binary system | 2003

[Binkele2004] P. Binkele, P. Kizler, S. Schmauder | Atomistic Monte Carlo simulations of the diffusion of P and C near grain boundaries in bcc iron | 30th MPA-Seminar in conjunction with the 9th German-Japanese Seminar, Stuttgart, October 6 and 7 2004

[Bombac2010] David Bombac, Goran Kugler | Precipitation in Alloys: A kinetic Monte Carlo and Class Model Study | World Journal of Engineering | 2010

[Bréchet2006] Yves Bréchet, Georges Martin | Nucleation problems in metallurgy of the solid state: recent developments and open questions | C. R. Physique 7 (2006) 959-976 | 2006

[Chen2005] Z. Chen, Z. Zheng, S. P. Ringer | Effect of Small Addition of Sc on the Microstructural Evolution in Al-15Ag Alloy | J. Mater. Sci. Technol., Vol. 21 No. 5 | 2005

[Clouet2004a] E. Clout, M. Naster, and C. Sigli. Nucleation of Al_3Zr and Al_3Sc in aluminum alloys: From kinetic Monte Carlo simulations to classical theory. Physical Review B 69, 064109, 2004.

[Clouet2004b] E. Clout | Separation de Phase dans les Alliages Al-Zr-Sc: du Saut des atomes à la Croissance de Précipités Ordonnés| These: Grade de Docteur | École Centrale de Artes et Manufactures - École Centrale Paris| 2004

[Clouet2005a] E. Clouet et al | Precipitation in Al-Zr-Sc alloys: a comparison between Kinetic Monte Carlo, cluster dynamics and classical nucleation theory | Solid-solid Phase Transformation in Inorganic Materials | TMS | 2005

[Clouet2005b] E. Clouet, A. Barbu, L. Laé, and G. Martin | Precipitation kinetics of Al_3Zr and Al_3Sc in aluminum alloys modeled with cluster dynamics | Acta Materialia (53) 2313-2325, Elsevier | TMS | 2005

[Clouet2006] E. Clouet et al | Kinetic Monte Carlo Simulations of Precipitation | Advanced Engineering Materials 2006, 8, No. 12| 2006

[Clouet2007] E. Clouet, and M. Nastar. Classical nucleation theory in ordering alloys precipitating with $L1_2$ structure. Physical Review B 75, 132102, 2007.

[Clouet2010] E. Clouet & F. Soisson | Atomic simulations of diffusional phase transformations | C. R. Physique 11 (2010) 266-235 | 2010

[Deschamps2010] Alexis Deschamps, Michel Perez | Mesoscopic modeling of precipitation: A tool for extracting physical parameters of phase transformations in metallic alloys | C. R. Physique 11 (2010) 236-244 | 2010

[EAA2002] The Aluminium Automotive Manuel | European Aluminium Association | Version 2002 | 2002

[Escke2011] Sven K. Esche | Monte Carlo Simulations of Grain Growth in Metals| Applications of Monte Carlo Method in Science and Engineering | 2011

[Ester1996] Martin Ester et al | Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise | Published in Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining (KDD-96) | 1996

[Geuser2010] Frédéric De Geuser, Brain M Gable, Barry C Muddle | CALPHAD based kinetic Monte Carlo simulation of clustering in binary Al-Cu alloy | Philosophical Magazine & Philosophical Magazine Letters | 2010

[Hatch1984] John E. Hatch | Properties and Physical Metallurgy | American Society for Metals | 1984

[Hin2009] C. Hin | Kinetics of heterogeneous grain boundary precipitation of Ni_3Al in nickel alloy | Journal of Physics D: Applied Physics | 2009

[Holm2010] Holm, E.A. and S.M. Foiles | How Grain Stops: A Mechanism for Grain Growth Stagnation in Pure Materials |Science, 2010 328 (5982) | 2010

[Knipling2010] K. E. Knipling, R. A. Karnesky, C. P. Lee, D. C. Dumand, D. N. Seidman | Precipitation evolution in Al-0.1Sc, Al-0.1Zr and Al-0.1Sc-0.1Zr (at%) alloys during isochronal aging | Acta Materialia 58 (2010) 5184-5195 | 2010

[Jacobs1999] M. H. Jacobs | Precipitation Hardening | TALAT – Training in Aluminum Application Technologies | EAA – European Aluminum Association | 1999

[Kratzer2009] P. Kratzer | Monte Carlo and kinetic Monte Carlo methods – a tutorial | Institute for Advance Simulation | 2009

[Landau2012] David P. Landau, Kurt Binder | A Guide to Monte Carlo Simulations in Statistical Physics. Second Edition. | Cambridge University Press | 2000

[Lae2004] L. Lae, P. Guyot, C. Sigli | Cluster dynamics in AlZr and AlSc alloys | Materials Forum Volume 28 | Institute of Materials Engineering Australasia Ltd | 2004

[Lundstrom] Lundstrom et al | Investigative Tools: Theory, Modeling and Simulation |

[Marceau2008] Ross K. W. Marceau | Design in Light Alloys by Understanding Solute Clustering Processes During the Early Stages of Age Hardening in Al-Cu-Mg Alloys | A submitted thesis for the degree of Doctor of Philisophy | University of Sydney | 2008

[Marquis2001] E. A. Marquis & D. N. Seidman | Nanoscale Structural Evolution of Al3Sc Precipitates in Al(Sc) Alloys | Acta mater. 49(2001) 1909-1979

[Martin2005] G. Martin & F. Soisson | Kinetic Monte Carlo Method to Model Diffusion Controlled Phase Transformations in the Solid State | Handbook of Materials Modeling, 2223-2248 | 2005 Springer

[Nandipati2009] G. Nandipati, Y. Shim, J.G. Amar, A. Karim, A. Kara, T.S.Rahman, O. Trushin | Parallel kinetic Monte Carlo simulations of Ag(111) island coarsening using a large database | J. Phys Condens. Matter., 21(8):084214 | 2009.

[Naukin1965] O. Naumkin, V. Terekhova, Ye Savitskiy | Phase Diagram and Properties of the Alloys of the Aluminium-Scandium System | National Aeronautics and Space Administration Washington | July 1966

[Okamoto1991] H. Okamoto | Section III: Phase Diagram Updates | Journal of Phase Equilibria Vol. 12 No. 5 | 1991

[Perez2008] M. Perez, M. Dumont, D. Acevedo-Reyes | Implementation of classical nucleation and growth theories for precipitation | Acta Materialia 56 (2008) 2119-2132 | 2008

[Plimpton2009] S. Plimpton et al | Crossing the Mesoscale No-Man´s Land via Paralelel Kinetic Monte Carlo | Sandia Report | SAND2009-6226 | 2009

[Raabe1998] Dierk Raabe | Computational Materials Science | The Simulation of Materials Microstructures and Properties | WILEY-VCH | 1998

[Röyset] Jostein Röyset | Scandium in aluminum alloys overview: physical metallurgy, properties and applications | Metallurgical Science and Technology | Hydro Aluminium R&D Sunndal, N-6600 Sunndalsöra, Norway

[Røyset2005] J. Røyset, N. Ryum | Scandium in aluminum alloys | International Materials Reviews 2005 Vol. 50 No. 1 | 2005

[Sarker2009] T. Sarker, S. Dias, A. Mandal | A study of computer-based simulations for Nano-Systems and their types | Published in Proceedings of 2nd national conference on nano-materials and nanotechnology (21-23 December, 2009), University of Luknow, U.P. | 2009

[Schmauder2002] S. Schmauder & P. Binkele | Atomistic computer simulation of the formulation of Cu-precipitates in steels | Computational Materials Science 24 (2002) 42-53 | 2002

[Smallman1999] R. E. Smallman & R. J. Bishop | Modern Physical Metallurgy and Materials Engineering – Science, Process, Applications | Sixth Edition | Butterworth-Heinemann

[Soisson2007] Frédéric Soisson, Chu-Chun Fu | Cu-precipitation kinetics in α-Fe from atomistic simulations: vacancy-trapping effects and Cu-cluster mobility.| Physical Review B, 76, 214102 (2007)| 2007

[Soisson2010] F. Soisson, C. S. Becquart, N. Castin, C. Domain, L. Malerba, E. Vincent | Atomistic Kinetic Monte Carlo studies of microchemical evolutions driven by diffusion processes under irradiation | Journal of Nuclear Materials 406 (2010) 55-67 | 2010

[Styer2007] Daniel F. Styer | Statistical Mechanics | Department of Physics and Astronomy Oberlin College, Oberlin, Ohio 44074-1088| 2007

[Vincent2008] E. Vincent, C. S. Becquart, C. Pareige, P. Pareige, C. Domain | Precipitation of the FeCu system: a critical review of atomic Kinetic Monte Carlo simulations | Journal of Nuclear Materials 373 (2008) 387-401 | 2008

[Voter2005] Arthur F. Voter | Introduction to the Kinetic Monte Carlo Method | Radiation Efects in Solids | 2005

[Webelements2012] http://www.webelements.com/scandium/physics.html | 2012

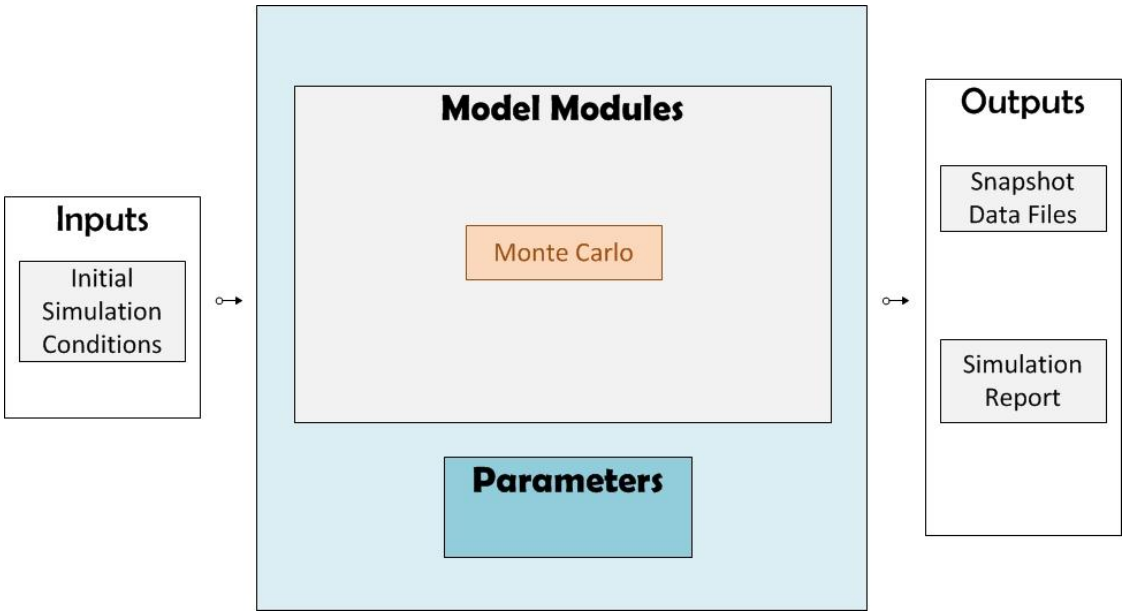
Appendices

A. Nomenclature and Units

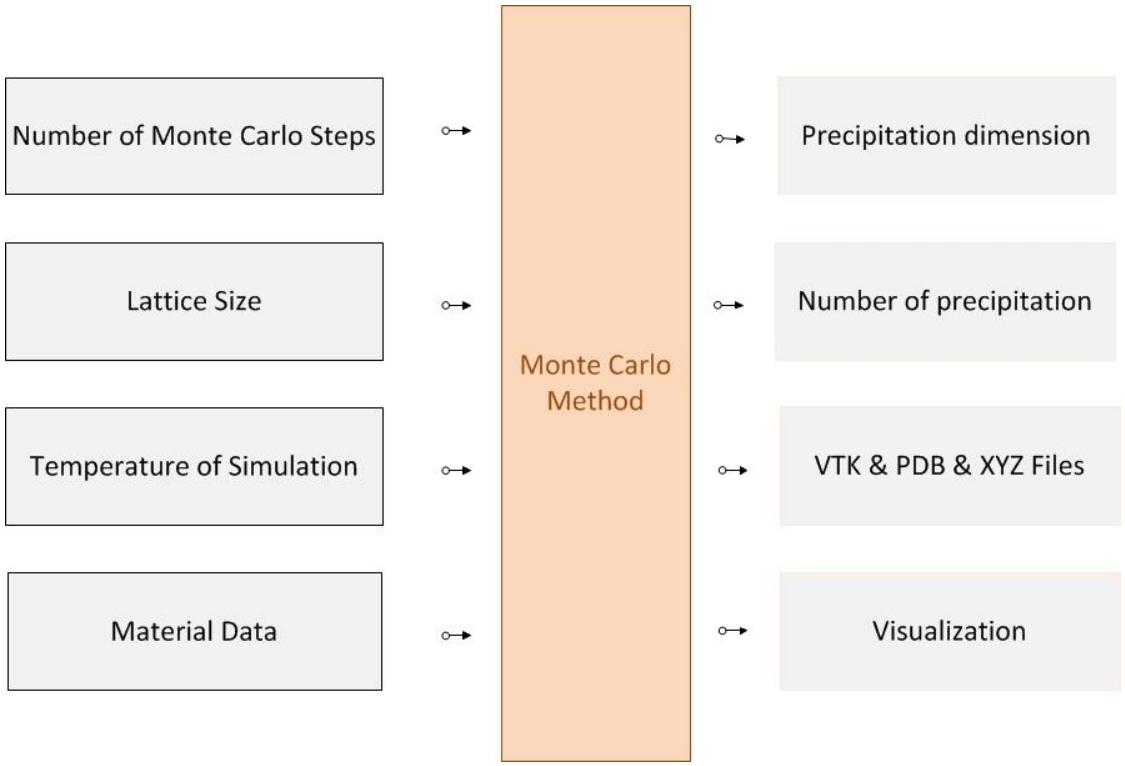
Al	Aluminum	
a	Lattice parameter	
Sc	Scandium	
Nm	Nanometer	10^{-9}m
Å	Angstrom	10^{-10}m
Γ	Vacancy exchange frequency	
ΔE	Activation Energy	
wt%	Weight	
D	Diffusion	
α, β, γ	Axel angles	
ω	Mixing energy / order energy	
E_{coh}	Cohesive energy	
E_v	Vacancy formation energy	
N_v	Number of vacancies	
N_s	Number of sites	
N_a	Number of atoms	
Z	Number of nearest neighbors	
DBSCAN	Density Based Spatial Clustering of Applications with Noise	
Eps	Predefined radius in DBSCAN clustering	
MinPts	Predefined minimum number of points in DBSCAN clustering	
MCS	Monte Carlo steps	
J^{st}	Steady state nucleation rate	nuclei/(m^3.s)
C_{nSc}	Cluster concentration	
X^{o}_{Sc}	Scandium nominal concentration	

B. Flowcharts of the Monte Carlo Implementation

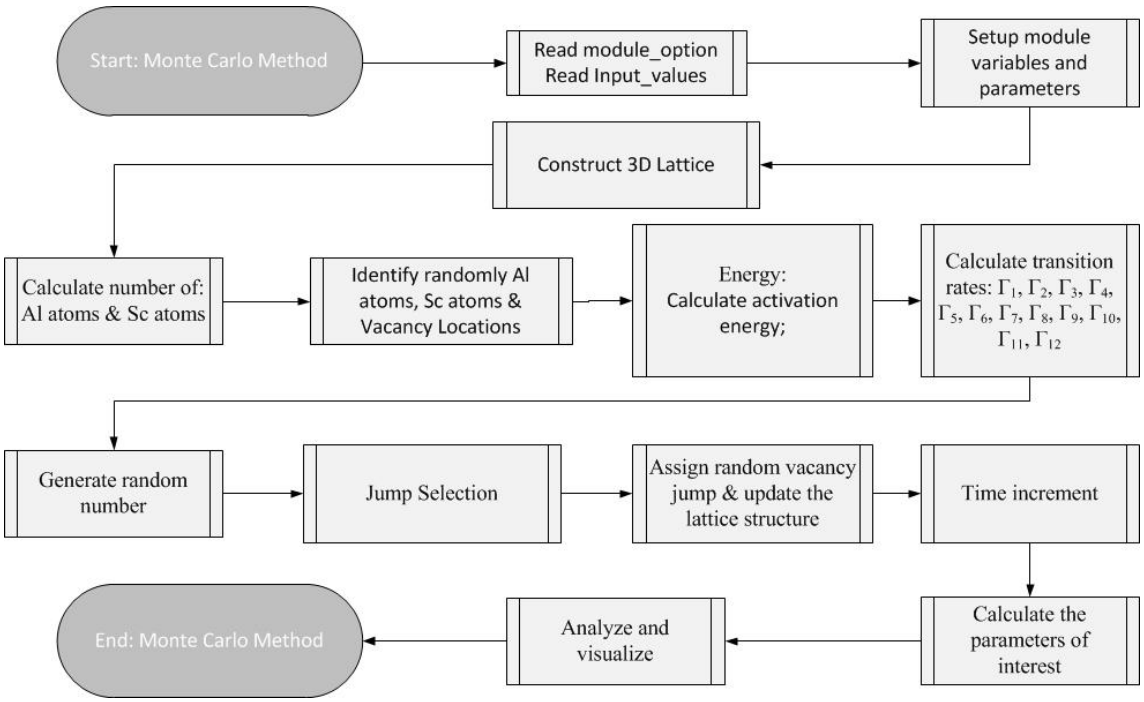
"Main" Module Interface



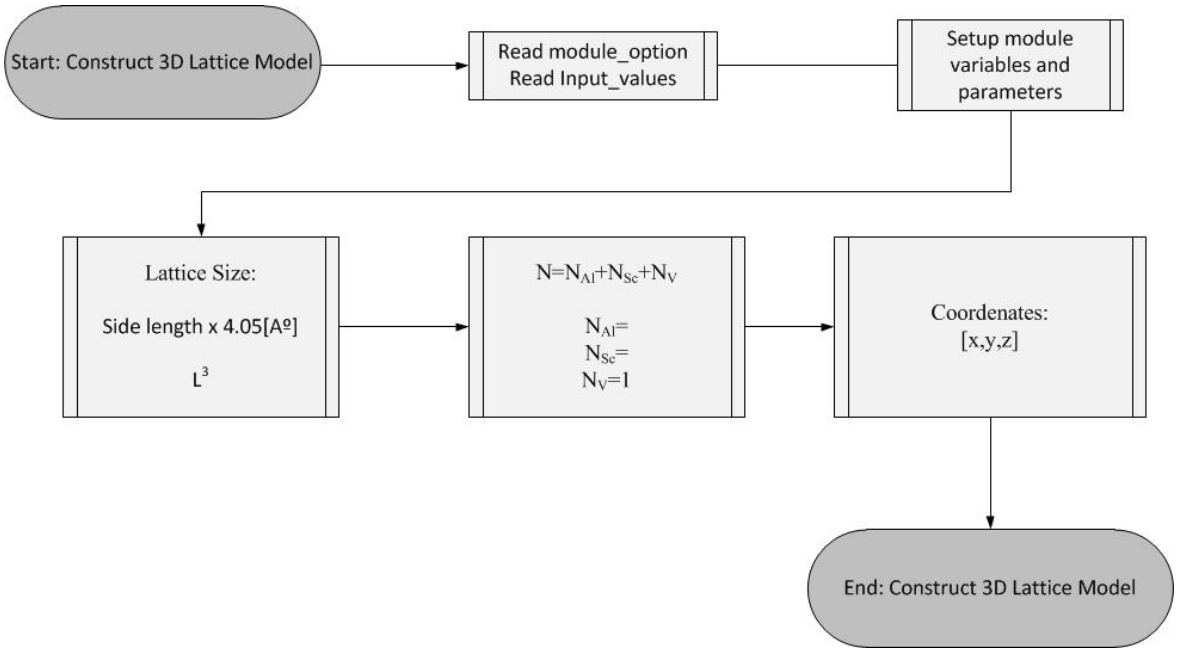
"Monte Carlo" Module Interface



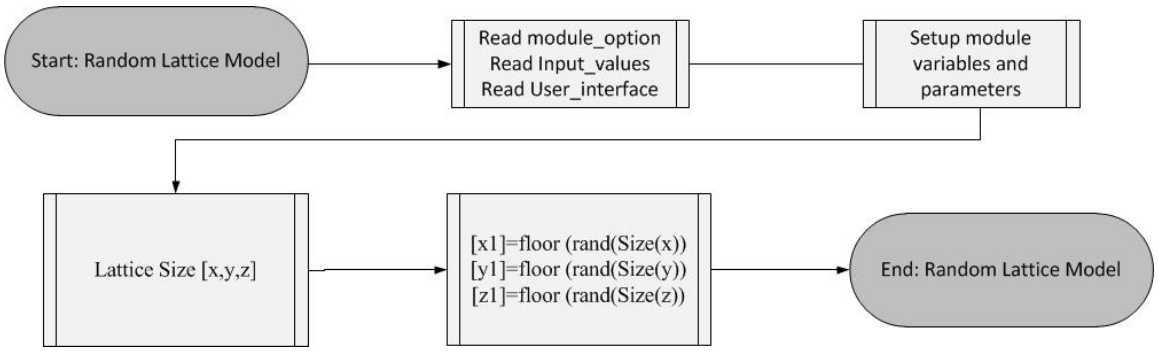
“Monte Carlo” Module Flowchart



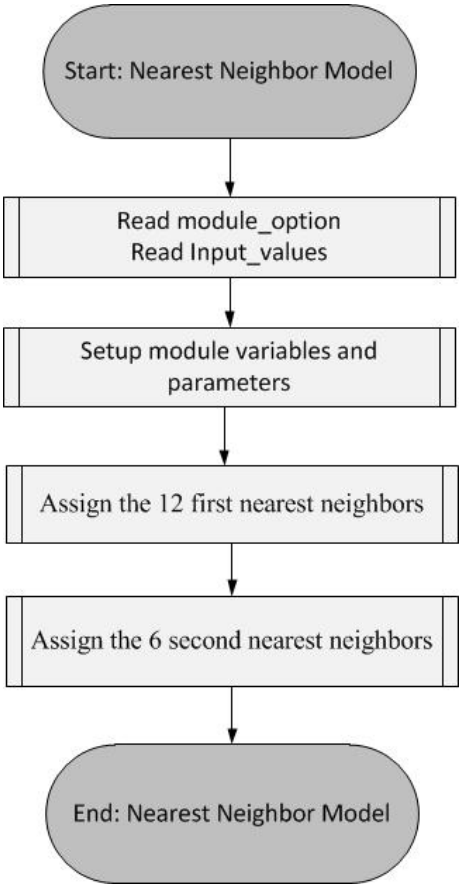
“Construct 3D FCC Lattice” Module Flowchart



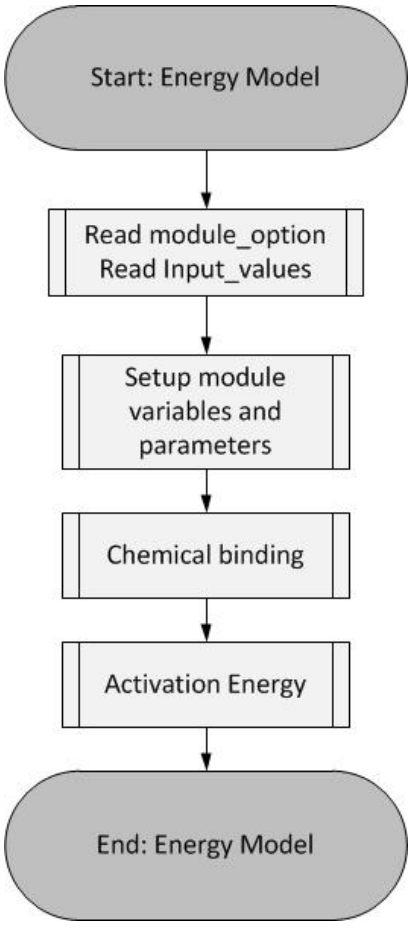
“Random Vacancy Lattice Site” Module Flowchart



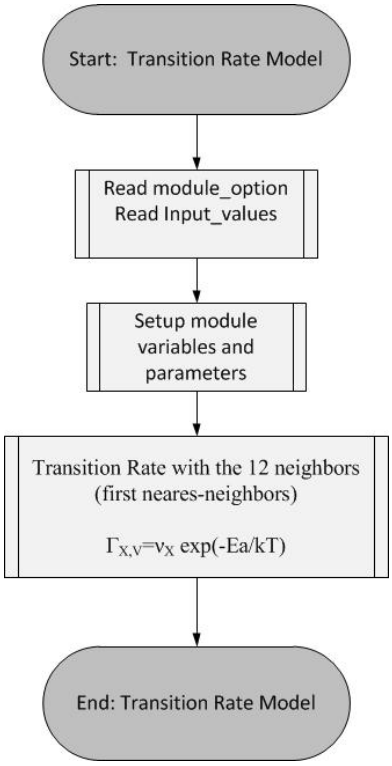
“Nearest Neighbor” Module Flowchart



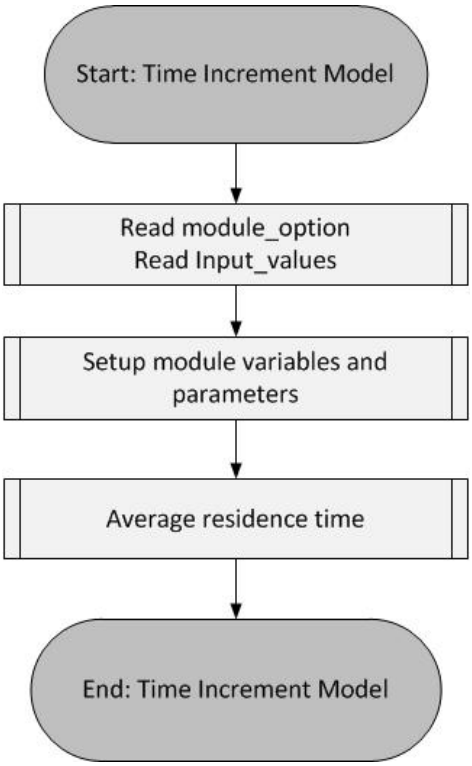
“Activation Energy” Module Flowchart



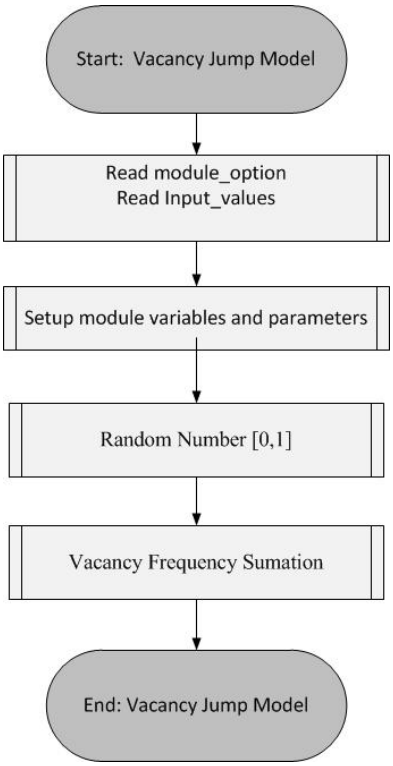
"Vacancy Exchange Frequency" Module Flowchart



"Real Time Calculation" Module Flowchart



"Random Vacancy Selection" Module Flowchart



C. Report of the Monte Carlo Series I of Simulations

```
**********************************************************************
Simulation Report
Made in: Fri Sep 28 04:55:30 2012

MSc Thesis 2012
**********************************************************************

**********************************************************************
Simulation material information
----------------------------------------------------------------------
Al-Al 1st nearest neighbor pair effective energy: -0.560000 [ eV ]
Sc-Sc 1st nearest neighbor pair effective energy: -0.650000 [ eV ]
Al-Sc 1st nearest neighbor pair effective energy: -0.740664 [ eV ]
Al-Sc 2nd nearest neighbor pair effective energy:  0.083837 [ eV ]
Al-Al 2nd nearest neighbor pair effective energy:  0.000000 [ eV ]
Sc-Sc 2nd nearest neighbor pair effective energy:  0.000000 [ eV ]
Al-Vacancy 1st nearest neighbor pair effective energy: -0.222000 [eV]
Sc-Vacancy 1st nearest neighbor pair effective energy: -0.757000 [eV]
Aluminum Saddle point energy: -8.219000  [ eV ]
Scandium Saddle point energy: -9.434000  [ eV ]
Aluminum attempt frequency:   1.3600e+14 [ eV ]
Scandium attempt frequency:   4.0000e+15 [ eV ]

**********************************************************************
Simulation input information
----------------------------------------------------------------------
Number of Aluminum atoms:          474999
Number of Scandium atoms:          25000
Number of Vacancy sites:           1
Number of Monte Carlo Steps:       500000000000
Number of snapshots saved to files:10
Temperature Applied:               873.150 [ Kelvin ]

**********************************************************************
Simulation output information
----------------------------------------------------------------------
Snapshot  1 simulated time: 0 d : 0 h : 0 m : 1.6004582622771860e+01 s
Snapshot  2 simulated time: 0 d : 0 h : 0 m : 2.9866777929521806e+01 s
Snapshot  3 simulated time: 0 d : 0 h : 0 m : 4.1981044547432020e+01 s
Snapshot  4 simulated time: 0 d : 0 h : 0 m : 5.3210892713519968e+01 s
Snapshot  5 simulated time: 0 d : 0 h : 1 m : 3.9012490713031140e+00 s
Snapshot  6 simulated time: 0 d : 0 h : 1 m : 1.3978703557246936e+01 s
Snapshot  7 simulated time: 0 d : 0 h : 1 m : 2.3684391384834811e+01 s
Snapshot  8 simulated time: 0 d : 0 h : 1 m : 3.3337252549986736e+01 s
Snapshot  9 simulated time: 0 d : 0 h : 1 m : 4.3001841336461155e+01 s
Snapshot 10 simulated time: 0 d : 0 h : 1 m : 5.2390045661691694e+01 s
Simulated time:             0 d : 0 h : 1 m : 5.2390045661691694e+01 s
Simulation duration:       12 d : 2 h : 42 m : 41 s
**********************************************************************
```

D. Report Relative to the Application of DBSCAN to a Final Snapshot of Series I of Simulations

```
**********************************************************************
Simulation Analysis Report
Made in: Fri Sep 28 13:39:01 2012

Machine: 4 Cores of type Intel64 Family 6 Model 42 Stepping 7,
GenuineIntel running Windows_NT
MSc Thesis 2012
**********************************************************************


**********************************************************************
Simulation information
----------------------------------------------------------------------
Number of Aluminum atoms:           474999
Number of Scandium atoms:           25000
Number of Vacancy sites:            1
Scandium percentage:                5.000
Number of Monte Carlo Steps:        500000000000
Number of snapshots saved to files:10
Temperature Applied:                873.150 [ Kelvin ]


**********************************************************************
Clustering Analysis input information
----------------------------------------------------------------------
Radius used to define atom's neighborhood (DBSCAN): 4.10
Minimum number of neighbors that turns an atom into a core atom
      of a cluster (DBSCAN): 3
Minimum number of atoms to consider a cluster as valid: 13
Input VTK file:    AlSc_50x50x50_5x10E11_873K_percent5_Sc_10.vtk
Generated VTK file: AlSc_50x50x50_5x10E11_873K_percent5_Sc_10_dbs.vtk
Configuration file: AlSc_50x50x50_5x10E11_percent5_config.txt


**********************************************************************
Clustering Analysis output information
----------------------------------------------------------------------
Percentage of Sc atoms in Al solid solution:     0.157 %
Percentage of Sc atoms in precipitates:          4.843 %
Number of identified clusters:                   33
Average size among all clusters   (in atoms):    733
Average radius among all clusters (in Angstrom): 17.40
```

ClusterID	Cluster Size (atoms)	Clusters Radius (A)
0	656	17.327
1	390	14.569
2	709	17.781
3	1597	23.309
4	933	19.485
5	530	16.138
6	801	18.519
7	868	19.022
8	367	14.277
9	382	14.469
10	817	18.642
11	1826	24.373
12	807	18.566
13	562	16.456
14	298	13.319
15	564	16.476
16	361	14.199
17	972	19.753
18	292	13.229
19	1373	22.164
20	1046	20.242
21	410	14.814
22	641	17.194
23	822	18.680
24	771	18.285
25	1383	22.217
26	1055	20.300
27	893	19.203
28	597	16.791
29	75	8.409
30	624	17.040
31	523	16.066
32	269	12.873

E. Snapshots Configurations for a Series I Simulation

This appendix presents images of the snapshots configuration, before and after applying the DBSCAN clustering algorithm, in the case of a series I simulation with 1% Sc.

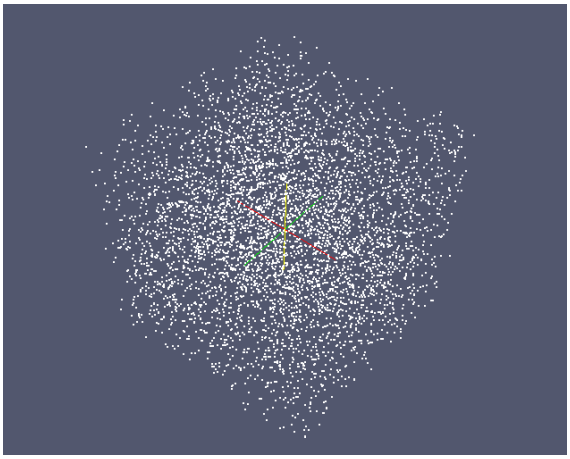


Figure 113 – Initial configuration.

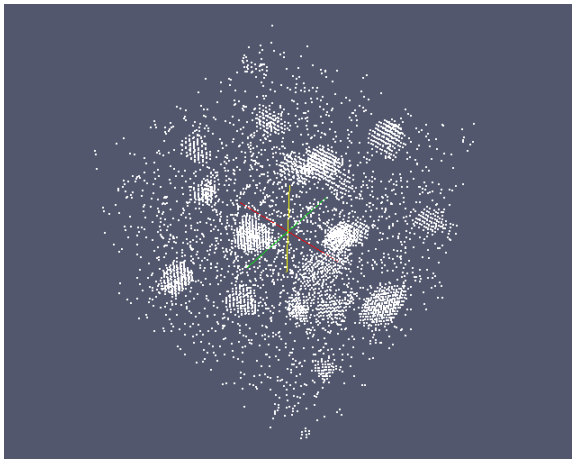


Figure 114 – Snapshot 1.

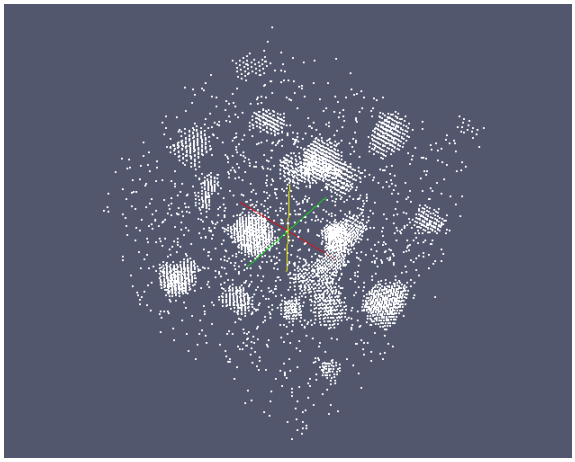


Figure 115 – Snapshot 2.

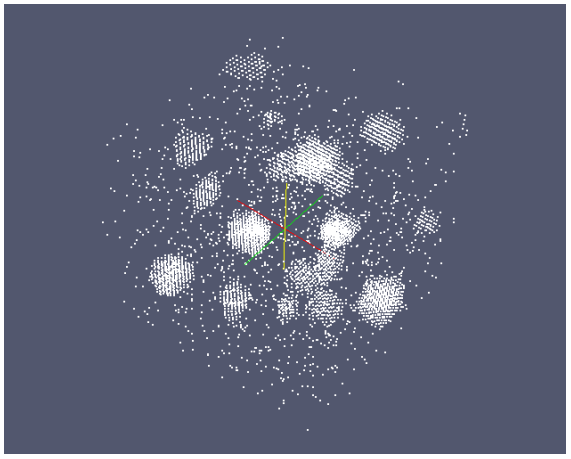


Figure 116 – Snapshot 3.

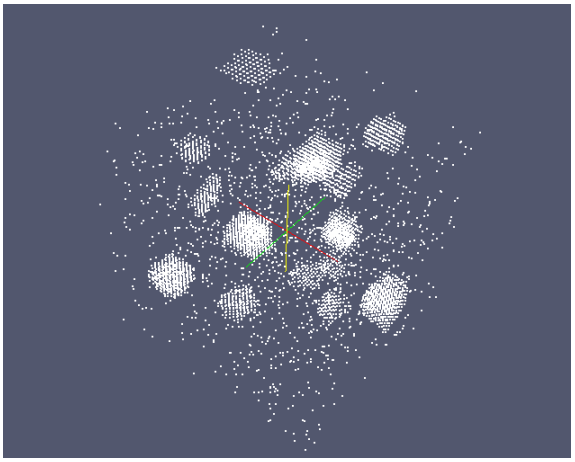


Figure 117 – Snapshot 4.

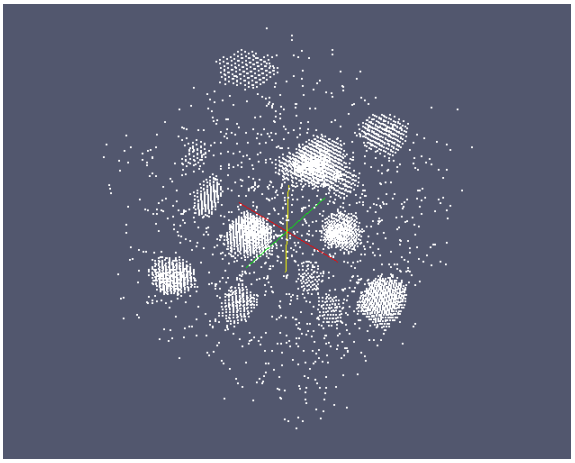


Figure 118 – Snapshot 5.

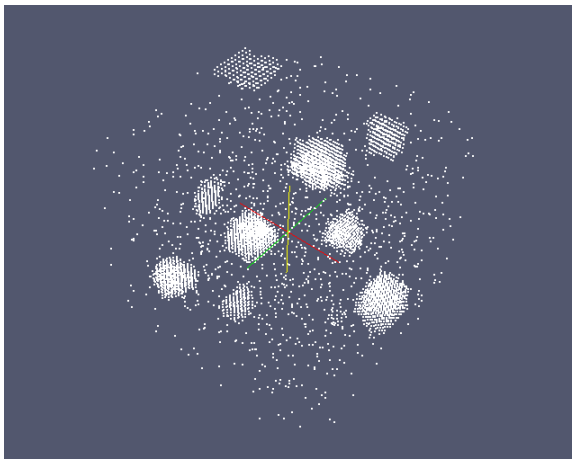


Figure 119 – Snapshot 6.

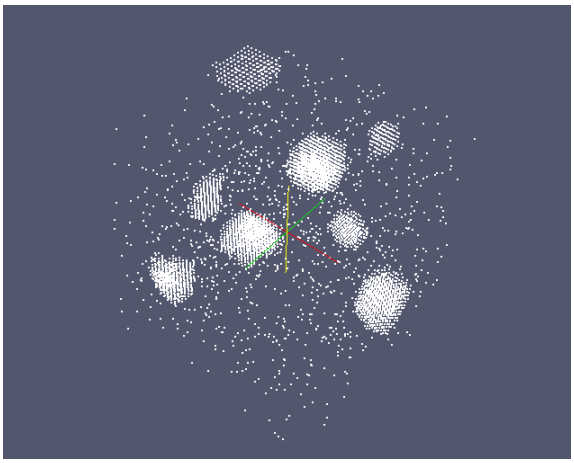


Figure 122 – Snapshot 9.

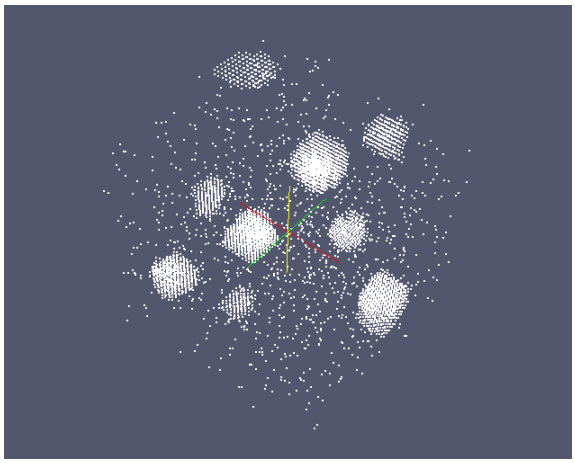


Figure 120 – Snapshot 7.

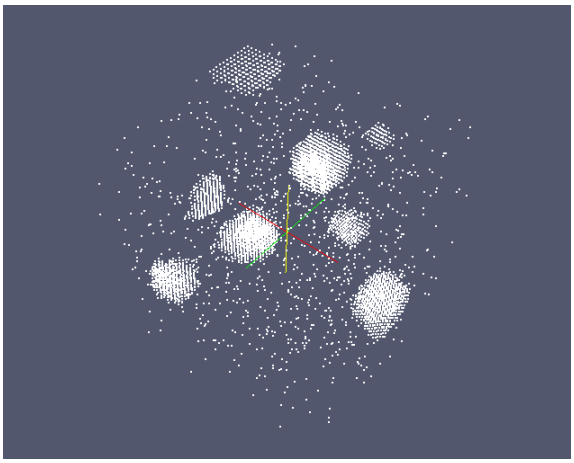


Figure 123 – Snapshot 10.

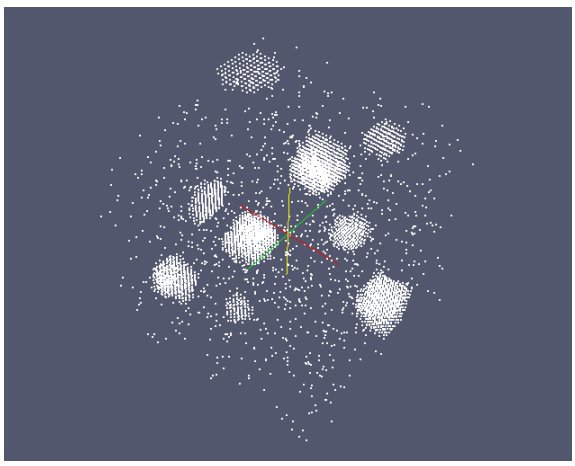


Figure 121 – Snapshot 8.

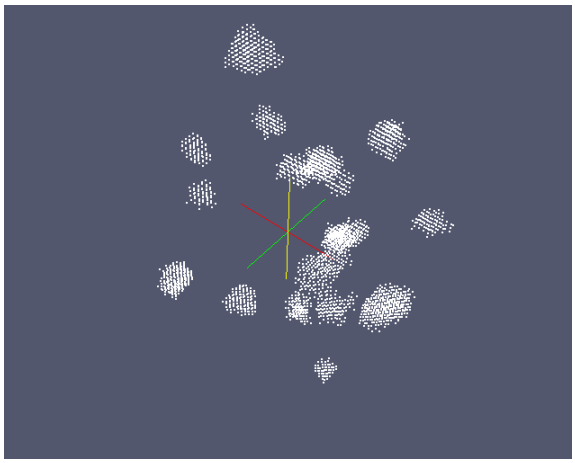


Figure 124 – Snapshot 1 after clustering.

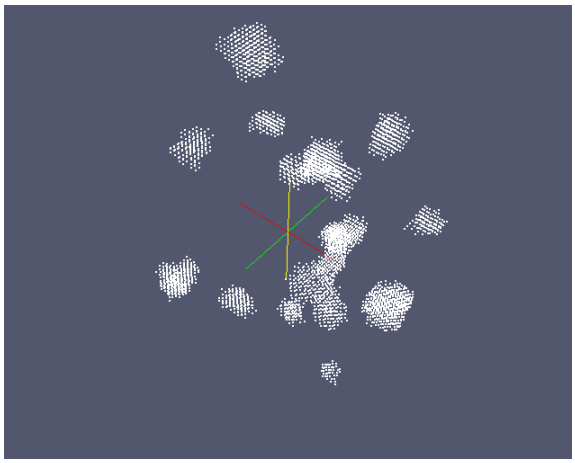


Figure 125 – Snapshot 2 after clustering.

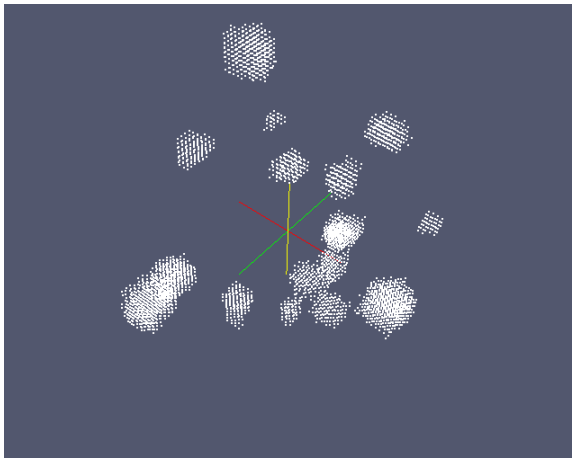


Figure 126 – Snapshot 3 after clustering.

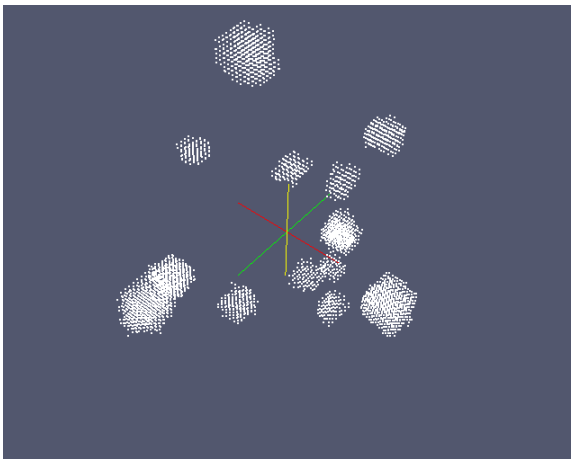


Figure 127 – Snapshot 4 after clustering.

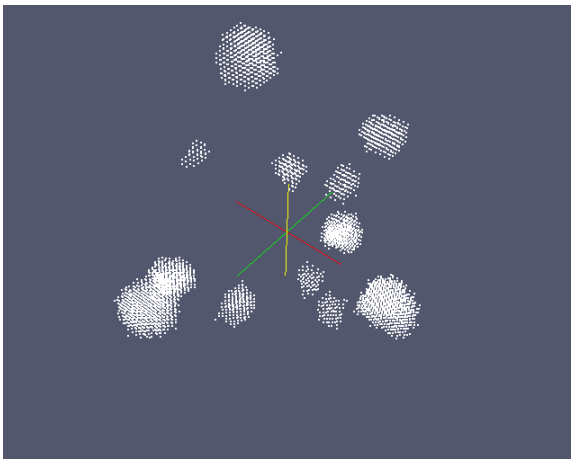


Figure 128 – Snapshot 5 after clustering.

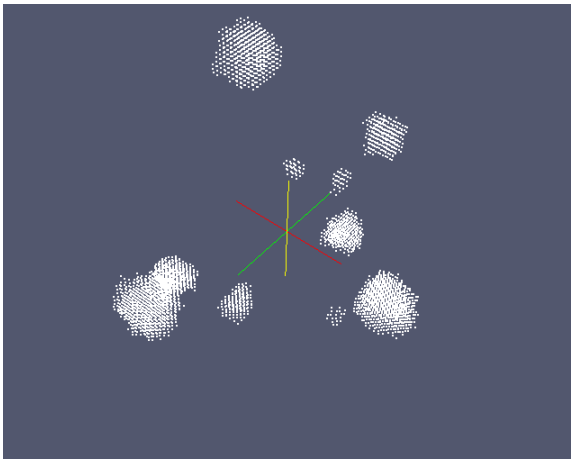


Figure 129 – Snapshot 6 after clustering.

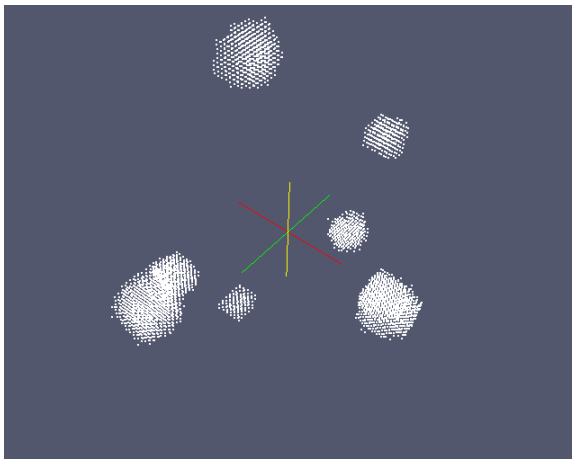


Figure 130 – Snapshot 7 after clustering.

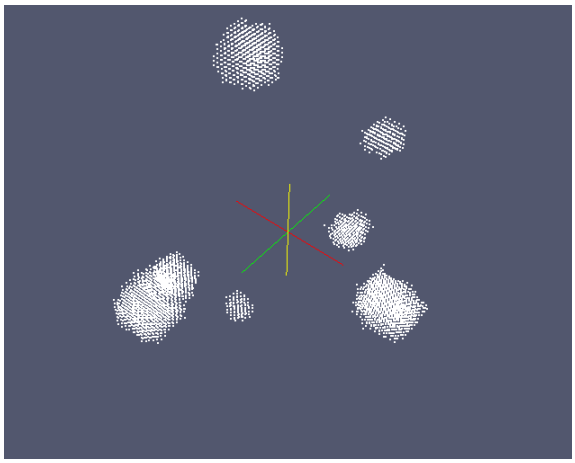


Figure 131 – Snapshot 8 after clustering.

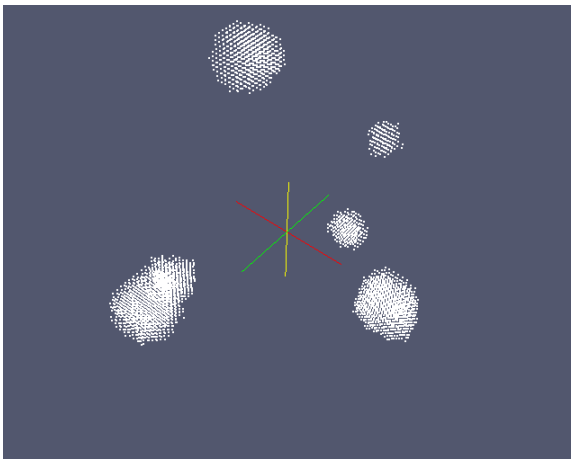


Figure 132 - Snapshot 9 after clustering.

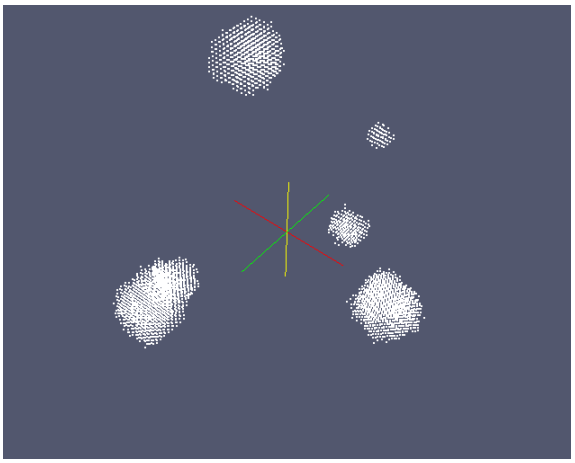


Figure 133 – Snapshot 10 after clustering.

F. Steady State Nucleation Rate by Kinetic Monte Carlo

i. Temperature of 723 K

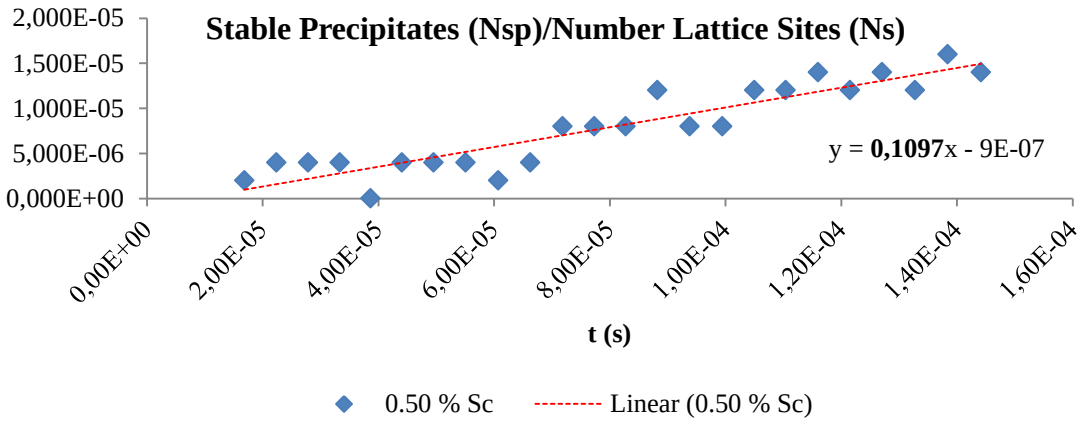


Figure 134 – Gradient calculation for a 0.50 % Sc simulation at 723.15K.

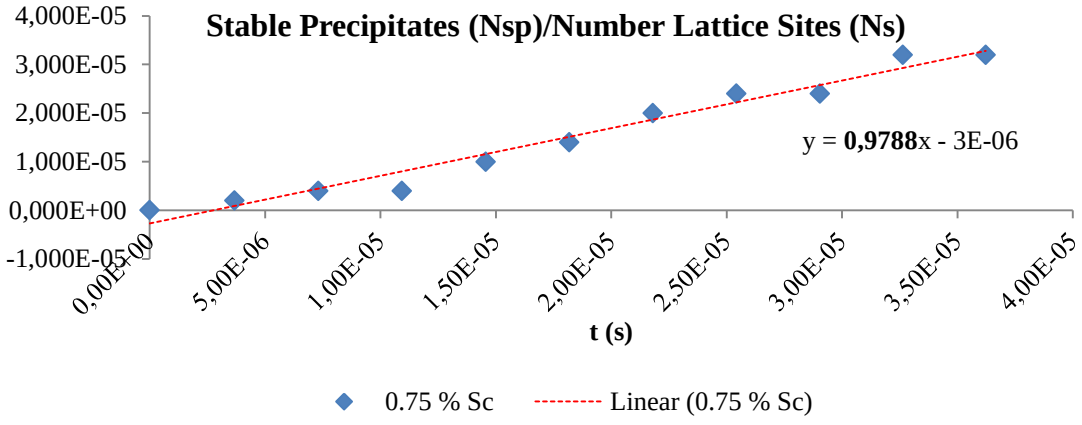


Figure 135 – Gradient calculation for a 0.75 % Sc simulation at 723.15K.

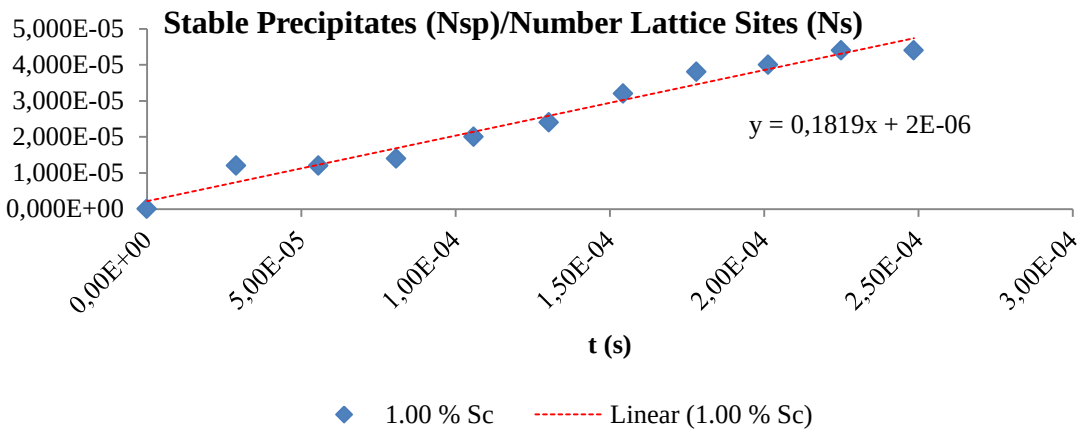


Figure 136 – Gradient calculation for a 1.00 % Sc simulation at 723.15K.

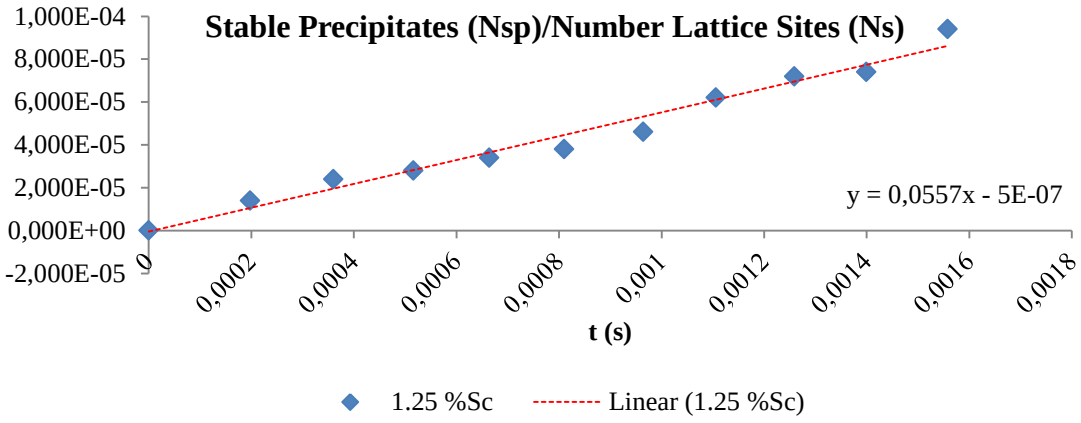


Figure 137 – Gradient calculation for a 1.25 % Sc simulation at 723.15K.

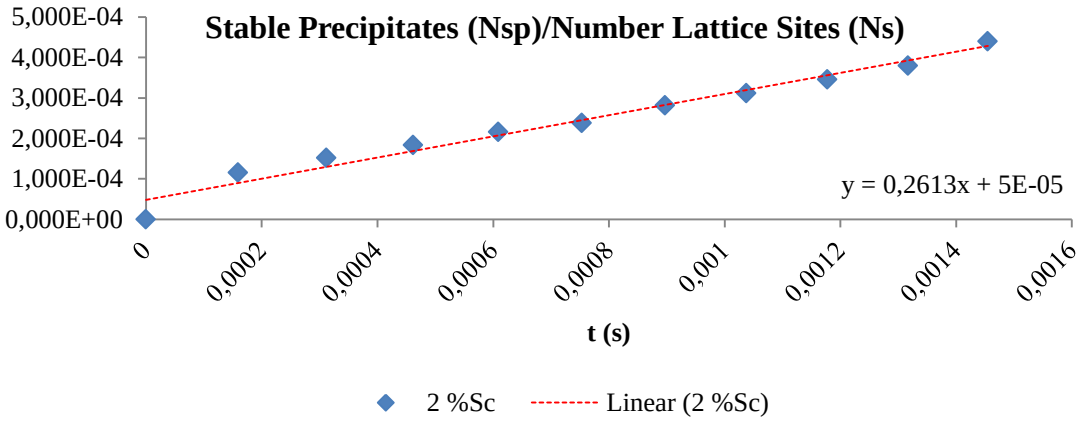


Figure 138 – Gradient calculation for a 2 % Sc simulation at 723.15K.

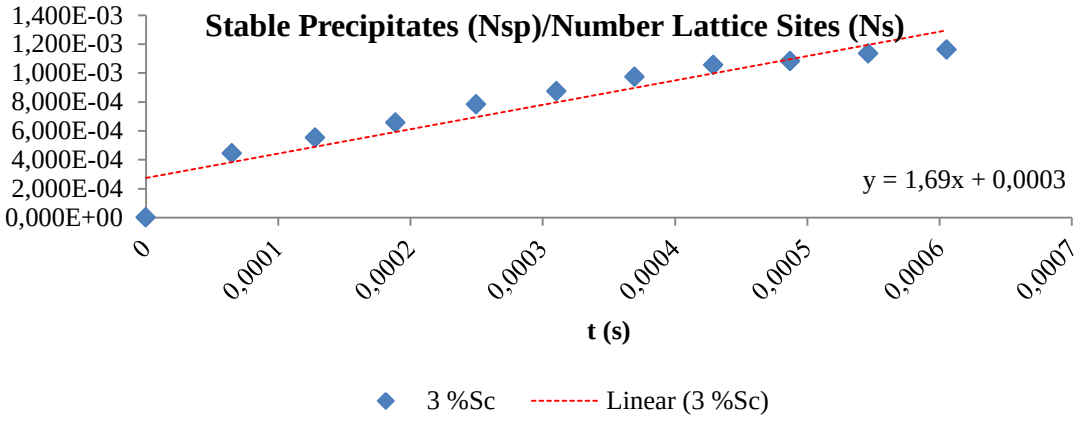


Figure 139 – Gradient calculation for a 3 % Sc simulation at 723.15K.

ii. Temperature of 773 K

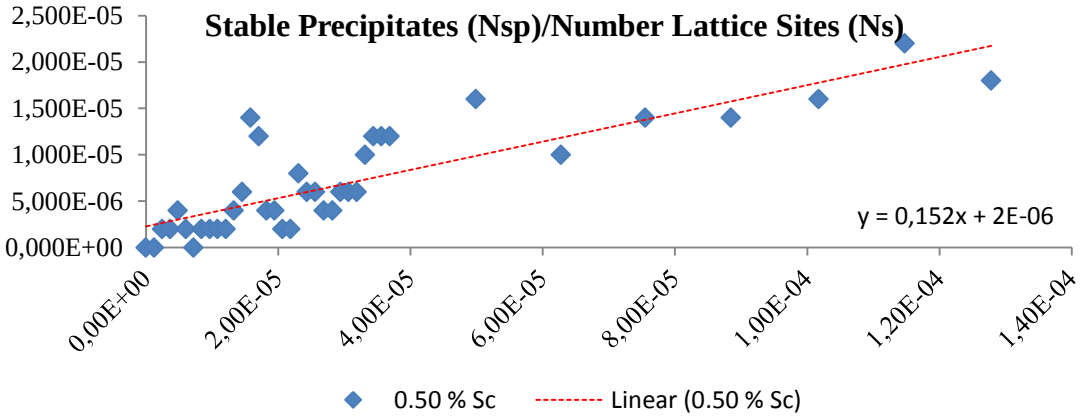


Figure 140 – Gradient calculation for a 0.50 % Sc simulation at 773.15K.

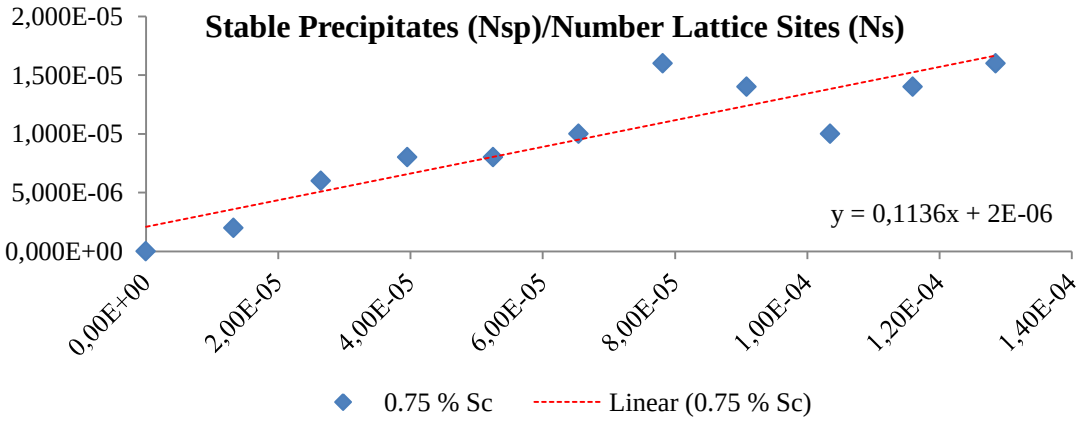


Figure 141 – Gradient calculation for a 0.75 % Sc simulation at 773.15K.

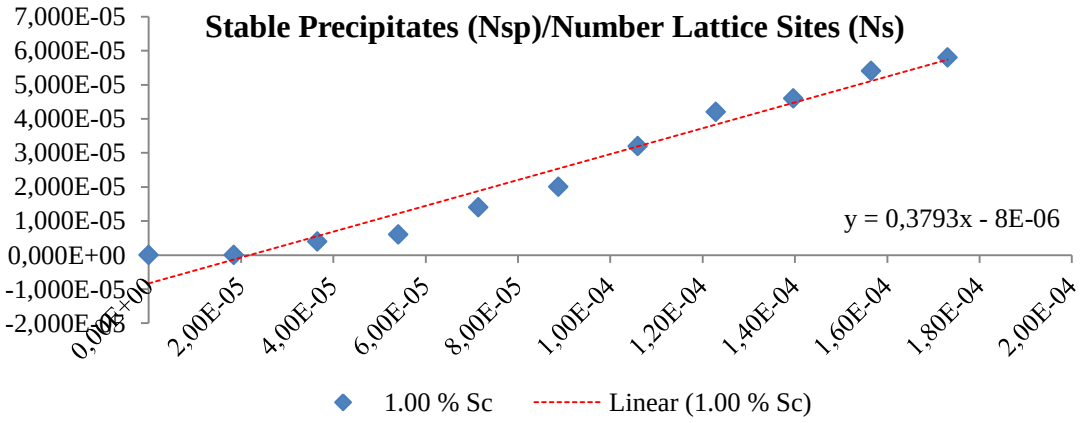


Figure 142 – Gradient calculation for a 1.00 % Sc simulation at 773.15K.

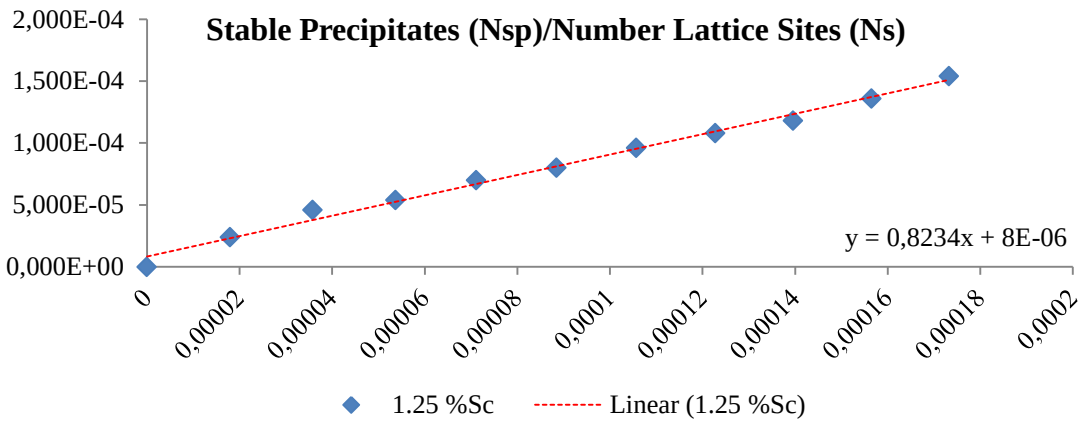


Figure 143 – Gradient calculation for a 1.25 % Sc simulation at 773.15K.

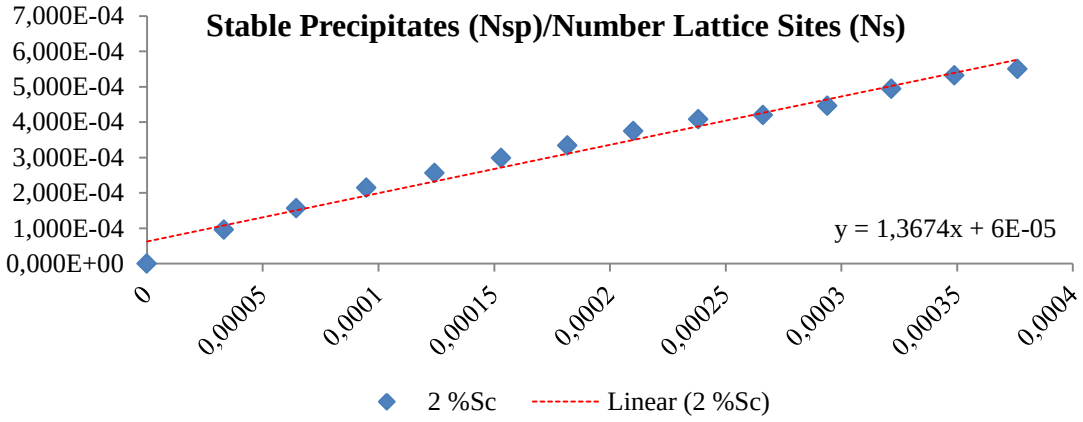


Figure 144 – Gradient calculation for a 2 % Sc simulation at 773.15K.

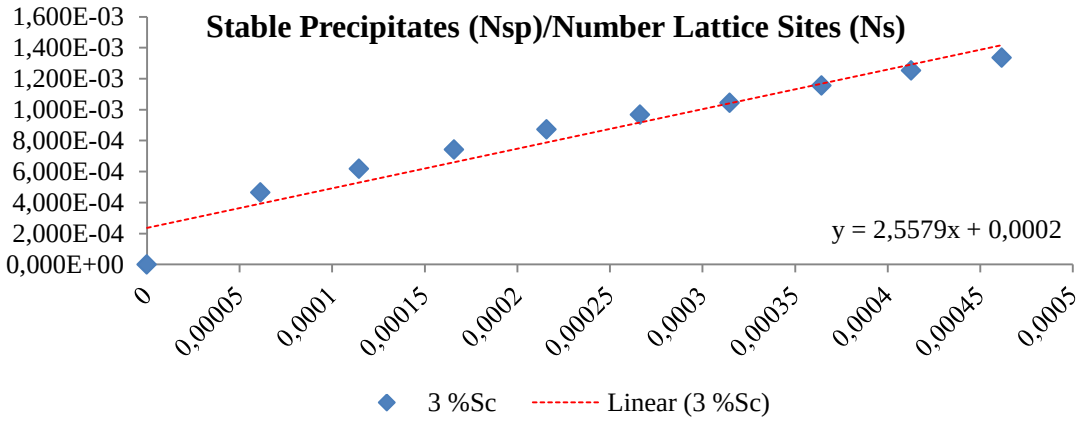


Figure 145 – Gradient calculation for a 3 % Sc simulation at 773.15K.

G. Matlab Implemention of the Kinetic Monte Carlo

i. Function main.m

```
function [flag_1]=Main(l,m,n,ScC,T,MCS,snap,flag_1,t)
%
%  Purpose:
%          Main Monte Carlo Code.
%
%  Synopsis:
%
%  Variable Description:
%          l
%          m
%          n
%          ScC
%          T
%          MCS
%          snap
%          flag_1
%          t
%%
clear global Database
%%
global Database database

%%
% Initial Necessary Parameters
[~, ~, lc, ~]=al_sc_parameters();

% Define Initial Structure / Distribution of atoms
[database]=fcclattice(l,m,n,lc,ScC);

% Number of Snapshots
intervals=(1:(MCS/snap):MCS);

% Create the database
Database(:,:,1)=database;

% Find the the initial vacancy site
randomVacancy=find(database(:,5)==1);

% Real time
Ts=0;

%% Start Monte Carlo
for mcs=1:MCS

        % Identify the 12 nearest neighbors sites
        [nN,n_sec_N]=new_neighbors(lc,randomVacancy,database);

        % Calculate the activation energy
        [Eact]=activationEnergy(nN,n_sec_N,database);

        % Calculate the vacancy exchange frequency
        [vEF,sumVef]=vacancyExchangeFrequency(Eact,nN,database,T);
```

```
        % Generate a random number & the random vacancy selection
        [x]=randomVacancySelection(vEF,sumVef);

        % Update the database
        TheNewVacancySite=database(x,1);
        OldVacancySiteAtom=database(x,5);

        database(TheNewVacancySite,5)=1;
        database(randomVacancy,5)=OldVacancySiteAtom;

        NewVacancySite=TheNewVacancySite;

        % Real Time Calculation
        [ts]=realTime_Calculation(vEF);
        Ts=Ts+ts;

        % Update the Database
        [Database]=updateDatabase(intervals,mcs,Database,database);

        randomVacancy=NewVacancySite;

        etime(clock, t);

end

% End timer:
flag_1=1;
end
```

ii. Function al_sc_parameters.m

```
function [al_Radius, sc_Radius, lc, kB]=al_sc_parameters()

% Atom radius: [Aº]

al_Radius = 1.18;
sc_Radius = 1.84;

% Aluminum lattice constant (a=4.05Aº)
lc = 4.05;

% BoltzmannConstant=8.617?3324e-5;       %(eV/K) (Electron volts/Kelvin)

kB = 8.6173324e-5;
```

iii. Function fcclattice.m

```
function [database] = fcclattice(l,m,n,lc,ScC)
%
% This function creates a list of coordinates of fcc lattice points.
% l, m, and n are the number of unit cells in the x, y, and z directions.
%
%%
% Corner / Boundary / Square atoms:
N = l*m*n;

[x,y,z] = meshgrid(1:lc:lc*l,1:lc:lc*m,1:lc:lc*n);

p(1:N,1) = x(:);
p(N+1:2*N,1) = x(:)+lc/2;
p(2*N+1:3*N,1) = x(:)+lc/2;
p(3*N+1:4*N,1) = x(:);

p(1:N,2) = y(:);
p(N+1:2*N,2) = y(:)+lc/2;
p(2*N+1:3*N,2) = y(:);
p(3*N+1:4*N,2) = y(:)+lc/2;

p(1:N,3) = z(:);
p(N+1:2*N,3) = z(:);
p(2*N+1:3*N,3) = z(:)+lc/2;
p(3*N+1:4*N,3) = z(:)+lc/2;

% Dimension Corrections:
p(p(:,1)==max(p(:,1),[],1), :)=[];
p(p(:,2)==max(p(:,2),[],1), :)=[];
p(p(:,3)==max(p(:,3),[],1), :)=[];

p=sortrows(p,3);
p=sortrows(p,1);

%% Atom Material
% Total number of atoms:
atomNumber=(1:1:length(p))';
```

```
%% Number of Al & Sc Lattice Sites
% Number of vacancies:
num_V_Sites=1;
% Number of Scandium atoms:
num_Sc_Sites=ceil((length(p)*ScC)/100);
% Number of Aluminum atoms:
num_Al_Sites=length(p)-num_Sc_Sites-num_V_Sites;

% Total random distribution:

% aluminum atomic number=13; scandium atomic number=21
atomMaterial=[ones(num_Al_Sites,1)*13;ones(num_Sc_Sites,1)*21;ones(num_V_Sites,1)];
atomMaterial=atomMaterial(randperm(length(atomMaterial)));

database=[atomNumber p atomMaterial];
```

iv. Function neighbors.m

```
function [nN,n_sec_N]=neighbors(lc,randomVacancy,database)
%% Nearest Neighbors:
%
% xa=coordenate in of randomVacancy
% ya=coordenate in of randomVacancy
% za=coordenate in of randomVacancy
%
%%
a=lc; b=lc; c=lc;

Lx=max(database(:,2));
Ly=max(database(:,3));
Lz=max(database(:,4));

xa=database(randomVacancy,2);
ya=database(randomVacancy,3);
za=database(randomVacancy,4);

%% First Nearest Neighbors
n=zeros(12,3);
% 4 Neighbors on the XY Plane:
% [x,y,z]
n(1,:)=[xa+(a/2),ya+(b/2),za];
n(2,:)=[xa-(a/2),ya+(b/2),za];
n(3,:)=[xa-(a/2),ya-(b/2),za];
n(4,:)=[xa+(a/2),ya-(b/2),za];
%
% 4 Neighbors on the YZ Plane:
% [x,y,z]
n(5,:)=[xa,ya+(b/2),za+(c/2)];
n(6,:)=[xa,ya+(b/2),za-(c/2)];
n(7,:)=[xa,ya-(b/2),za-(c/2)];
n(8,:)=[xa,ya-(b/2),za+(c/2)];
%
% 4 Neighbors on the XZ Plane:
% [x,y,z]
n(9,:)=[xa+(a/2),ya,za+(c/2)];
n(10,:)=[xa-(a/2),ya,za+(c/2)];
n(11,:)=[xa-(a/2),ya,za-(c/2)];
n(12,:)=[xa+(a/2),ya,za-(c/2)];
```

```
%
%% Wrap Corrections:
%
for i=1:12
        % XY Plane:
        if n(i,1)>Lx; n(i,1)=1+a/2; end
        if n(i,2)>Ly; n(i,2)=1+b/2; end
        if n(i,1)<1; n(i,1)=Lx-a/2; end
        if n(i,2)<1; n(i,2)=Ly-b/2; end
%
        % YZ Plane:
        if n(i,2)>Ly; n(i,2)=1+b/2; end
        if n(i,3)>Lz; n(i,3)=1+c/2; end
        if n(i,2)<1; n(i,2)=Ly-b/2; end
        if n(i,3)<1; n(i,3)=Lz-c/2; end
%
        % XZ Plane:
        if n(i,1)>Lx; n(i,1)=1+a/2; end
        if n(i,3)>Lz; n(i,3)=1+c/2; end
        if n(i,1)<1; n(i,1)=Lx-a/2; end
        if n(i,3)<1; n(i,3)=Lz-c/2; end
end
%
%%
% Find

nN=zeros(1,12);
tolerance = 0.5;
for i=1:12
        [x]=find(abs(database(:,2)-n(i,1))<tolerance & abs(database(:,3)-n(i,2))<tolerance &
abs(database(:,4)-n(i,3))<tolerance);
        nN(1,i)=x;
end

%% Second Nearest Neighbors
sec_N=zeros(6,3);
% 4 Neighbors on the XY Plane:
% [x,y,z]
sec_N(1,:)=[xa+a,ya,za];
sec_N(2,:)=[xa-a,ya,za];
sec_N(3,:)=[xa,ya+b,za];
sec_N(4,:)=[xa,ya-b,za];
% 2 Neighbors on the YZ Plane/XZ Plane:
% [x,y,z]
sec_N(5,:)=[xa,ya,za+c];
sec_N(6,:)=[xa,ya,za-c];
%
%% Wrap Corrections:
%
for i=1:6
        % XY Plane:
        if sec_N(i,1)>Lx+a/2; sec_N(i,1)=1+a; end
        if sec_N(i,1)>Lx && sec_N(i,1)<=Lx+a/2; sec_N(i,1)=1+a/2; end
        if sec_N(i,1)<(1-a/2); sec_N(i,1)=Lx-a; end
        if sec_N(i,1)>(1-a) && sec_N(i,1)<1; sec_N(i,1)=Lx-a/2; end
        if sec_N(i,2)>Ly+b/2; sec_N(i,2)=1+b; end
        if sec_N(i,2)>Ly && sec_N(i,2)<=Ly+b/2; sec_N(i,2)=1+b/2; end
        if sec_N(i,2)<(1-b/2); sec_N(i,2)=Ly-b; end
        if sec_N(i,2)>(1-b) && sec_N(i,2)<1; sec_N(i,2)=Ly-b/2; end
%
```

```
        % YZ Plane & XZ Plane:
        if sec_N(i,3)>Lz+c/2; sec_N(i,3)=1+c; end
        if sec_N(i,3)>Lz && sec_N(i,3)<=Lz+c/2; sec_N(i,3)=1+c/2; end
        if sec_N(i,3)<(1-c/2); sec_N(i,3)=Lz-c; end
        if sec_N(i,3)>(1-c) && sec_N(i,3)<1; sec_N(i,3)=Lz-c/2; end
%
end
%
%%
% Find

n_sec_N=zeros(1,6);
tolerance = 0.5;
for i=1:6
        [x]=find(abs(database(:,2)-sec_N(i,1))<tolerance & abs(database(:,3)-sec_N(i,2))<tolerance &
abs(database(:,4)-sec_N(i,3))<tolerance);
        n_sec_N(1,i)=x;
end
```

v. Function activationEnergy.m

```
function [Eact]=activationEnergy(nN,n_sec_N,database)
%
%  Purpose:
%        .
%
%  Synopsis:
%        [Eact]=activationEnergy()
%
%  Variable Description:
%        Eact
%        nN
%        n_sec_N
%        database
%%
global E_AlAl_1 E_ScSc_1 E_AlSc_1 E_AlSc_2 E_AlV_1 E_ScV_1
global e_spAl e_spSc

E_AlAl_2=E_AlAl_1/2;
E_ScSc_2=E_ScSc_1/2;

% First Neighbor Types
num_Al_1=0;
num_Sc_1=0;
for i=1:length(nN)
        if database(nN(i),5)==13
                num_Al_1=num_Al_1+1;
        elseif database(nN(i),5)==21
                num_Sc_1=num_Sc_1+1;
        end
end

% Neighboring Bonds
% Neighboring Bonds for Al atom
n_AlAl_1=12-1-num_Sc_1;
n_AlSc_1=12-1-n_AlAl_1;

% Neighboring Bonds for Sc atom
```

```
n_ScSc_1=12-1-num_Al_1;
n_ScAl_1=12-1-n_ScSc_1;

% Neighboring Bonds for the vacancy
n_AlV_1=num_Al_1;
n_ScV_1=num_Sc_1;
%%
% Second Neighbor Types
num_Al_2=0;
num_Sc_2=0;
for i=1:length(n_sec_N)
        if database(n_sec_N(i),5)==13
                num_Al_2=num_Al_2+1;
        elseif database(n_sec_N(i),5)==21
                num_Sc_2=num_Sc_2+1;
        end
end

% Neighboring Bonds
% Neighboring Bonds for Al atom
n_AlAl_2=6-num_Sc_2;
n_AlSc_2=6-n_AlAl_2;
% Neighboring Bonds for Sc atom
n_ScSc_2=6-num_Al_2;

%%

Eact=zeros(2,12);
for i=1:length(nN)
        if database(nN(i),5)==13
                Eact(1,i)=e_spAl-
((n_AlAl_1*E_AlAl_1)+(n_AlSc_1*E_AlSc_1)+(n_AlAl_2*E_AlAl_2)+(n_AlSc_2*E_AlSc_2)+(n_Al
V_1*E_AlV_1)+(n_ScV_1*E_ScV_1));
        elseif database(nN(i),5)==21
                Eact(1,i)=e_spSc-
((n_ScSc_1*E_ScSc_1)+(n_ScAl_1*E_AlSc_1)+(n_ScSc_2*E_ScSc_2)+(n_AlSc_2*E_AlSc_2)+(n_Sc
V_1*E_ScV_1)+(n_ScV_1*E_ScV_1));
        end
        Eact(2,i)=nN(1,i);
end
```

vi. Function vacancyExchangeSelection.m

```
function [vEF,sumVef]=vacancyExchangeFrequency(Eact,nN,database,T)
%
%  Purpose:
%         .
%
%  Synopsis:
%         [vEF]=vacancyExchangeFrequency(Eact,nN)
%
%  Variable Description:
%          Eact
%          nN
%          database
%          T
%
%%
```

```
global vAl vSc

% Initial Parameters
[~, ~, ~, kB]=al_sc_parameters();

% Absolute Temperature [K]
T=T+273.15;

vEF=zeros(2,12);

for i=1:length(nN)
        if database(nN(i),5)==13
                vEF(1,i)=vAl*(exp(-(Eact(1,i)/(kB*T))));
                vEF(2,i)=nN(i);
        elseif database(nN(i),5)==21
                vEF(1,i)=vSc*(exp(-(Eact(1,i)/(kB*T))));
                vEF(2,i)=nN(i);
        end
end

%%

sumVef=zeros(1,12);
for i=1:12
        if i==1
                sumVef(1,i)=vEF(1,i);
        else
                sumVef(1,i)=sumVef(1,(i-1))+vEF(1,i);
        end
end
```

vii. Function randomVacancySelection.m

```
function [x]=randomVacancySelection(vEF,sumVef)
%
%  Purpose:
%          .
%
%  Synopsis:
%          [x]=randomVacancySelection(vEF,sumVef)
%
%  Variable Description:
%           vEF
%           sumVef
%
%% Generate a random number between 0 & 1
randNum = rand(1);

x=0;
for i=2:12
        if randNum > sumVef(i-1) && randNum <= sumVef(i)
                x=vEF(2,i);
        end
end

if x==0
        x=vEF(2,1);
end
```

viii. Function realTime_Calculation.m

```
function [ts]=realTime_Calculation(vEF)
%
%  Purpose:
%          .
%
%  Synopsis:
%          [ts]=realTime_Calculation(vEF)
%
%  Variable Description:
%           vEF
%
%% Real Time Calculation

ts=1/(vEF(1,1)+vEF(1,2)+vEF(1,3)+vEF(1,4)+vEF(1,5)+vEF(1,6)...
          +vEF(1,7)+vEF(1,8)+vEF(1,9)+vEF(1,10)+vEF(1,11)+vEF(1,12));
```

ix. Function precipitation.m

```
function [precipitationSites]=precipitation()
%
%  Purpose:
%          .
%
%  Synopsis:
%          [precipitationSites]=precipitation()
%
%  Variable Description:
%
%
%%
global Database

% Initial Parameters
[~, ~, lc, ~]=al_sc_parameters();

precipitationSites=[];

[x]=find(Database(:,5,end)==21);

for i=1:length(x)
          ii=x(i,1);
          [nN]=new_neighbors(lc,ii,Database(:,:,end));

          sumScAtoms=0;
          for y=1:12
                    if Database(nN(y),5,end)==21
                              sumScAtoms=sumScAtoms+1;
                    else
                              continue
                    end
          end
```

```
        if sumScAtoms>=6
                precipitationSites(1,end+1)=i; %#ok
        end

end
```

x. Function vtkDataFile.m

```
function vtkDataFile_1(snapShot,atomType)
%
%  Purpose:
%          Write a VTK DataFile.
%
%  Synopsis:
%          vtkDataFile_1(snapShot,atomType)
%
%  Variable Description:
%           snapShot
%           atomType
%
%%
global Database

%% Save to a ".vtk" file
[File,Folder]=uiputfile('.vtk','Save a VTK file');
if File == 0
        return
end

%% Snapshot

if snapShot(2)==1
        snapshot=1;
elseif snapShot(3)==1
        [size1 size2 size3]=size(Database); %#ok
        snapshot=size3;
end

%% Atomic Type

if atomType(1)==1
        atomicNumber=1;
elseif atomType(2)==1
        atomicNumber=13;
elseif atomType(3)==1
        atomicNumber=21;
end
%% Create file

% New line.
nl = sprintf('\n');

%% Write the file
fid=fopen(fullfile(Folder,File),'w');

% Write the file header.
if atomicNumber==13 || atomicNumber==21
        fwrite(fid, ['# vtk DataFile Version 2.0' nl...
```

```
                'Generated by Alfredo de Moura' nl...
                'ASCII' nl ...
                'DATASET POLYDATA' nl...
                'POINTS ' num2str(length(find(Database(:,5)==atomicNumber))) ' ' 'float ' nl]);
else
        fwrite(fid, ['# vtk DataFile Version 2.0' nl...
                'Generated by Alfredo de Moura' nl...
                'ASCII' nl ...
                'DATASET POLYDATA' nl...
                'POINTS ' num2str(length(Database)) ' ' 'float ' nl]);
end

for i = 1:length(Database)
        if atomicNumber==1
                x1=num2str(Database(i,2,snapshot),'%.3f');
                y1=num2str(Database(i,3,snapshot),'%.3f');
                z1=num2str(Database(i,4,snapshot),'%.3f');
                % Write the values as text numbers.
                fwrite(fid, [num2str(x1) ' ' num2str(y1) ' ' num2str(z1) ' ']);
                % Newline.
                fwrite(fid, nl);
        elseif Database(i,5)==atomicNumber
                x1=num2str(Database(i,2,snapshot),'%.3f');
                y1=num2str(Database(i,3,snapshot),'%.3f');
                z1=num2str(Database(i,4,snapshot),'%.3f');
                % Write the values as text numbers.
                fwrite(fid, [num2str(x1) ' ' num2str(y1) ' ' num2str(z1) ' ']);
                % Newline.
                fwrite(fid, nl);
        else
                continue
        end

end


if atomicNumber==13 || atomicNumber==21
        fwrite(fid, ['VERTICES '  num2str(length(find(Database(:,5)==atomicNumber))) ' '
num2str(2*length(find(Database(:,5)==atomicNumber))) ' ' nl]);
        fwrite(fid, [num2str(1) ' ' num2str(0) ' ' nl]);
        for i = 1:(length(find(Database(:,5)==atomicNumber))-1)
                % Write the values as text numbers.
                fwrite(fid, [num2str(1) ' ' num2str(i) ' ' ]);
                % Newline.
                fwrite(fid, nl);
        end
        fwrite(fid, ['POINT_DATA '  num2str(length(find(Database(:,5)==atomicNumber))) ' ' nl]);
        fwrite(fid, ['SCALARS atom_type int 1 ' nl]);
        fwrite(fid, ['LOOKUP_TABLE default ' nl]);
        fwrite(fid, [num2str(ones(1,length(find(Database(:,5)==atomicNumber)))*39) ' ' nl]);
else
        fwrite(fid, ['VERTICES '  num2str(length(Database)) ' ' num2str(2*length(Database)) ' ' nl]);
        fwrite(fid, [num2str(1) ' ' num2str(0) ' ' nl]);
        for i = 1:(length(Database)-1)
                % Write the values as text numbers.
                fwrite(fid, [num2str(1) ' ' num2str(i) ' ' ]);
                % Newline.
                fwrite(fid, nl);
        end
        fwrite(fid, ['POINT_DATA '  num2str(length(Database)) ' ' nl]);
```

```
        fwrite(fid, ['SCALARS atom_type int 1 ' nl]);
        fwrite(fid, ['LOOKUP_TABLE default ' nl]);
        fwrite(fid, [num2str(ones(1,length(Database))*39) ' ' nl]);

end
% Close the file.
fclose(fid);
```

xi. Function pdbDataFile.m

```
function pdbDataFile(snapShot,atomType)
%
%  Purpose:
%          .
%
%  Synopsis:
%          pdbDataFile(snapShot,atomType)
%
%  Variable Description:
%           snapShot
%           atomType
%
%%
global Database

%% Save to a ".pdb" file

[File,Folder]=uiputfile('.pdb','Save a PDB file');
if File == 0
        return
end

file=fullfile(Folder,File);

%% Snapshot

if snapShot(2)==1
        snapshot=1;
elseif snapShot(3)==1
        [size1 size2 size3]=size(Database); %#ok
        snapshot=size3;
end

%% Atomic Type

if atomType(2)==1
        atomicNumber=13;
elseif atomType(3)==1
        atomicNumber=21;
end

%% Create file

fid=fopen(file,'w');

n=length(Database(:,:,snapshot));
count=1;
for i=1 : n
```

```
        if Database(i,5)==atomicNumber && atomicNumber==21
                atoms='Sc';
                fprintf(fid, 'ATOM          %u  %s                      %u       %.5f %.5f %.5f\n',
count, atoms, 1, Database(i,2,snapshot), Database(i,3,snapshot), Database(i,4,snapshot));
                count=count+1;
        elseif Database(i,5)==atomicNumber && atomicNumber==13
                atoms='Al';
                fprintf(fid, 'ATOM          %u  %s                      %u       %.5f %.5f %.5f\n',
count, atoms, 1, Database(i,2,snapshot), Database(i,3,snapshot), Database(i,4,snapshot));
                count=count+1;
        end

end
fprintf(fid,'END\n');

fclose(fid);
```

xii. Function printable_report_simulation.m

```
function [] = Printable_report_simulation()
%
%  Purpose:
%          .
%
%  Synopsis:
%          [] = Printable_report_simulation()
%
%  Variable Description:
%
%%
global E_AlAl_1 E_ScSc_1 E_AlSc_1 E_AlSc_2 E_AlV_1 E_ScV_1
global e_spAl e_spSc vAl vSc
global num_Al_Sites num_Sc_Sites
global MCS T
%
%%
[File, Folder]=uiputfile('.txt','Save Reference Results');
if File==0
        return
end

%%
% Main Header

dlmwrite(fullfile(Folder,File), '
################################','delimiter', '','coffset',5,'-append');
dlmwrite(fullfile(Folder,File), '                                                    Simulation
Report','delimiter', '','coffset',5,'-append');
dlmwrite(fullfile(Folder,File), ['                                  Made in: ' datestr(clock)],'delimiter',
'','roffset',1,'coffset',5,'-append');
dlmwrite(fullfile(Folder,File), ['                                                    By: '
getenv('COMPUTERNAME')],'delimiter', '','roffset',1,'coffset',5,'-append');
dlmwrite(fullfile(Folder,File), '                                                    Thesis
2012','delimiter', '','roffset',1,'coffset',5,'-append');
dlmwrite(fullfile(Folder,File), '
################################','delimiter', '','coffset',5,'-append');

% End of Main header
```

```
dlmwrite(fullfile(Folder,File), '
=====================================','delimiter', ','coffset',5,'-append');

a=3;
% Parameter Header
dlmwrite(fullfile(Folder,File), 'Simulation material information','delimiter', ','roffset',2,'coffset',5,'-
append');
dlmwrite(fullfile(Folder,File),
'**************************************************************','delimiter', ','coffset',5,'-
append');

dlmwrite(fullfile(Folder,File), ['Aluminum-Aluminum 1st nearest neighbor pair effective energy: '
num2str(E_AlAl_1) ' [ eV ]'],'delimiter', ','roffset',2,'coffset',5,'-append');
dlmwrite(fullfile(Folder,File), ['Scandium-Scandium 1st nearest neighbor pair effective energy: '
num2str(E_ScSc_1) ' [ eV ]'],'delimiter', ','roffset',2,'coffset',5,'-append');
dlmwrite(fullfile(Folder,File), ['Aluminum-Scandium 1st nearest neighbor pair effective energy: '
num2str(E_AlSc_1) ' [ eV ]'],'delimiter', ','roffset',2,'coffset',5,'-append');
dlmwrite(fullfile(Folder,File), ['Aluminum-Scandium 2nd nearest neighbor pair effective energy: '
num2str(E_AlSc_2) ' [ eV ]'],'delimiter', ','roffset',2,'coffset',5,'-append');
dlmwrite(fullfile(Folder,File), ['Aluminum-Vacancy 1st nearest neighbor pair effective energy: '
num2str(E_AlV_1) ' [ eV ]'],'delimiter', ','roffset',2,'coffset',5,'-append');
dlmwrite(fullfile(Folder,File), ['Scandium-Vacancy 1st nearest neighbor pair effective energy: '
num2str(E_ScV_1) ' [ eV ]'],'delimiter', ','roffset',2,'coffset',5,'-append');
dlmwrite(fullfile(Folder,File), ['Aluminum Saddle point energy: ' num2str(e_spAl) ' [ eV ]'],'delimiter',
','roffset',2,'coffset',5,'-append');
dlmwrite(fullfile(Folder,File), ['Scandium Saddle point energy: ' num2str(e_spSc) ' [ eV ]'],'delimiter',
','roffset',2,'coffset',5,'-append');
dlmwrite(fullfile(Folder,File), ['Aluminum attempt frequency: ' num2str(vAl) ' [ eV ]'],'delimiter',
','roffset',2,'coffset',5,'-append');
dlmwrite(fullfile(Folder,File), ['Scandium attempt frequency: ' num2str(vSc) ' [ eV ]'],'delimiter',
','roffset',2,'coffset',5,'-append');

dlmwrite(fullfile(Folder,File),
'**************************************************************','delimiter', ','coffset',5,'-
append');

dlmwrite(fullfile(Folder,File), 'Simulation input information','delimiter', ','roffset',2,'coffset',5,'-append');
dlmwrite(fullfile(Folder,File),
'**************************************************************','delimiter', ','coffset',5,'-
append');

dlmwrite(fullfile(Folder,File), ['Number of Aluminum atoms: ' num2str(num_Al_Sites) ' [ atoms
]'],'delimiter', ','roffset',2,'coffset',5,'-append');
dlmwrite(fullfile(Folder,File), ['Number of Scandium atoms: ' num2str(num_Sc_Sites) ' [ atoms
]'],'delimiter', ','roffset',2,'coffset',5,'-append');
dlmwrite(fullfile(Folder,File), ['Number of Monte Carlo Steps: ' num2str(MCS)'],'delimiter',
','roffset',2,'coffset',5,'-append');
dlmwrite(fullfile(Folder,File), ['Temperature Applied: ' num2str(T) ' [ ºC ]'],'delimiter',
','roffset',2,'coffset',5,'-append');

dlmwrite(fullfile(Folder,File),
'**************************************************************','delimiter', ','coffset',5,'-
append');

dlmwrite(fullfile(Folder,File), 'Simulation output information','delimiter', ','roffset',2,'coffset',5,'-append');
```

```
dlmwrite(fullfile(Folder,File),
'**********************************************************************','delimiter', '','coffset',5,'-
append');

dlmwrite(fullfile(Folder,File), ['Number of precipitation sites: ' num2str(a)'],'delimiter',
'','roffset',2,'coffset',5,'-append');
dlmwrite(fullfile(Folder,File), ['Average precipitation size: ' num2str(a) ' [ nm ]'],'delimiter',
'','roffset',2,'coffset',5,'-append');
dlmwrite(fullfile(Folder,File), ['Simulation time: ' num2str(a) ' [ s ]'],'delimiter', '','roffset',2,'coffset',5,'-
append');

dlmwrite(fullfile(Folder,File),
'**********************************************************************','delimiter', '','coffset',5,'-
append');

dlmwrite(fullfile(Folder,File), 'Comparative with the Classical Nucleation Theory','delimiter',
'','roffset',2,'coffset',5,'-append');
dlmwrite(fullfile(Folder,File),
'**********************************************************************','delimiter', '','coffset',5,'-
append');

dlmwrite(fullfile(Folder,File), ['Number of precipitation sites: ' num2str(a)'],'delimiter',
'','roffset',2,'coffset',5,'-append');
dlmwrite(fullfile(Folder,File), ['Average precipitation size: ' num2str(a) ' [ nm ]'],'delimiter',
'','roffset',2,'coffset',5,'-append');
dlmwrite(fullfile(Folder,File), ['Simulation time: ' num2str(a) ' [ s ]'],'delimiter', '','roffset',2,'coffset',5,'-
append');

dlmwrite(fullfile(Folder,File),
'**********************************************************************','delimiter', '','coffset',5,'-
append');
```

H. User Interface of the Matlab Implementation

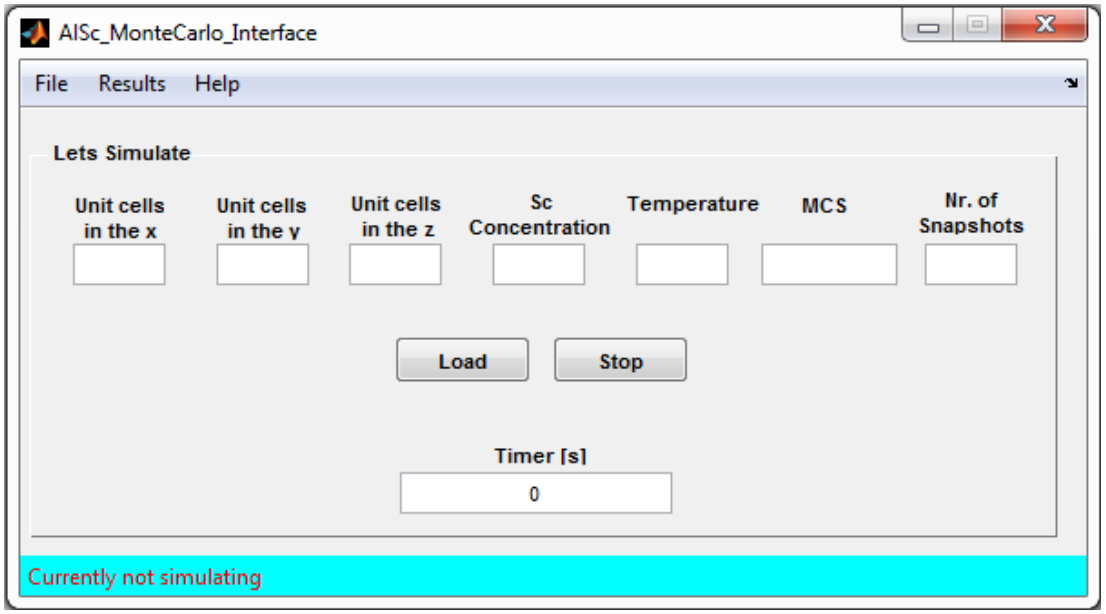


Figure 146 – "Main Window" Graphical User Interface.

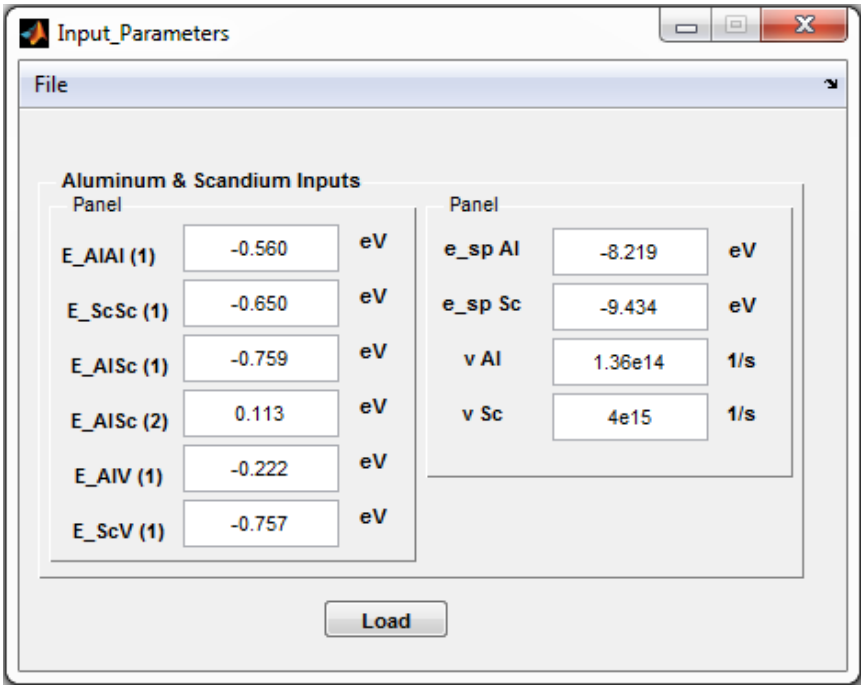


Figure 147 – "Input Parameters" Graphical User Interface.

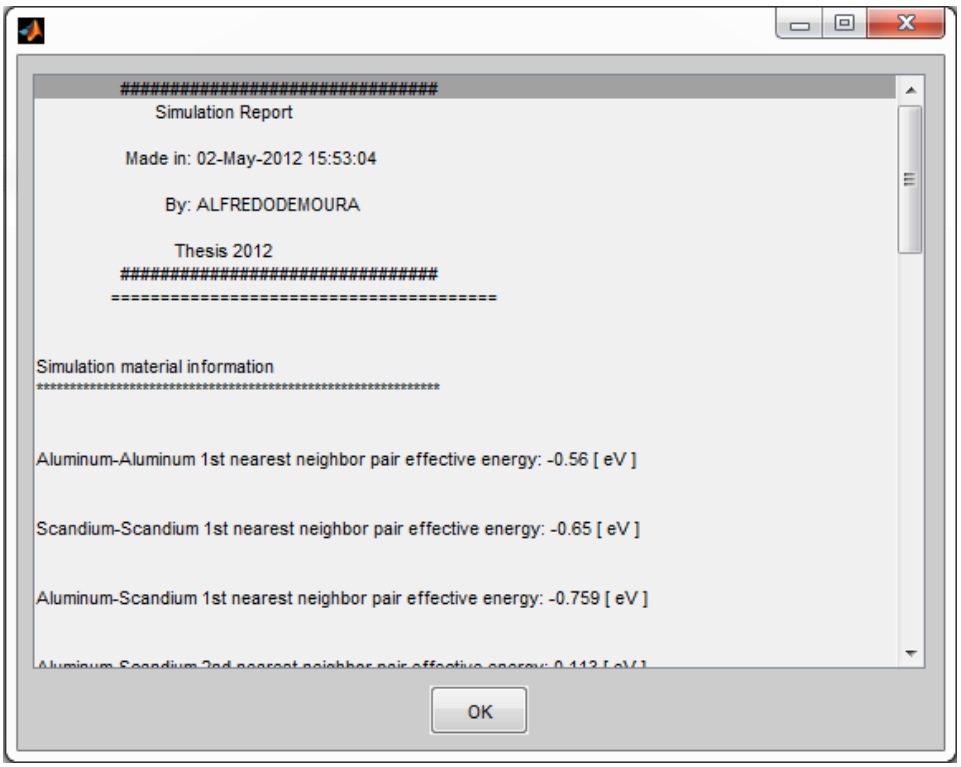


Figure 148 – "Simulation Report" Graphical User Interface.

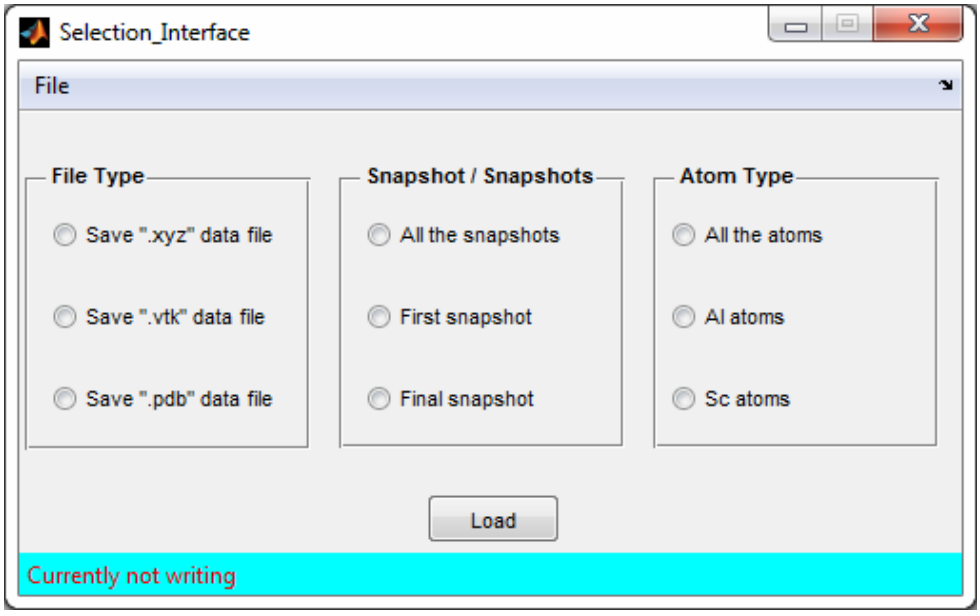


Figure 149 – "Save Output File" Graphical User Interface.

I. C Language Source and Header Files

i. activationEnergy.c

```
#include <math.h>
#include "main_includes.h"

// Date: 07 July 2012

// Approximation of real vacancy concentration in Aluminiun as a function of T.
//
//		Parameters:
//			T			Temperature, in range [446.8 : 923.1] Kelvin.
//		Return:
//						Real vacancy concentration in Aluminiun.
//
// C_V_real = -0.005792301654 + 5.281432466e-5*T - 1.916781695e-7*T^2 +
//				3.466630615e-10*T^3 - 3.132467044e-13*T^4 + 1.135950846e-16*T^5
//
double realVacancyConcentration(double T){
	double C_V_real, powerT;

	if(T<446.8) {
		printf("Real vacancy concentration can't be correctly computed for T below 446.8 K.\n");
		T=446.8;
	}
	if(T>923.1) {
		printf("Real vacancy concentration can't be correctly computed for T above than 923.1 K.\n");
		T=923.1;
	}
	powerT   = T;
	C_V_real  = -0.005792301654 + 5.281432466e-5*powerT;
	powerT   *= T; // T^2
	C_V_real -= 1.916781695e-7*powerT;
	powerT   *= T; // T^3
	C_V_real += 3.466630615e-10*powerT;
	powerT   *= T; // T^4
	C_V_real -= 3.132467044e-13*powerT;
	powerT   *= T; // T^5
	C_V_real += 1.135950846e-16*powerT;
	printf("Real vacancy concentration in Aluminiun at %f Kelvin is %e.\n", T, C_V_real);
	return (C_V_real);
}
```

```
// Simulation vacancy concentration in used lattice.
//
//          Parameters:
//                    latP                Lattice parameters.
//          Return:
//                                        Simulation vacancy concentration.
//
double simVacancyConcentration(LatticeParameters *latP) {
          double C_V_sim;

          C_V_sim =(double)latP->num_V_Sites/(double)(latP->nbz*latP->nby*latP->nbx*FCC_ATOMS_CELL);

          printf("Simulation vacancy concentration is %e.\n", C_V_sim);
          return (C_V_sim);
}

// ********************************************************************
// Compute the activation energy (Eact) associated with the transition to
// every first nearest neighbor of the vacancy.
//
//  Parameters:
//                    simP                Pointer to the simulation parameters.
//                    energy              Pointer to the ebergy&kinetic parameters.
//                    posV                The vacancy position in all atoms' array.
//                    atomN1N2            Pointer to the array where the position of all atoms' 1st
//                                        and 2nd nearest neighbors is stored.
//                    atomT               Pointer to the array where all atoms' type is stored.
//                    Eact                Pointer to the array where activation energy will be stored.
//
//  Compute: Eact
//
void activationEnergy(SimulationParameters *simP, EnergyParameters *energyP,
          int posV, NeighborsType atomN1N2[], AtomType *atomT, double *Eact){

          int          i, j, nPos, nnPos, n_AlV_1a, n_ScV_1a, n_AlV_1b, n_ScV_1b;
          int          n_AlAl_1, n_AlSc_1, n_ScSc_1;
          int          n_AlAl_2, n_AlSc_2, n_ScSc_2;
          AtomType nType, nnType;

          // Count the number of vacancy's first neighbors of Al and Sc type

          n_AlV_1b = 0; // (5b) Vacancy Al-V bonds to its 1st neighbors
          n_ScV_1b = 0; // (6b) Vacancy Sc-V bonds to its 1st neighbors
          for (i=0;i<NUM_1ST_NEIGHBORS;++i) {
                    nPos = atomN1N2[posV].N1[i]; // vacancy 1st neighbor position in the array of all atoms' position
                    if (atomT[nPos] == Al) {
                              ++n_AlV_1b;
                    }
                    else if (atomT[nPos] == Sc) {
                              ++n_ScV_1b;
                    }
          }

          // ---------------------------------------------------------------------
          // Iterate over all 12 vacancy neighbors
```

```
for (i=0;i<NUM_1ST_NEIGHBORS;++i) {

        nPos  = atomN1N2[posV].N1[i];       // vacancy 1st i-th neighbor position
        nType = atomT[nPos];                // vacancy 1st i-th neighbor type

        // ------------------------------------------------------------------
         if (nType == Al) { // vacancy 1st i-th neighbor is an Aluminum

                n_AlAl_1=0;       // (1)  Al-Al(1) bonds from j-th 1st neighbor of the vacancy
                n_AlSc_1=0;       // (2)  Al-Sc(1) bonds from j-th 1st neighbor of the vacancy
                n_AlV_1a=0;       // (5a) Al-V(1)  bonds from j-th 1st neighbor of the vacancy

                for (j=0;j<NUM_1ST_NEIGHBORS;++j) {
                        // j-th 1st neighbor of the vacancy 1st neighbor position
                        nnPos  = atomN1N2[nPos].N1[j];
                        nnType = atomT[nnPos];
                        if (nnType == Al) {
                                ++n_AlAl_1; // (1)
                        }
                        else if (nnType == Sc) {
                                ++n_AlSc_1; // (2)
                        }
                        else if (nnType == Vacancy) {
                                ++n_AlV_1a; // (5a)
                        }
                }

                n_AlAl_2=0;       // (3)  Al-Al(2) bonds from j-th 1st neighbor of the vacancy
                n_AlSc_2=0;       // (4)  Al-Sc(2) bonds from j-th 1st neighbor of the vacancy

                for (j=0;j<NUM_2ND_NEIGHBORS;++j) {
                        // j-th 2nd neighbor of the vacancy 1st neighbor position
                        nnPos  = atomN1N2[nPos].N2[j];
                        nnType = atomT[nnPos];
                        if (nnType == Al) {
                                ++n_AlAl_2;  // (3)
                        }
                        else if (nnType == Sc) {
                                ++n_AlSc_2;  // (4)
                        }
                }

                // [Binkele&Schmauder 2003, eq. 3]
                // Eact = EspAl -{(1)*E1+(2)*E2+(3)*E3+(4)*E4+[(5a)+(5b)-1]*E5+(6b)*E6}

                Eact[i] = energyP->e_spAl-(n_AlAl_1*energyP->E_AlAl_1+
                  n_AlSc_1*energyP ->E_AlSc_1+n_AlAl_2*energyP->E_AlAl_2+
                  n_AlSc_2*energyP->E_AlSc_2+(n_AlV_1a+n_AlV_1b-1)*energyP->E_AlV_1+
                  n_ScV_1b*energyP->E_ScV_1);
        }
        // ------------------------------------------------------------------
        else if (nType == Sc) { // vacancy 1st i-th neighbor is a Scandium

                n_AlSc_1=0;       // (2)  Al-Sc(1) bonds from j-th 1st neighbor of the vacancy
                n_ScSc_1=0;       // (7)  Sc-Sc(1) bonds from j-th 1st neighbor of the vacancy
                n_ScV_1a=0;       // (6a) Sc-V(1)  bonds from j-th 1st neighbor of the vacancy
```

```
                        for (j=0;j<NUM_1ST_NEIGHBORS;++j) {
                                // j-th 1st neighbor of the vacancy 1st neighbor position
                                nnPos  = atomN1N2[nPos].N1[j];
                                nnType = atomT[nnPos];
                                if (nnType == Al) {
                                        ++n_AlSc_1; // (2)
                                }
                                else if (nnType == Sc) {
                                        ++n_ScSc_1; // (7)
                                }
                                else if (nnType == Vacancy) {
                                        ++n_ScV_1a; // (6a)
                                }
                        }

                        n_AlSc_2=0;     // (4)  Al-Sc(2) bonds from j-th 1st neighbor of the vacancy
                        n_ScSc_2=0;     // (8)  Sc-Sc(2) bonds from j-th 1st neighbor of the vacancy

                        for (j=0;j<NUM_2ND_NEIGHBORS;++j) {
                                // j-th 2nd neighbor of the vacancy 1st neighbor position
                                nnPos  = atomN1N2[nPos].N2[j];
                                nnType = atomT[nnPos];
                                if (nnType == Al) {
                                        ++n_AlSc_2;  // (4)
                                }
                                else if (nnType == Sc) {
                                        ++n_ScSc_2;  // (8)
                                }
                        }

                        // [Binkele&Schmauder 2003, eq. 4]
                        // Eact = EspSc -{(2)*E2+(7)*E7+(4)*E4+(8)*E8+[(6a)+(6b)-1]*E6+(5b)*E5}

                        Eact[i] = energyP->e_spSc-(n_AlSc_1*energyP->E_AlSc_1+n_ScSc_1 *
                          energyP->E_ScSc_1+ n_AlSc_2*energyP->E_AlSc_2+n_ScSc_2 *
                          energyP->E_ScSc_2+(n_ScV_1a+n_ScV_1b-1)*energyP->E_ScV_1+
                          n_AlV_1b*energyP->E_AlV_1);
                }
        }
}

// ********************************************************************
// Compute the vacancy exchange frequencies (vEF) and the accumulated exchange
//  frequencies (sumVef).
//
//  Parameters:
//                  simP            Pointer to the simulation parameters.
//                  energy          Pointer to the ebergy&kinetic parameters.
//                  posV            The vacancy position in all atoms' array.
//                  atomN1N2        Pointer to the array where the position of all atoms' 1st
//                                  and 2nd nearest neighbors is stored.
//                  atomT           Pointer to the array where all atoms' type is stored.
//                  Eact            Pointer to the array with the activation energy.
//                  vEF             Pointer to the array where exchange frequencies will be saved.
//                  sumVef          Pointer to the array where accumulated exchange frequencies will be saved.
//
//  Compute: vEF, sumVef
```

```
//
//  Return: the real time that corresponds to the present Monte Carlo step.
//
double vacancyExchangeFrequency(SimulationParameters *simP, EnergyParameters *energyP,
        int posV, NeighborsType atomN1N2[], AtomType *atomT, double *Eact, double *vEF,
        double *sumVef) {

        int     i, nPos;
        double expoent, sumAbsoluteVef, ts, kBxT;

        kBxT = simP->kB*simP->T;

        // Compute absolute exchange frequencies

        for (i=0;i<NUM_1ST_NEIGHBORS;++i) {
                expoent = -(Eact[i]/kBxT);

                nPos = atomN1N2[posV].N1[i]; // vacancy 1st neighbor position in the array of all atoms' position
                if (atomT[nPos] == Al) {
                        vEF[i] = energyP->vAl*exp(expoent);
                }
                else if (atomT[nPos] == Sc) {
                        vEF[i] = energyP->vSc*exp(expoent);
                }
        }

        // Compute the sum of all 1st neighbors absolute exchange frequencies

        for (i=0, sumAbsoluteVef = 0.0; i<NUM_1ST_NEIGHBORS; ++i) {
                sumAbsoluteVef += vEF[i];
        }

        // Compute the real time that corresponds to the present Monte Carlo step

        ts = 1.0/sumAbsoluteVef;

        // Compute the relative exchange frequencies among 1st neighbors

        for (i=0;i<NUM_1ST_NEIGHBORS;++i) {
                vEF[i] = vEF[i]/sumAbsoluteVef;
        }

        // Compute accumulated relative exchange frequencies among 1st neighbors

        sumVef[0] = vEF[0];
        for (i=1;i<NUM_1ST_NEIGHBORS;++i) {
                sumVef[i] = sumVef[i-1]+vEF[i];
        }

        return ts;
}
```

```
// ********************************************************************
// Select a 1st nearest neighbor, based on (i) a random number (between 0 and 1)
// and (ii) the exchange frequencies.
//
//  Parameters:
//          vEF             Pointer to the array of exchange frequencies.
//          sumVef          Pointer to the array of accumulated exchange frequencies.
//          posV            The vacancy position in all atoms' array.
//          atomN1N2        Pointer to the array where the position of all atoms' 1st
//                          and 2nd nearest neighbors is stored.
//
//          Return:   The position of the selected neighbor (a value from 0..11).
//
int randomNeighborSelection(double *vEF, double *sumVef, int posV, NeighborsType atomN1N2[]) {

        int         i, pos;
        double      randNum;

        genRandDoubleNoSeed(&randNum, 0.0, 1.0, 1);

        pos   = atomN1N2[posV].N1[0];
        for (i=1;i<NUM_1ST_NEIGHBORS;++i) {
                if ((randNum > sumVef[i-1]) && (randNum <= sumVef[i])) {
                        pos = atomN1N2[posV].N1[i];
                }
        }

        return pos;
}
```

ii. ArrayList.c

```
#include <stdio.h>
#include <stdlib.h>

#include "ArrayList.h"
#include "ArrayUtil.h"

/*
 * ArrayList.c      (INTEGER DATA ONLY!)
 *
 * Date: 10 September 2012
 */

void init (ArrayList *const list) {
        initWithSize(list, 100);
}
void initWithSize(ArrayList *const list, int capacity) {
        initWithSizeAndIncrementRate(list, capacity, 50);
}

void initWithSizeAndIncrementRate(ArrayList *const list, int capacity, int rate) {
        list->capacity                  = capacity;
        list->increment_rate             = rate;
        list->elements                  = (Element*) calloc(sizeof(Element), list->capacity);
        list->current                   = -1;
}
```

```
// Create an array of elements of type 'ArrayList' with size 'sizeAL'

ArrayList* initArrayWithSizeAndIncrementRate(int sizeAL, int capacity, int rate) {
	ArrayList *arrayAL;
	int	n;

	if ((arrayAL = (ArrayList *)calloc(sizeof(ArrayList),sizeAL)) == NULL) {
		printf("Error on memory allocation for 'arrayAL'\n");
		return NULL;
	}
	for(n=0; n<sizeAL; ++n) {
		arrayAL[n].capacity		= capacity;
		arrayAL[n].increment_rate	= rate;
		arrayAL[n].elements		= (Element*) calloc(sizeof(Element), capacity);
		arrayAL[n].current		= -1;
	}
	return arrayAL;
}

// Free the memory allocated to an array of ArrayList

void freeMemoryArray(ArrayList *arrayAL, int sizeAL) {
	int n;

	for(n=0; n<sizeAL; ++n) {
		free(arrayAL[n].elements);
	}
	free(arrayAL);
}

void removeAll (ArrayList *const list) {
	while (list->current>=0) {
		list->elements[list->current] = (Element){0};
		list->current--;
	}
}

int replace (ArrayList *const list, Element e, int index) {
	if ((index <= list->current) && (index>=0)) {
		list->elements[index] = e;
	}
	return 0;
}

Element* get (ArrayList *const list, int index) {
	if ((index <= list->current) && (index>=0)) {
		Element *e = &list->elements[index];
		return e;
	}
	return NULL;
}

int add (ArrayList *const list, Element e) {
	if (++list->current < list->capacity) {
		list->elements[list->current] = e;
		return 1;
	}
```

```
	else {
		wide(list);
		list->elements[list->current] = e;
		return 1;
	}
	return 0;
}

int size(ArrayList *const list) {
	return list->current+1;
}

int capacity(ArrayList *const list) {
	return list->capacity;
}

static void wide(ArrayList* const list) {
	list->capacity += list->increment_rate;
	Element *newArr = (Element*) calloc(sizeof(Element), list->capacity) ;
	arrayCopy(newArr, 0, list->elements, 0, list->current, list->capacity, sizeof(Element));
	free(list->elements);
	list->elements = newArr;
}

int insert (ArrayList *const list, Element e, int index) {
	if (index <= list->current && ++list->current < list->capacity) {
		shift(list, index, 1, RIGHT);
		list->elements[index] = e;
		return 1;
	}
	else {
		wide(list);
		shift(list, index, 1, RIGHT);
		list->elements[index] = e;
		return 2;
	}
}

int lastIndexOf (const ArrayList *const list, Element e) {
	int index = list->current;
	while (index >-1) {
		if (e.data == list->elements[index].data)
			return (index);
		index--;
	}
	return -1;
}

int firstIndexOf (const ArrayList *const list, Element e) {
	int index = 0;
	while (index <= list->current) {
		if (e.data == list->elements[index].data)
			return index;
		index++;
	}
	return -1;
}
```

```
int isEmpty (const ArrayList *const list) {
	return list->current == -1;
}

Element* removeAt(ArrayList *const list, int index) {
	if (list->current >= index) {
		Element *e = &list->elements[index];
		shift(list, index, 1, LEFT);
		list->current--;
		return e;
	}
	return NULL;
}

// Remove the element (an ArrayList) at position 'index' from an array of elements
// of type ArrayList with size 'sizeAL'.
//
// Return: the new size of the array.

int removeFromArrayAt(ArrayList *const arrayAL, int sizeAL, int index) {
	int  n;

	if(index==(sizeAL-1)) {
		free(arrayAL[sizeAL-1].elements);
		return(sizeAL-1);
	}
	else if(index<(sizeAL-1)) {
		free(arrayAL[index].elements);
		for(n=index; n<(sizeAL-1); ++n) {
			arrayAL[n].capacity		= arrayAL[n+1].capacity;
			arrayAL[n].increment_rate	= arrayAL[n+1].increment_rate;
			arrayAL[n].elements		= arrayAL[n+1].elements;
			arrayAL[n].current		= arrayAL[n+1].current;
		}
		return(sizeAL-1);
	}
	else
		return(sizeAL);
}

void printArray (ArrayList *alist, int sizeAL) {
	int i, n;
	for (n=0; n<sizeAL; n++) {
		printf("ARRAYLIST[%d]:\n",n);
		for (i=0; i<=alist[n].current; i++) {
			Element e = alist[n].elements[i];
			printElement(&e);
		}
		printf("'\n");
	}
}


void print (const ArrayList *const list) {
	int i;
```

```
	for (i=0; i<=list->current; i++) {
		Element e = list->elements[i];
		printElement(&e);
	}
	printf("\n");
}

static void printElement(const Element *const e) {
	printf("%i ", e->data);
}

void freeMemory(ArrayList *list) {
	free(list->elements);
}

static void shift(ArrayList *const list, int index, int rooms, Shift dir) {
	if (dir == RIGHT) {
		arrayCopy(list->elements, index+1, list->elements, index, rooms, list->current, sizeof(Element));
	}
	else { // LEFT
		arrayCopy(list->elements, index, list->elements, index+1, rooms, list->current, sizeof(Element));
	}
}
```

iii. ArrayList.h

```
/*
 * ArrayList.h
 *
 * Date: 10 September 2012
 */
#ifndef ARRAYLIST_H
#define ARRAYLIST_H

#define INITIAL_CAPACITY   16
#define INCREMENT_CAPACITY 16

typedef struct {
	int data;
	} Element;

typedef struct {
	int current;
	int capacity;
	int increment_rate;
	Element *elements;
	} ArrayList;

typedef enum {
	RIGHT, LEFT
	} Shift;

// public functions

void		init  (ArrayList*const);
void		initWithSize  (ArrayList*const, int);
void		initWithSizeAndIncrementRate (ArrayList*const, int, int);
```

```
ArrayList*          initArrayWithSizeAndIncrementRate (int, int, int);
void                removeAll  (ArrayList*);
int                  add (ArrayList*const, Element);
int                  insert (ArrayList*const, Element, int);
Element*            removeAt (ArrayList*const, int);
int                  removeFromArrayAt (ArrayList *const, int, int);
void                freeMemory (ArrayList*const);
void                freeMemoryArray (ArrayList *, int);
int                  replace (ArrayList*const, Element, int);
Element*            get (ArrayList*const, int);
void                print (const ArrayList*const);
void                printArray (ArrayList *, int);
int                 lastIndexOf (const ArrayList*const, Element);
int                 indexOf (const ArrayList*const, Element);
int                 isEmpty (const ArrayList*const);
int                 size (ArrayList *const);
int                 capacity   (ArrayList *const );
// TODO
int                 hashCode (const ArrayList*const);

// static (private) utility functions

// Abstracting the print method of the element by delegating it to the element
// itself (OOP-like feature)

static void         printElement(const Element*const);
static void         shift(ArrayList *const list, int index, int rooms, Shift dir);
static void         wide(ArrayList* const);

#endif // ARRAYLIST_H
```

iv. ArrayUtil.c

```
#include <string.h>
#include <stdint.h>
#include <stdlib.h>

#include "ArrayUtil.h"

/*
 * ArrayUtil.c
 *
 * Date: 09 August 2012
 */

void arrayCopy(void *dest, int dIndex, const void* src, int sIndex, int len, int destLen, size_t size) {
        uint8_t *udest      = (uint8_t*) dest;
        uint8_t *usrc       = (uint8_t*) src;
        dIndex              *= size;
        sIndex              *= size;
        len                 *= size;
        destLen             *= size;

        if (src != dest) {
                memcpy(&udest[dIndex], &usrc[sIndex], len);
        }
```

```
	else {
		if (dIndex > sIndex) {
			uint8_t *tmp = (uint8_t*) calloc(destLen, size);
			memcpy(tmp, &udest[dIndex], (destLen-dIndex));
			memcpy(&udest[dIndex], &usrc[sIndex], len);
			memcpy(&udest[dIndex+len], tmp, (destLen-dIndex));
			free(tmp);
		}
		else if (sIndex > dIndex) {
			memcpy(&udest[dIndex], &usrc[sIndex], (destLen-sIndex)+1);
		}
		else
			return;
	}
}
```

v. ArrayUtil.h

```
/*
 * ArrayUtil.h
 *
 * Date: 09 August 2012
 */

#ifndef ARRAYUTIL_H
#define ARRAYUTIL_H

#include <stddef.h>

/**
 * dest		destination array
 * dIndex		index to which we will start copying
 * src		source array
 * sIndex		index from which we will start copying
 * len		 number of elements that will be copied from source array
 * destArrLen	the length of the destination array (hence C doesn't know any length info about passed arrays)
 * size		the size of the type of the array (ex: if the array of type long, put in this parameter sizeof(long))
 */
void arrayCopy (void *dest, int dIndex, const void* src, int sIndex, int len, int destArrLen, size_t size);

#endif // ARRAYUTIL_H
```

vi. clustering1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "main_includes.h"
#include "ArrayList.h"

// Date: 09 October 2012
```

```
// ********************************************************************************
//  Compute the neighborhood of each atom A, belonging to the same cluster
//  than A, on a radius of value 'radi'.
//
//  Parameters:
//                      atoms       Pointer to the array where all atoms (coordinates, type, classID)
//                                  are stored.
//                      nAtoms      The number of atoms.
//                      cluster     Pointer to the array of Arraylist's where the clusters (atom's
//                                  position ONLY) are stored.
//                      radi        The radius used to define the neighborhood of each atom.
//                      N           Pointer to the array of 'Arraylist' where atoms' neighborhood
//                                  will be stored (atom's position ONLY).
//                      latP        Pointer to the lattice parameters.
//
//  Return: ----
//
void getNeighborsInCluster (point3Dclassified *atoms, int nAtoms, ArrayList *cluster,
                    int nClusters, float radi, ArrayList* N, LatticeParameters *latP) {

        int         i, j, nc, sizeC, posI, posJ;
        float   dist, lx, ly, lz;
        Element *e;

        // Simulation cell size ----------------------------------------------------

        lx = (float)(latP->nbx * latP->lc);
        ly = (float)(latP->nby * latP->lc);
        lz = (float)(latP->nbz * latP->lc);

        for (nc=0;nc<nClusters;++nc) {
                sizeC = size(cluster+nc);

                for (i=0;i<sizeC;++i) {

                        for(j=0;j<sizeC;++j) {
                                if (j != i) {

                                        e       = get(cluster+nc, i);
                                        posI    = (e!=NULL) ? e->data : -1;

                                        e       = get(cluster+nc, j);
                                        posJ    = (e!=NULL) ? e->data : -1;

                                        // Compute the Euclidean distance between atoms i,j --------
                                        dist = euclideanDistancePBC (*(atoms+posI), *(atoms+posJ),
                                                lx,ly,lz);

                                        if(dist <= radi) {
                                                add(N+posI, (Element){posJ});
                                        }
                                }
                        }
                }
        }
}
```

```
// ****************************************************************************
//  Given an array of classified/clustered points, organize them in an array of  ArrayList's, with the
//  atoms of each cluster stored in a different position (ArrayList) of the array.
//
//  Parameters:
//          atoms       Pointer to the array where the atoms (coordinates, type, classID) are stored.
//          nAtoms   The number atoms.
//          cluster     Pointer to the array of Arraylist's where the clusters (atoms' position only) will be stored.
//
//  Return: ----
//
void organizeClusters (point3Dclassified *atoms, int nAtoms, ArrayList *cluster) {
        int          i, cid;

        for(i=0;i<nAtoms;++i) {
                cid = atoms[i].classId;
                if( (cid!=NOISE) && (cid!=UNCLASSIFIED) ) {
                        add(cluster+cid, (Element){i});
                }
        }
}

// Compute and store in an array the (approximated) radius of all clusters.
//
//  Parameters:
//          sizeCluster         Pointer to the array that stores the size of the clusters.
//          nClusters           Number of clusters.
//          lattice             Aluminum lattice constant.
//
//  Return: Pointer to the array that stores the (approximated) radius of the clusters.
//
double* radiusClusters (int *sizeCluster, int nClusters, double latticeC) {

        int         nc;
        double    *radiusCluster, radiusC;

        if ((radiusCluster = (double *)malloc(sizeof(double)*nClusters)) == NULL) {
                printf("Error on memory allocation for 'radiusCluster'\n");
                return NULL;
        }

        for (nc=0;nc<nClusters;++nc) {
                radiusC  = (double)(3*sizeCluster[nc]);
                radiusC /= (double)(8.0*PI);
                radiusC  = cbrt(radiusC)*latticeC;
                radiusCluster[nc] = radiusC;
        }
        return radiusCluster;
}

void printRadiusClusters (double *radiusCluster, int nClusters) {
        int          nc;

        printf("****************************************************************************\n");
        for (nc=0;nc<nClusters;++nc) {
                printf("Radius of cluster %d is %.2f Angstrom\n", nc, radiusCluster[nc]);
        }
}
```

```
// Compute the average radius among all clusters.
//
//  Parameters:
//          radiusCluster       Pointer to the array that stores the (approximated) radius of the clusters.
//          nClusters           Number of clusters.
//
//  Return: The average radius among all clusters.
//
double averageRadiusClusters (double *radiusCluster, int nClusters) {

        int         nc;
        double      avgRadius=0.0;

        for (nc=0;nc<nClusters;++nc) {
                avgRadius +=radiusCluster[nc];
        }
        avgRadius /= (double)nClusters;

        return avgRadius;
}

// Compute and store in an array the size of all clusters.
//
//  Parameters:
//          atoms       Pointer to the array where all atoms are stored.
//          nAtoms      The number atoms.
//          cluster     Pointer to the array of Arraylist's where the clusters (atom's position only) are stored.
//          nClusters   Number of clusters
//
//  Return: Pointer to the array that stores the size of the clusters.
//
int* sizeClusters(point3Dclassified *atoms, int nAtoms, ArrayList *cluster,
        int nClusters) {

        int  nc, *sizeCluster;

        if ((sizeCluster = (int *)malloc(sizeof(int)*nClusters)) == NULL) {
                printf("Error on memory allocation for 'sizeCluster'\n");
                return NULL;
        }

        for (nc=0;nc<nClusters;++nc) {
                sizeCluster[nc] = size(cluster+nc);
        }
        return sizeCluster;
}

void printSizeClusters(int *sizeCluster, int nClusters) {
        int           nc;

        printf("*********************************************************************************\n");
        for (nc=0;nc<nClusters;++nc) {
                printf("Size of cluster %d is %d\n", nc, sizeCluster[nc]);
        }
}
```

```
// Compute the average size among all clusters.
//
//  Parameters:
//          sizeCluster         Pointer to the array that stores the size of the clusters.
//          nClusters           Number of clusters.
//
//  Return: The average size among all clusters.
//
int averageSizeClusters (int *sizeCluster, int nClusters) {

        int     avgSize=0, nc;

        for (nc=0;nc<nClusters;++nc) {
                avgSize +=sizeCluster[nc];
        }
        avgSize /= nClusters;

        return avgSize;
}

// Merge the clusters that are split in several parts in a single spatial region per cluster.
//
//  Parameters:
//                  atoms    Pointer to the array where all atoms are stored.
//                  nAtoms   The number atoms.
//                  cluster  Pointer to the array of Arraylist's where the clusters (atom's
//                           position only) are stored.
//                  nClusters Number of clusters
//                  N        Pointer to the array of Arraylist where atoms' neighborhood
//                           (inside a cluster) are stored.
//                  latP     Pointer to the lattice parameters.
//
//  Return: ----
//
void mergeClusters (point3Dclassified *atoms, int nAtoms, ArrayList *cluster,
        int nClusters, ArrayList *N, LatticeParameters *latP) {

        int     nc, i, v, sizeC, sizeNi, posI, posV, nMoves;
        float   lx, ly, lz, lx_half, ly_half, lz_half;
        float   delta, minus_lx_half, minus_ly_half, minus_lz_half;
        Element  *e;

        lx          = (float)(latP->nbx * latP->lc);
        ly          = (float)(latP->nby * latP->lc);
        lz          = (float)(latP->nbz * latP->lc);
        lx_half     = lx/2.0;
        ly_half     = ly/2.0;
        lz_half     = lz/2.0;
        minus_lx_half = -lx_half;
        minus_ly_half = -ly_half;
        minus_lz_half = -lz_half;

        for (nc=0;nc<nClusters;++nc) {

                sizeC = size(cluster+nc);

                do {
                        nMoves = 0; // number of moved atoms
```

```
                        for (i=0;i<sizeC;++i) {

                                e = get(cluster+nc, i);
                                posI = (e!=NULL) ? e->data : -1;
                                sizeNi = size(N+posI);

                                for (v=0;v<sizeNi;++v) {

                                        e       = get(N+posI, v);
                                        posV    = (e!=NULL) ? e->data : -1;

                                        delta   = atoms[posI].x - atoms[posV].x;
                                        if(delta > lx_half) {
                                                atoms[posV].x += lx;
                                                ++nMoves;
                                        }
                                        else if(delta < minus_lx_half) {
                                                atoms[posI].x += lx;
                                                ++nMoves;
                                        }

                                        delta = atoms[posI].y - atoms[posV].y;
                                        if(delta > ly_half) {
                                                atoms[posV].y += ly;
                                                ++nMoves;
                                        }
                                        else if(delta < minus_ly_half) {
                                                atoms[posI].y += ly;
                                                ++nMoves;
                                        }

                                        delta = atoms[posI].z - atoms[posV].z;
                                        if(delta > lz_half) {
                                                atoms[posV].z += lz;
                                                ++nMoves;
                                        }
                                        else if(delta < minus_lz_half) {
                                                atoms[posI].z += lz;
                                                ++nMoves;
                                        }
                                }
                        }
                } while (nMoves>0);     // repeat the process until there is a cycle over
                                        // all cluster's atoms without moving any atom
        }
}

// ****************************************************************************
//  Given an array of ArrayList's, with the atoms of each cluster stored in a
//  different position (ArrayList) of the array, this function removes the clusters
//  that have a size smaller than a given threshold. The function also redefines
//  the indexes of the remaining clusters to have a continuous set of indexes.
//
//  Parameters:
//                  atoms   Pointer to the array where the atoms (coordinates, type, classID) are stored.
//                  nAtoms  The number atoms.
//                  cluster Pointer to the array of Arraylist's where the clusters (atom's
//                          position only) are stored.
```

```
//                      nClusters Number of clusters
//                      minSize   Minimum number of atoms that a cluster must have to don't be removed.
//
//  Return:               The new number of clusters, after removing the small ones.
//
int removeSmallClusters(point3Dclassified *atoms, int nAtoms, ArrayList *cluster,
        int nClusters, int minSize) {

        int         nRemoved = 0, nc, i, newNclusters, pos, index2assign, sizeC;
        booleano *removedCluster, stop;
        Element  *e;

        if (nClusters == 0)
                return 0;

        if ((removedCluster = (booleano *)malloc(sizeof(booleano)*nClusters)) == NULL) {
                printf("  Error on memory allocation for 'removedCluster'\n");
                return -1;
        }

        // Initially setup all clusters with a flag "don't remove"
        for (nc=0;nc<nClusters;++nc) {
                removedCluster[nc] = FALSE;
        }

        printf("  Identifying the clusters to be romoved ...\n");

        // Identify the clusters to be romoved and mark their atoms as "NOISE" (unclustered)
        for (nc=0;nc<nClusters;++nc) {
                sizeC = size(cluster+nc);

                if(sizeC<minSize) {
                        for(i=0;i<sizeC;++i){
                                e  = get(cluster+nc, i);
                                pos   = (e!=NULL) ? e->data : -1;
                                atoms[pos].classId = NOISE;
                        }
                        removedCluster[nc] = TRUE;
                        ++nRemoved;
                }
        }

        // Go to the first cluster to be removed
        nc   = 0;
        stop = FALSE;
        do {
                if( (removedCluster[nc]==TRUE) || (nc==nClusters) )
                        stop = TRUE;
                else
                        ++nc;
        } while (stop==FALSE);

        // Redefine the indexes of the clusters that will not be removed, in order
        // to have a continuous set of indexes
        printf("  Redefining the indexes of the clusters that will not be removed...\n");
        index2assign = nc;
        for(;nc<nClusters;++nc) {
```

```
		if(removedCluster[nc]!=TRUE) {
			sizeC = size(cluster+nc);
			for(i=0;i<sizeC;++i){
				e   = get(cluster+nc, i);
				pos   = (e!=NULL) ? e->data : -1;
				atoms[pos].classId = index2assign;
			}

			printf("  * cluster %d (size %d) -> cluster %d\n", nc, sizeC, index2assign);
			++index2assign;
		}
	}

	// Remove each cluster (ArrayList) from the array 'cluster' (array of ArrayList's)
	// and adjust the size of the array ('newNclusters')
	printf("  Removing clusters marked to be deleted from the array of clusters...\n");
	newNclusters = nClusters;
	for(nc=nClusters-1;nc>=0;--nc) {
		if(removedCluster[nc]==TRUE) {
			printf("  * cluster %d\n", nc);
			newNclusters = removeFromArrayAt(cluster, newNclusters, nc);
		}
	}

	free(removedCluster);

	return (newNclusters);
}

// ****************************************************************************
//  Given an array of ArrayList's, with the atoms of each cluster stored in a
//  different position (ArrayList) of the array, this function counts the clusters
//  that have the same size. It only counts clusters with a size between [minPts:minSize].
//
//  Parameters:
//		cluster	Pointer to the array of Arraylist's where the clusters (atom's
//			position only) are stored.
//		nClusters Number of clusters
//		clustP	Pointer to the clustering parameters.
//		aRP	Pointer to the clustering analysis report structure.
//
//  Return:		------
//
void countNumberSmallClusters(ArrayList *cluster, int nClusters,
		ClusteringParameters *clustP, AnalysisReportParam *aRP) {

	int	nc, n, sizeC;

	// Initialize to zero the number of cluster with all sizes

	for (n=0;n<=clustP->minSize;++n) {
		*(aRP->nClustersSize+n) = 0;
	}

	// Count the number of clusters with each size in the range [minPts:minSize]
	for (nc=0;nc<nClusters;++nc) {
		sizeC = size(cluster+nc);
```

```
                if( (sizeC>=clustP->minPts) && (sizeC<=clustP->minSize) ) {
                        *(aRP->nClustersSize+sizeC) += 1;
                }
        }
}
```

vii. cnt_config.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "cnt_includes.h"

// Date: 05 October 2012

// ********************************************************************************
// Read a text line from an ASCII file, until NEWLINE or EOF is found.
//
//      Parameters:
//              fp      File pointer to open file.
//              buffer  Buffer where characters will be written.
//
//      Return: TRUE (didn't find EOF), FALSE (found EOF).
//
booleano readLine (FILE *fp, char *buffer) {
        int   n=0;
        char  ch=0;
        if(fp != NULL) {
                while ( (!feof(fp)) && (ch!=NEWLINE) && (n<MAX_LINE_SIZE) ) {
                        fscanf(fp,"%c",&ch);
                        buffer[n]=ch;
                        ++n;
                }
                --n;
                buffer[n]='\0';

                if((n>0)&&(buffer[n-1]==ENTER)) // to remove the ENTER at end of a line
                        buffer[n-1]=='\0';

                if(n==MAX_LINE_SIZE) { // line size exceeds the limit
                        while ( (!feof(fp)) && (ch!=NEWLINE) ) { // discard the rest of line
                                fscanf(fp,"%c",&ch);
                        }
                }
        }

        if (feof(fp)) return(FALSE);
        else return(TRUE);
}

// ********************************************************************************
// Split a text line, read from configuration file, in a parameter tuple: (key,number,listValues).
//
// Parameters:
//      line    Pointer to the string that contains the line.
//      kv      Pointer to the structure where the tuple (key,number,listValues) will be saved.
//
```

```
// Return: TRUE (if tuple was found), FALSE (if tuple wasn't found).
//
booleano splitLine(char *line, KeyNumberValues *kv) {
	char	 key[MAX_LINE_SIZE], value[MAX_LINE_SIZE];
	int	 n=0, i=0, n1, nv, nf;
	booleano stop;

	while( (line[n]==' ') || (line[n]=='\t') ) {
		n++; // skip white chars (' ', '\t')
		if (n==strlen(line))
			return (FALSE);
	}

	while( (line[n]!=' ') && (line[n]!='=') ) { // extract parameter name (key)
		kv->key[i] = line[n];
		++n;
		++i;
		if (n==strlen(line))
			return (FALSE);
	}
	kv->key[i]='\0';

	if (line[n] != '=')
		while( line[n]!='=') {
		n++; // find key-value separator ("=")
		if (n==strlen(line))
			return (FALSE);
	}
	++n;

	// Count the number of strings after the "=" and before the "#" ----------

	kv->nValues = 0; // number of values to read from line
	n1 = n;
	stop = FALSE;
	do {
		value[0]='\0';		// clean up string 'value'

		while( (line[n1]==' ') || (line[n1]=='\t') ) {
			n1++; // skip white chars (' ', '\t')
			if (n1==strlen(line))
				return (FALSE);
		}

		i=0;
		while( (line[n1]!=' ') && (line[n1]!='\t') && (line[n1]!='#') &&
				(n1<strlen(line)) ) { // extract string
			value[i] = line[n1];
			++n1;
			++i;
		}
		value[i]='\0';
		if (strlen(value)>0)
			++kv->nValues;
		if ( (line[n1]=='#') || (n1==strlen(line)) )
			stop = TRUE;
	} while (stop==FALSE);
```

```
	// Extract the 'kv->nValues' values from line ------------------------------

	if((kv->value = (char**)malloc(sizeof(char*)*kv->nValues)) == NULL){
		printf("Error when allocating memory to kv->value in splitLine()!\n");
		return (FALSE);
	}

	for (nv=0;nv<kv->nValues;++nv){
		value[0]='\0';		// clean up string 'value'

		while( (line[n]==' ') || (line[n]=='\t') ) {
			n++; // skip white chars (' ', '\t')
		}

		i=0;
		while( (line[n]!=' ') && (line[n]!='\t') && (line[n]!='#') &&
				(n<strlen(line)) ) { // extract string
			value[i] = line[n];
			++n;
			++i;
		}
		value[i]='\0';
		if((kv->value[nv]=(char*)malloc(sizeof(char)*(strlen(value)+1))) == NULL){
			printf("Error when allocating memory to kv->value[%d] in splitLine()!\n",nv);
			// Free allocated memory on error ---------------------------------
			for (nv=nv-1;nv>=0;--nv){
				free(kv->value[nv]);
			}
			free(kv->value);
			return (FALSE);
		}
		strcpy(kv->value[nv],value);
	}

	return(TRUE);
}

// ********************************************************************************
// Identify the parameter read from configuration file, stored in the 'kv'
// (key,number,listValues) tuple, and assign its value(s) to the adequate structure field.
//
//		Parameters:
//			kv		Pointer to the (key,number,listValues) tuple that was read from configuration file.
//			cntP		Pointer to structure where the Classic Nucleation Theory parameters will be saved.
//
//	Return:		TRUE  (if the parameter was correctly identified and stored) OR
//			FALSE (if an error occurred on identifying/storing the parameter)
//
booleano identifyParameter (KeyNumberValues *kv, CntParameters *cntP) {

	int  nv;
	booleano success = TRUE;

	if(strcmp(kv->key,"alfa")==0){
		cntP->alfa = atof(kv->value[0]);
	}
	if(strcmp(kv->key,"kB")==0){
```

```
		cntP->kB = atof(kv->value[0]);
	}
	if(strcmp(kv->key,"Ns")==0){
		cntP->Ns = atoi(kv->value[0]);
	}
	if(strcmp(kv->key,"nSc_critical")==0){
		cntP->nSc_critical = atoi(kv->value[0]);
	}

	if(strcmp(kv->key,"T")==0){
		if((cntP->T = (double *)malloc(sizeof(double)*kv->nValues)) == NULL){
			printf("Error when allocating memory to 'cntP->T' in identifyParameter()!\n");
			success = FALSE;
		}
		else {
			cntP->num_T = kv->nValues;
			for (nv=0;nv<kv->nValues;++nv){
				cntP->T[nv] = atof(kv->value[nv]);
			}
		}
	}

	if(strcmp(kv->key,"X0_Sc")==0){
		if((cntP->X0_Sc = (double *)malloc(sizeof(double)*kv->nValues)) == NULL){
			printf("Error when allocating memory to 'cntP->X0_Sc' in identifyParameter()!\n");
			success = FALSE;
		}
		else {
			cntP->num_X0_Sc = kv->nValues;
			for (nv=0;nv<kv->nValues;++nv){
				cntP->X0_Sc[nv] = atof(kv->value[nv]);
			}
		}
	}

	if(strcmp(kv->key,"time")==0){
		if((cntP->time = (double *)malloc(sizeof(double)*kv->nValues)) == NULL){
			printf("Error when allocating memory to 'cntP->time' in identifyParameter()!\n");
			success = FALSE;
		}
		else {
			cntP->num_time = kv->nValues;
			for (nv=0;nv<kv->nValues;++nv){
				cntP->time[nv] = atof(kv->value[nv]);
			}
		}
	}

	if(strcmp(kv->key,"T_concentration")==0){
		cntP->T_concentration = atof(kv->value[0]);
	}
	if(strcmp(kv->key,"X0_Sc_concentration")==0){
		cntP->T_concentration = atof(kv->value[0]);
	}

	// Free allocated memory ---------------------------------------------------
	for (nv=0;nv<kv->nValues;++nv){
		free(kv->value[nv]);
```

```
	}
	free(kv->value);

	return (success);
}

// ****************************************************************************
//  Read configuration file.
//
//  Parameters:
//		fName	Configuration file name.
//		cntP	 Pointer to the Classic Nucleation Theory parameters.
//
//  Return: TRUE (if reading file succeeds), FALSE (if reading file fails).
//
booleano readConfigFile (char *fName, CntParameters *cntP) {

	char		line[MAX_LINE_SIZE];
	FILE		*fp;
	booleano	continueReading;
	KeyNumberValues  kv;

	if((fp = fopen(fName,"r"))==NULL){
		printf("Error when trying to open configuration file %s\n", fName);
		return FALSE;
	};

	do {
		continueReading = readLine(fp, line);
		if(strlen(line)>1) {
			if(splitLine(line, &kv) == FALSE)
				printf("Error on parsing configuration line!\n");
			else {
				identifyParameter(&kv, cntP);
			}
		}
	} while (continueReading == TRUE);

	fclose(fp);
	return TRUE;
}

// ****************************************************************************
//  Print the parameters for the classic nucleation teory (CNT) application
//
void printCntParameters (CntParameters *cntP){
	int n;

	printf(" * Aluminum lattice constant                                = %f\n",cntP->alfa);
	printf(" * Boltzmann Constant (ElectronVolt/Kelvin)                 = %.7e\n",cntP->kB);
	printf(" * Number of nucleation sites                               = %d\n",cntP->Ns);
	printf(" * Critical size (number Sc atoms in a stable precipitate)  = %d\n",cntP->nSc_critical);

	printf(" * Simulation temperatures (Kelvin) = ");
	for (n=0;n<cntP->num_T;++n){
		printf("%.3f  ",cntP->T[n]);
	}
	printf("\n");
```

```
	printf(" * Sc nominal concentrations = ");
	for (n=0;n<cntP->num_X0_Sc;++n){
		printf("%.4f  ",cntP->X0_Sc[n]);
	}
	printf("\n");

	printf(" * Time points where precipitates concentration will be evaluated = ");
	for (n=0;n<cntP->num_time;++n){
		printf("%.3f  ",cntP->time[n]);
	}
	printf("\n");

	printf(" * Number of Sc atoms in the precipitate = ");
	for (n=0;n<cntP->nSc_critical;++n){
		printf("%d  ",cntP->nSc[n]);
	}
	printf("\n");
	printf(" * Temperature used for computing precipitate concentration = %.3f\n",cntP->T_concentration);
	printf(" * Sc nominal concentration used for computing time evolution of precipitate concentration =
		%.3f\n",cntP->X0_Sc_concentration);
}
```

viii. cnt_functions1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "cnt_includes.h"

// Date: 04 October 2012

/* ************************************************************************
 * Compute the second nearest neighbors interaction between an Al and an Sc atoms:
 * W(2)AlSc [equation CNT8].
 *
 * Parameters:
 *	T	Temperature.
 *	k	Boltzman constant (in "eV").
 *	Xeq_Sc	Solubility limit.
 *
 * Return: 2nd nearest neighbors interaction between an Al and an Sc.
 *
 */
double W_2_AlSc (double T, double k, double Xeq_Sc) {

	double res, aux1, aux2, aux3;

	aux1 = 2.0/3.0;
	aux2 = 8.0/3.0;
	aux3 = k*T*(pow(Xeq_Sc,aux1)-aux2*Xeq_Sc);
	res = (k*T*log(Xeq_Sc))/6.0+aux3;

	return res;
}
```

```
/* Compute the second nearest neighbors interaction between an Al and an Sc atoms:
 * W(2)AlSc [using E(2)AlSc from Table 1, paper Clouet 2004].
 *
 * Parameters:
 *          T           Temperature.
 *
 * Return: 2nd nearest neighbors interaction between an Al and an Sc (in eV).
 *
 */
double W_2_AlSc_V1 (double T) {

        double res;

        res = 0.113 - 0.0000334*T;

        return res; // eV
}

/* ***************************************************************************
 * Compute the isotropic free energy: alfa^2*sigma [equation CNT12].
 *
 * Parameters:
 *          T           Temperature.
 *
 * Return: the isotropic free energy (in "eV").
 *
 */
double isotropicFreeEnergy (double T) {

        double alfa2sigma;

        alfa2sigma = (139.0 - 0.03178*T - 0.0000154*T*T)/1000.0; // result in "eV"

        return alfa2sigma;
}

/* ***************************************************************************
 * Compute the isotropic free energy: alfa^2*sigma [equation 13, paper Clouet 2007].
 *
 * Parameters:
 *          T           Temperature.
 *          k            Boltzman constant (in "eV").
 *
 * Return: the isotropic free energy (in "eV").
 *
 */
double isotropicFreeEnergy_v1(double T, double k) {

        double alfa2sigma, w2AlSc, aux1, aux2, aux3, kB_T;

        kB_T            = k * T;
        w2AlSc          = W_2_AlSc_V1(T);
        aux2            = 6.0/PI;
        aux3            = 1.0/3.0;
        aux1            = pow(aux2,aux3);
        alfa2sigma      = aux1;
```

```
	aux3		= (-4.0*w2AlSc)/kB_T;
	aux2		= w2AlSc-2.0*kB_T*exp(aux3);
	aux3		= (-6.0*w2AlSc)/kB_T;
	aux1		= kB_T*exp(aux3);
	aux3		= aux2-aux1;
	alfa2sigma	*= aux3;

	return alfa2sigma; // result in "eV"
}

/* ************************************************************************
 * Compute the precipitate concentration at a given time, for a specific
 * temperature and nominal concentration <=> the probability to find a
 * precipitate with nSc Sc atoms at a given time: C_nSc_t(X0_sc,T) [equation CNT16].
 *
 * Parameters:
 *	t		Time.
 *	Jst_nSc	Steady-state nucleation rate.
 *
 * Return: the precipitate concentration at time 't'.
*/
double precipitateConcentration_timeEvol (double t, double Jst_nSc) {
	return (t*Jst_nSc);
}

/* ************************************************************************
 * Compute the precipitate concentration <=> the probability to find a
 * precipitate with nSc Sc atoms : C_nSc(X0_sc,T) [equation CNT1].
 *
 * Parameters:
 *	nSc	 Number of Sc atoms in the cluster/precipitate.
 *	X0_Sc	Sc nominal concentration.
 *	T		Temperature.
 *	k		Boltzman constant (in "eV").
 *
 * Return: the probability to find a precipitate with nSc Sc atoms.
 *
 */
double precipitateConcentration (int nSc, double X0_Sc, double T, double k) {
	double dGnSc, aux, res;

	dGnSc = formationFreeEnergy(nSc, X0_Sc, T, k);
	aux   = - dGnSc/(k*T);
	res   = exp(aux);

	return res;
}

/* ************************************************************************
 * Compute the solubility limit of Sc in Al: Xeq_Sc [equation CNT2].
 *
 * Parameters:
 *	T		Temperature.
 *	k		Boltzman constant (in "eV").
 *
 * Return: the solubility limit of Sc in Al.
 *
 */
```

```
double solubilityLimit_Sc_Al (double T, double k) {
	double aux, Xeq_Sc;

	aux	= -0.701 + 0.00023*T;
	aux	= aux / (k*T);
	Xeq_Sc	= exp(aux);

	return	Xeq_Sc;
}

/* ***************************************************************************
 * Compute the solubility limit of Sc in Al: Xeq_Sc [equation 7, paper clouet 2007].
 *
 * Parameters:
 *	T	Temperature.
 *	k	Boltzman constant (in "eV").
 *
 * Return: the solubility limit of Sc in Al.
*/
double solubilityLimit_Sc_Al_V1(double T, double k) {
	double Xeq_Sc, w_2_AlSc, w2_kb_T, aux1, aux2;

	w_2_AlSc  = W_2_AlSc_V1(T);
	w2_kb_T   = w_2_AlSc/(k*T);

	aux1	= -6.0*w2_kb_T;
	Xeq_Sc	= exp(aux1);

	aux1	= -10.0*w2_kb_T;
	aux2	= 6.0*exp(aux1);
	Xeq_Sc  += aux2;

	aux1	= -12.0*w2_kb_T;
	aux2	= 16.0*exp(aux1);
	Xeq_Sc  -= aux2;

	return	Xeq_Sc;
}

/* ***************************************************************************
 * Compute the Sc impurity diffusion coefficient: D_Sc [equation CNT3].
 *
 * Parameters:
 *	T	Temperature.
 *	k	Boltzman constant (in "eV").
 *
 * Return: the Sc diffusion coefficient.
*/
double diffusionCoefficient_Sc (double T, double k) {
	double aux, aux1, Dsc, k_T;

	k_T  = k*T;
	aux  = -1.79/k_T;
	aux1 = pow(MATH_E,aux);
	Dsc  = 0.000531*aux1;

	return	Dsc;
}
```

```
/* ***************************************************************************
 * Compute the formation free energy: deltaG_nSc(X0Sc) [equation CNT4].
 *
 * deltaG_nSc =4*n_Sc*deltaG_nuc(X0_Sc)+(36*PI)^(1/3)*n_Sc^(2/3)*a^2*sigma_nSc
 *
 * Parameters:
 *          nSc         Number of Sc atoms in the cluster/precipitate.
 *          X0_Sc       Sc nominal concentration.
 *          T           Temperature.
 *          k           Boltzman constant (in "eV").
 *
 * Return: the formation free energy.
 *
 */
double formationFreeEnergy (int nSc, double X0_Sc, double T, double k) {

          double   res, a2sigma, dG_nuc, aux1, aux2, aux3;

          a2sigm     = isotropicFreeEnergy(T);
          dG_nuc   = nucleationFreeEnergy_3rdLTE(T, k, X0_Sc);
          res          = 4.0*nSc*dG_nuc;
          aux1       = 36.0*PI;
          aux2       = 1.0/3.0;
          aux3       = pow(aux1,aux2);
          aux1       = 2.0/3.0;
          aux2       = pow(nSc,aux1);
          aux1       = aux3*aux2*a2sigma;
          res        += aux1;

          return res;
}

/* ***************************************************************************
 * Compute q(x) needed by the nucleation free energy calculation through the
 *   3rd order Low Temperature Expansion: q(x) [equation CNT7].
 *
 * Parameters:
 *          x                   Sc concentration (can be X0_Sc or Xeq_Sc).
 *          T                   Temperature.
 *          k                   Boltzman constant (in "eV").
 *          w_2_AlSc            2nd nearest neighbors interaction between an Al and an Sc.

 *
 * Return: the q(x) term needed by 3rd order LTE.
 *
 */
double qx_3rdLTE (double x, double T, double k, double w_2_AlSc) {

          double qx, aux1, aux2;

          aux1 = (2.0*w_2_AlSc)/(k*T);
          aux2 = 1.0+4.0*x*(6.0*exp(aux1)-19.0);
          aux1 = 1.0+sqrt(aux2);
          qx   = (2.0*x)/aux1;

          return qx;
}
```

```
/* ***************************************************************************
 * Compute the nucleation free energy, using 3rd order Low Temperature Expansion:
 *          deltaG_nuc(X0Sc) [equation CNT6].
 *
 * Parameters:
 *          T                   Temperature.
 *          k                   Boltzman constant (in "eV").
 *          X0_Sc               Sc nominal concentration.
 *
 * Return: the nucleation free energy.
 *
 */
double nucleationFreeEnergy_3rdLTE(double T, double k, double X0_Sc) {

          double dGnuc, Xeq_Sc, w_2_AlSc, q_X0_sc, q_Xeq_sc, aux1, aux2;

          Xeq_Sc   = solubilityLimit_Sc_Al_V1(T, k);
          w_2_AlSc = W_2_AlSc_V1(T);

          q_X0_sc  = qx_3rdLTE(X0_Sc, T, k, w_2_AlSc);

          q_Xeq_sc = qx_3rdLTE(Xeq_Sc, T, k, w_2_AlSc);

          dGnuc    = k*T*(q_X0_sc - q_Xeq_sc);

          aux1       = (2.0*w_2_AlSc)/(k*T);
          aux2       = 3.0*k*T*exp(aux1)*(q_X0_sc*q_X0_sc-q_Xeq_sc*q_Xeq_sc);
          dGnuc   += aux2;
          aux1       = log(q_X0_sc)-log(q_Xeq_sc);
          aux2       = 0.25 * k * T * aux1;
          dGnuc   -= aux2;

          return dGnuc;
}

/* ***************************************************************************
 * Compute the Zeldovitch factor: Z(nSc) [equation CNT14].
 *
 * Parameters:
 *          dG_nSc_X0Sc      Formation free energy.
 *          alfa2sigma       Isotropic free energy.
 *          T                Temperature.
 *          k                Boltzman constant (in "eV").
 *
 * Return: the computed Zeldovitch factor.
 *
 */
double zeldovitchFactor (double dG_nSc_X0Sc, double alfa2sigma, double T, double k) {

          double res, aux1, aux2, aux3;

          res  = dG_nSc_X0Sc * dG_nSc_X0Sc;

          aux1 = 3.0/2.0;
          aux2 = 2.0*PI*pow(alfa2sigma,aux1);
          aux1 = k*T;
          aux3 = sqrt(aux1);
```

```
	aux1 = aux2 * aux3;

	res  /= aux1;

	return res;
}

/* ***************************************************************************
 * Compute the condensation rate of precipitates: beta*(nSc) [equation CNT15].
 *
 * Parameters:
 *		dG_nSc_X0Sc	Formation free energy.
 *		alfa2sigma	Isotropic free energy.
 *		X0_Sc		Sc nominal concentration.
 *		alfa		Al lattice constant.
 *		T		Temperature.
 *		k		Boltzman constant (in "eV").
 *
 * Return: the computed condensation rate.
 *
 */
double condensationRate (double dG_nSc_X0Sc, double alfa2sigma, double X0_Sc,
			double alfa, double T, double k) {

	double res, aux1, aux2;

	aux1 = diffusionCoefficient_Sc(T, k); // Dsc
	aux2 = alfa2sigma / dG_nSc_X0Sc;
	res  = -32.0 * PI * aux2;
	aux2 = aux1/(alfa*alfa);
	res  = res * aux2 * X0_Sc;

	return res;
}

/* ***************************************************************************
 * Compute the steady-state nucleation rate: Jst(nSc) [equation 14, paper Clouet 2007].
 *
 * Parameters:
 *		X0_Sc	Sc nominal concentration.
 *		alfa	Al lattice constant.
 *		Ns	Number of nucleation sites.
 *		T	Temperature.
 *		k	Boltzman constant (in "eV").
 *
 * Return: the computed steady-state nucleation rate.
 *
 */
double steadyStateNucleationRate_V1 (double X0_Sc, double alfa, int Ns,
			double T, double k) {

	double   aux1, aux2, dG_nuc, alfa2sigma, Dsc, Jst;

	alfa2sigma  = isotropicFreeEnergy_v1(T, k);

	dG_nuc    = nucleationFreeEnergy_3rdLTE(T, k, X0_Sc);
	Dsc       = diffusionCoefficient_Sc(T, k);
```

```
	Jst			= -16*Ns;

	aux2			= k*T*alfa2sigma;
	aux1			= dG_nuc/sqrt(aux2);
	Jst			*= aux1;

	aux1			= (X0_Sc*Dsc)/(alfa*alfa);
	Jst			*= aux1;

	aux2			= -PI*pow(alfa2sigma, 3.0);
	aux1			= 3.0*k*T*dG_nuc*dG_nuc;
	aux2			= aux2/aux1;
	aux1			=exp(aux2);
	Jst			*= aux1;

	// compute the Jst per 1 cubic meter of volume

	aux1			= (double)Ns/4.0;    // number of lattice cells
	aux2			= pow(alfa,3.0)*aux1; // volume of all cells

	Jst			/= aux2;  // divide the total rate by the volume

	return Jst;
}

double steadyStateNucleationRate (int nSc, double X0_Sc, double alfa, int Ns,
			double T, double k) {

	double   aux1, aux2, dG_nSc_X0Sc, alfa2sigma, Zfactor, beta;

	alfa2sigma		= isotropicFreeEnergy(T);
	dG_nSc_X0Sc	= formationFreeEnergy(nSc, X0_Sc, T, k);
	Zfactor		= zeldovitchFactor(dG_nSc_X0Sc, alfa2sigma, T, k);
	beta			= condensationRate(dG_nSc_X0Sc, alfa2sigma, X0_Sc, alfa, T, k);
	aux1			= - dG_nSc_X0Sc/(k*T);
	aux2			= exp(aux1);
	aux1			= Ns * Zfactor * beta * aux2;

	return aux1;
}
```

ix. cnt_includes.h

```
#include <stdio.h>

// Date: 05 October 2012

// CONSTANTS ***********************************************************

#define PI			(double)3.14159265358979323846264338327
#define MATH_E		(double)2.71828182845904523536028747135
#define MAX_LINE_SIZE	1000  // Maximum size of a line to read from file
#define NEWLINE		10
#define ENTER		13

typedef enum {FALSE, TRUE} booleano;
```

```
typedef struct cnt_parameters {
        double          alfa;           // Aluminum lattice constant (Aº)
        double          kB;             // Boltzmann Constant (eV/K)
        int             Ns;             // Number of nucleation sites
        int             nSc_critical;   // Critical size (min. number of Sc atoms in a
                                        //      stable precipitate)
        int             num_T;          // Number of "simulation temperatures"
        double          *T;             // Simulation temperatures (Kelvin)
        int             num_X0_Sc;      // Number of "Sc nominal concentrations"
        double          *X0_Sc;         // Sc nominal concentrations
        int             num_time;       // Number of "time points where precipitates
                                        //      concentration will be evaluated"
        double          *time;          // Time points where precipitates concentration
                                        //      will be evaluated
        int             *nSc;           // Number of Sc atoms in the precipitate (1..nSc_critical)
        double          T_concentration;   // Temperature used for computing precipitate concentration
        double          X0_Sc_concentration; // Sc nominal concentration used for computing
                                        // time evolution of precipitate concentration
} CntParameters;

typedef struct cnt_results {
        double  *deltaG_nuc;            // Nucleation free energy
        double  *Dsc;                   // Sc diffusion coefficient
        double  *Jst;                   // Steady-state nucleation rate
        double  *a2sigma;               // Isotropic free energy
        double  *C_nSc_X0;              // Precipitates concentration as a function of nominal concentration
        double  *C_nSc_t;               // Precipitates concentration as a function of time
} CntResults;

typedef struct key_num_values { // Tuple (KEY, NUMBER OF VALUES, LIST OF VALUES)
        char key[MAX_LINE_SIZE+1];
        int  nValues;
        char **value;       // pointer to an array that will store 'nValues' elements,
                            // each one is an array of variable length (maximum MAX_LINE_SIZE+1)
} KeyNumberValues;

// FUNCTIONS ***************************************************

void        graphNucleationFreeEnergy(CntParameters *, CntResults *, FILE *);
void        graphDiffusionCoefficient(CntParameters *, CntResults *, FILE *);
void        graphIsotropicFreeEnergy(CntParameters *cntP, CntResults *, FILE *);
void        graphSteadyStateNucleationRate(CntParameters *, CntResults *, FILE *);
void        graphPrecipitateConcentration_Xc(CntParameters *, CntResults *, FILE *);

booleano readConfigFile(char *, CntParameters *);
void        printCntParameters(CntParameters *);
double   nucleationFreeEnergy_3rdLTE(double, double, double);
double   formationFreeEnergy(int, double, double, double);
double   diffusionCoefficient_Sc(double , double );
double   W_2_AlSc(double, double, double);
double   W_2_AlSc_V1(double);
double   isotropicFreeEnergy(double);
double   isotropicFreeEnergy_v1(double, double);
double   precipitateConcentration_timeEvol(double, double);
double   precipitateConcentration(int, double, double, double);
double   solubilityLimit_Sc_Al(double, double);
double   solubilityLimit_Sc_Al_V1(double, double);
```

```
double   qx_3rdLTE(double, double, double, double);
double   zeldovitchFactor(double, double, double, double);
double   condensationRate(double, double, double, double, double, double);
double   steadyStateNucleationRate_V1(double, double, int, double, double);
double   steadyStateNucleationRate(int, double, double, int, double, double);
```

x. cnt_main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "cnt_includes.h"

// Date: 04 October 2012

/* ***************************************************************************
 * Compute the nucleation free energy graph: deltaG_nuc(X0_Sc, T) in meV.
 *
 * Parameters:
 *          cntP        Pointer to the CNT parameters structure.
 *          cntR        Pointer to the CNT results structure.
 *          fp          Pointer to the open file where graph will be written.
 *
 *          Result: ---------
 */
void graphNucleationFreeEnergy (CntParameters *cntP, CntResults *cntR, FILE *fp) {

          int  nT, nX;

          for(nT=0;nT<cntP->num_T;++nT) {

                    for(nX=0;nX<cntP->num_X0_Sc;++nX) {

                              *(cntR->deltaG_nuc+nT*cntP->num_X0_Sc+nX) =
                              nucleationFreeEnergy_3rdLTE(*(cntP->T+nT), cntP->kB, *(cntP->X0_Sc+nX));
                    }
          }

          fprintf(fp,"+-------------------------------------------------------------------+\n");
          fprintf(fp,"|   Nucleation free energy graph: deltaG_nuc(X0_Sc, T)                |\n");
          fprintf(fp,"+-------------------------------------------------------------------+\n");
          fprintf(fp,"T [K]\tX0_Sc\tdeltaG_nuc [meV/at.]\n");
          for(nT=0;nT<cntP->num_T;++nT) {
                    for(nX=0;nX<cntP->num_X0_Sc;++nX) {
                              fprintf(fp,"%f\t%f\t%f\n", *(cntP->T+nT), *(cntP->X0_Sc+nX),
                                        *(cntR->deltaG_nuc+nT*cntP->num_X0_Sc+nX)*1000.0);
                    }
          }
}
```

```
/* ****************************************************************************
 * Compute the Sc impurity diffusion coefficient graph: D_Sc(T).
 *
 * Parameters:
 *          cntP        Pointer to the CNT parameters structure.
 *          cntR        Pointer to the CNT results structure.
 *          fp          Pointer to the open file where graph will be written.
 *
 *  Result: ---------
 */
void graphDiffusionCoefficient (CntParameters *cntP, CntResults *cntR, FILE *fp) {

        int         nT;

        fprintf(fp,"+-----------------------------------------------------------------------+\n");
        fprintf(fp,"|                    Sc impurity diffusion coefficient graph: D_Scn(1/T)                    |\n");
        fprintf(fp,"+-----------------------------------------------------------------------+\n");
        fprintf(fp,"1/T [K^-1]\t\t1/T [K^-1]\t\tD_Sc [m^2/s]\n");
        for(nT=cntP->num_T-1;nT>=0;--nT) {
                *(cntR->Dsc+nT) = diffusionCoefficient_Sc(*(cntP->T+nT), cntP->kB);
                fprintf(fp,"1/%.3f\t%f\t%e\n",*(cntP->T+nT), 1/(*(cntP->T+nT)),*(cntR->Dsc+nT));
        }
}

/* ****************************************************************************
 * Compute the isotropic free energy graph: a^2.sigma(T).
 *
 * Parameters:
 *          cntP        Pointer to the CNT parameters structure.
 *          cntR        Pointer to the CNT results structure.
 *          fp          Pointer to the open file where graph will be written.
 *
 *  Result: ---------
 */
void graphIsotropicFreeEnergy (CntParameters *cntP, CntResults *cntR, FILE *fp) {

        int         nT;

        fprintf(fp,"+-----------------------------------------------------------------------+\n");
        fprintf(fp,"|                    Isotropic free energy graph: a^2.sigma(T)                    |\n");
        fprintf(fp,"+-----------------------------------------------------------------------+\n");
        fprintf(fp,"T [K]\t\t\ta^2.sigma(T) [eV]\n");
        for(nT=0; nT<cntP->num_T;++nT) {
                *(cntR->a2sigma+nT) = isotropicFreeEnergy_v1(*(cntP->T+nT), cntP->kB);
                fprintf(fp,"%f\t%e\n",*(cntP->T+nT), *(cntR->a2sigma+nT));
        }
}

/* ****************************************************************************
 * Compute the steady state nucleation rate graph: Jst(X0_Sc,T).
 *
 * Parameters:
 *          cntP        Pointer to the CNT parameters structure.
 *          cntR        Pointer to the CNT results structure.
 *          fp          Pointer to the open file where graph will be written.
 *
 *  Result: ---------
 */
```

```
void graphSteadyStateNucleationRate (CntParameters *cntP, CntResults *cntR, FILE *fp) {

	int	nT, nX;

	for(nT=0;nT<cntP->num_T;++nT) {

		for(nX=0;nX<cntP->num_X0_Sc;++nX) {

			*(cntR->Jst+nT*cntP->num_X0_Sc+nX) =
				steadyStateNucleationRate_V1(*(cntP->X0_Sc+nX), cntP->alfa, cntP->Ns,
				*(cntP->T+nT), cntP->kB);
		}
	}

	fprintf(fp,"+---------------------------------------------------------------------+\n");
	fprintf(fp,"|	Steady state nucleation rate (per m^3) graph : Jst(X0_Sc, T)		|\n");
	fprintf(fp,"+---------------------------------------------------------------------+\n");
	fprintf(fp,"T [K]\tX0_Sc\tJst [nuclei/(m^3.s)]\n");
	for(nT=0;nT<cntP->num_T;++nT) {
		for(nX=0;nX<cntP->num_X0_Sc;++nX) {
			fprintf(fp,"%f\t%f\t%e\n", *(cntP->T+nT), *(cntP->X0_Sc+nX),
				*(cntR->Jst+nT*cntP->num_X0_Sc+nX));
		}
	}
}

/* ************************************************************************************
 * Compute the precipitate concentration graph for several X0_Sc: C_nSc(X0_Sc,nSc).
 *
 * Parameters:
 *	cntP	Pointer to the CNT parameters structure.
 *	cntR	Pointer to the CNT results structure.
 *	fp	Pointer to the open file where graph will be written.
 *
 *	Result: ---------
 */
void graphPrecipitateConcentration_Xc (CntParameters *cntP, CntResults *cntR, FILE *fp) {

	int	nS, nX;

	for(nX=0;nX<cntP->num_X0_Sc;++nX) {
		for(nS=0;nS<cntP->nSc_critical;++nS) {

			*(cntR->C_nSc_X0+nX*cntP->nSc_critical+nS) =
				precipitateConcentration(*(cntP->nSc+nS), *(cntP->X0_Sc+nX),
				cntP->T_concentration, cntP->kB);
		}
	}

	fprintf(fp,"+---------------------------------------------------------------------+\n");
	fprintf(fp,"| Precipitate concentration graph for several X0_Sc: C_nSc(X0_Sc,nSc) |\n");
	fprintf(fp,"| Temperatue is fixed at %f K
	 |\n",
		cntP->T_concentration);
	fprintf(fp,"+---------------------------------------------------------------------+\n");
	fprintf(fp,"X0_Sc\t\tnSc\t\tC_nSc\n");
	for(nX=0;nX<cntP->num_X0_Sc;++nX) {
		for(nS=0;nS<cntP->nSc_critical;++nS) {
```

```
                                fprintf(fp,"%f\t%d\t%e\n", *(cntP->X0_Sc+nX), *(cntP->nSc+nS),
                                        *(cntR->C_nSc_X0+nX*cntP->nSc_critical+nS));
                        }
                }
}

/* ************************************************************************************
 * Compute the precipitate concentration graph for several times: C_nSc(time,nSc).
 *
 * Parameters:
 *          cntP        Pointer to the CNT parameters structure.
 *          cntR        Pointer to the CNT results structure.
 *          fp          Pointer to the open file where graph will be written.
 *
 *  Result: ---------
 */
//void graphPrecipitateConcentration_time (CntParameters *cntP, CntResults *cntR, FILE *fp) {
//
//          int           nS, nt;
//
//          for(nt=0;nt<cntP->num_time;++nt) {
//                  for(nS=0;nS<cntP->nSc_critical;++nS) {
//
//                          *(cntR->C_nSc_t+nt*cntP->nSc_critical+nS) =
//                                  precipitateConcentration(....);
//                  }
//          }
//
//          fprintf(fp,"+--------------------------------------------------------------------+\n");
//          fprintf(fp,"| Precipitate concentration graph for several times: C_nSc(time,nSc)  |\n");
//          fprintf(fp,"| Temperatue is fixed at %f K
           |\n",
//                  cntP->T_concentration);
//          fprintf(fp,"| Sc nominal concentration is fixed at %f                                                    |\n",
//                  cntP->X0_Sc_concentration);
//          fprintf(fp,"+--------------------------------------------------------------------+\n");
//          fprintf(fp,"Time [s]\t\tnSc\t\tC_nSc\n");
//          for(nt=0;nt<cntP->num_time;++nt) {
//                  for(nS=0;nS<cntP->nSc_critical;++nS) {
//                          fprintf(fp,"%f\t%d\t%e\n", *(cntP->time+nt), *(cntP->nSc+nS),
//                                  *(cntR->C_nSc_t+nt*cntP->nSc_critical+nS));
//                  }
//          }
//}

// ************************************************************************************
int main(int argc, char *argv[]) {
          KeyNumberValues           kv;
          CntParameters             cntParam;
          CntResults                cntRes;
          int                        n;
          FILE                      *fp;

          printf("************************************************************************\n");
          printf("*   Classic Nucleation Theory (CNT) applied to Al3Sc Precipitation   *\n");
          printf("************************************************************************\n");
          if(argc < 3) {
                  printf("Run program with the following arguments:\n");
```

```
		printf("  argument #1 -> configuration file\n");
		printf("  argument #2 -> output file with graphs data\n");
		printf("*********************************************************************\n");
		return(-1);
}
else {
		if (readConfigFile(argv[1], &cntParam) == FALSE)
			return(-1);

		// Assign values to 'nSc' (number of Sc atoms in the precipitates = [1..nSc_critical])
		if((cntParam.nSc = (int *)malloc(sizeof(int)*cntParam.nSc_critical)) == NULL){
			printf("Error when allocating memory to 'cntParam->nSc' in main()!\n");
			return(-2);
		}
		for (n=1;n<=cntParam.nSc_critical;++n){
			cntParam.nSc[n-1] = n;
		}
}

printf("*********************************************************************\n");
printCntParameters(&cntParam);
printf("*********************************************************************\n");

if((fp = fopen(argv[2],"w"))==NULL){
		printf("Error when trying to open output file %s\n", argv[2]);
		return (-3);
};

// ---------------------------------------------------------------------
// Compute the nucleation free energy graph --> deltaG_nuc(X0_Sc, T)
// allocate memory for the matrix that will store deltaG_nuc(X0_Sc, T) values
if ((cntRes.deltaG_nuc=(double *)malloc(sizeof(double)*cntParam.num_T*cntParam.num_X0_Sc))
		 == NULL) {
		printf("Error when allocating memory to 'cntRes.deltaG_nuc' in main()!\n");
		return(-4);
}

graphNucleationFreeEnergy(&cntParam, &cntRes, fp);

// ---------------------------------------------------------------------
// allocate memory for the array that will store D_Sc(T) values
if ((cntRes.Dsc=(double *)malloc(sizeof(double)*cntParam.num_T)) == NULL) {
		printf("Error when allocating memory to 'cntRes.Dsc' in main()!\n");
		return(-5);
}

graphDiffusionCoefficient(&cntParam, &cntRes, fp);

// ---------------------------------------------------------------------
// allocate memory for the array that will store a^2.sigma(T) values
if ((cntRes.a2sigma =(double *)malloc(sizeof(double)*cntParam.num_T)) == NULL) {
		printf("Error when allocating memory to 'cntRes.a2sigma' in main()!\n");
		return(-6);
}

graphIsotropicFreeEnergy(&cntParam, &cntRes, fp);
```

```
	// ---------------------------------------------------------------------
	// allocate memory for the matrix that will store Jst(X0_Sc, T) values
	if ((cntRes.Jst=(double *)malloc(sizeof(double)*cntParam.num_T*cntParam.num_X0_Sc)) == NULL) {
		printf("Error when allocating memory to 'cntRes.Jst' in main()!\n");
		return(-7);
	}

	graphSteadyStateNucleationRate(&cntParam, &cntRes, fp);

	// ---------------------------------------------------------------------
	// allocate memory for the matrix that will store C_nSc_X0 (X0_Sc, nSc) values
	if ((cntRes.C_nSc_X0 =(double *)malloc(sizeof(double)*cntParam.nSc_critical*cntParam.num_X0_Sc))
		= NULL) {
		printf("Error when allocating memory to 'cntRes.C_nSc_X0' in main()!\n");
		return(-8);
	}
	// allocate memory for the matrix that will store C_nSc_t (time, nSc) values
	if ((cntRes.C_nSc_t =(double *)malloc(sizeof(double)*cntParam.nSc_critical*cntParam.num_time))
		== NULL) {
		printf("Error when allocating memory to 'cntRes.C_nSc_t' in main()!\n");
		return(-9);
	}

	graphPrecipitateConcentration_Xc(&cntParam, &cntRes, fp);

	// Free allocated memory and close open file ------------------------------

	fclose(fp);

	if(cntRes.C_nSc_t != NULL)
		free(cntRes.C_nSc_t);
	if(cntRes.C_nSc_X0 != NULL)
		free(cntRes.C_nSc_X0);
	if(cntRes.Jst != NULL)
		free(cntRes.Jst);
	if(cntRes.a2sigma != NULL)
		free(cntRes.a2sigma);
	if(cntRes.Dsc != NULL)
		free(cntRes.Dsc);
	if(cntRes.deltaG_nuc != NULL)
		free(cntRes.deltaG_nuc);

	if(cntParam.T != NULL)
		free(cntParam.T);
	if(cntParam.X0_Sc != NULL)
		free(cntParam.X0_Sc);
	if(cntParam.time != NULL)
		free(cntParam.time);
	if(cntParam.nSc != NULL)
		free(cntParam.nSc);

	return(0);
}
```

xi. config.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "main_includes.h"

// Date: 09 October 2012

// Read a text line from an ASCII file, until NEWLINE or EOF is found.
//
//		Parameters:
//			fp		File pointer to open file.
//			buffer	Buffer where characters will be written.
//
//  Return: TRUE (didn't find EOF), FALSE (found EOF).
//
booleano readLine (FILE *fp, char *buffer) {
	int   n=0;
	char  ch=0;
	if(fp != NULL) {
		while ( (!feof(fp)) && (ch!=NEWLINE) && (n<MAX_LINE_SIZE) ) {
			fscanf(fp,"%c",&ch);
			buffer[n]=ch;
			++n;
		}
		--n;
		buffer[n]='\0';

		if((n>0)&&(buffer[n-1]==ENTER)) // to remove the ENTER at end of a line
			buffer[n-1]=='\0';

		if(n==MAX_LINE_SIZE) { // line size exceeds the limit
			while ( (!feof(fp)) && (ch!=NEWLINE) ) { // discard the rest of line
				fscanf(fp,"%c",&ch);
			}
		}
	}

	if (feof(fp)) return(FALSE);
	else return(TRUE);
}

// Split a text line, read from configuration file, in its parameter (key,value) pair.
//
// Parameters:
//		line		Pointer to the string that contains the line.
//		kv		Pointer to the structure where the the (key,value) pair will be saved.
//
// Return: TRUE (if pair was found), FALSE (if pair wasn't found).
//
booleano splitLine (char *line, KeyValue *kv) {
	char key[MAX_LINE_SIZE], value[MAX_LINE_SIZE];
	int  n=0, i=0;

	while( (line[n]==' ') || (line[n]=='\t') ) {
		n++; // skip white chars (' ', '\t')
```

```
		if (n==strlen(line))
			return (FALSE);
	}

	while( (line[n]!=' ') && (line[n]!='=') ) { // extract parameter name (key)
		kv->key[i] = line[n];
		++n;
		++i;
		if (n==strlen(line))
			return (FALSE);
	}
	kv->key[i]='\0';

	if (line[n] != '=')
		while( line[n]!='=') {
		n++; // find key-value separator ("=")
		if (n==strlen(line))
			return (FALSE);
	}
	++n;

	while( (line[n]==' ') || (line[n]=='\t') ) {
		n++; // skip white chars (' ', '\t')
		if (n==strlen(line))
			return (FALSE);
	}

	i=0;
	while( (line[n]!=' ') && (line[n]!='#') && (n<strlen(line)) ) { // extract parameter value
		kv->value[i] = line[n];
		++n;
		++i;
	}
	kv->value[i]='\0';
	return(TRUE);
}

// Identify the parameter read from configuration file, stored in the 'kv'
// (key,value) pair, and assign its value to the adequate structure field.
//
//	Parameters:
//		kv	Pointer to the (key,value) pair that was read from configuration file.
//		alScP	Pointer to structure where AlSc parameters will be saved.
//		latP	Pointer to structure where lattice parameters will be saved.
//		simP	Pointer to structure where simulation parameters will be saved.
//		energyP	Pointer to structure where energy parameters will be saved.
//		clustP	Pointer to structure where clustering parameters will be saved.
//
//	Return: -----
//
int identifyParameter(KeyValue *kv, AlScParameters *alScP, LatticeParameters *latP,
	SimulationParameters *simP, EnergyParameters *energyP, ClusteringParameters *clustP) {

	if(strcmp(kv->key,"al_Radius")==0){
		alScP->al_Radius = atof(kv->value);
	}
```

```
if(strcmp(kv->key,"sc_Radius")==0){
        alScP->sc_Radius = atof(kv->value);
}
if(strcmp(kv->key,"lc")==0){
        alScP->lc = atof(kv->value);
        latP->lc  = alScP->lc;
}

if(strcmp(kv->key,"nbz")==0){
        latP->nbz = atoi(kv->value);
}
if(strcmp(kv->key,"nby")==0){
        latP->nby = atoi(kv->value);
}
if(strcmp(kv->key,"nbx")==0){
        latP->nbx = atoi(kv->value);
}
if(strcmp(kv->key,"num_Al_Sites")==0){
        latP->num_Al_Sites = atoi(kv->value);
}
if(strcmp(kv->key,"num_Sc_Sites")==0){
        latP->num_Sc_Sites = atoi(kv->value);
}
if(strcmp(kv->key,"num_V_Sites")==0){
        latP->num_V_Sites = atoi(kv->value);
}
if(strcmp(kv->key,"ScC")==0){
        latP->ScC = atof(kv->value);
}

if(strcmp(kv->key,"mcs")==0){
        simP->mcs = atoll(kv->value);
}
if(strcmp(kv->key,"T")==0){
        simP->T = atof(kv->value);
}
if(strcmp(kv->key,"snap")==0){
        simP->snap = atoll(kv->value);
}
if(strcmp(kv->key,"kB")==0){
        simP->kB = atof(kv->value);
}
if(strcmp(kv->key,"wrXYZ")==0){
        if (strcmp(kv->value,"TRUE")==0){
                simP->wrXYZ = TRUE;
        }
        else if (strcmp(kv->value,"FALSE")==0) {
                simP->wrXYZ = FALSE;
        }
}
if(strcmp(kv->key,"wrVTK")==0){
        if (strcmp(kv->value,"TRUE")==0){
                simP->wrVTK = TRUE;
        }
        else if (strcmp(kv->value,"FALSE")==0) {
                simP->wrVTK = FALSE;
        }
}
```

```
if(strcmp(kv->key,"wrPDB")==0){
        if (strcmp(kv->value,"TRUE")==0){
                simP->wrPDB = TRUE;
        }
        else if (strcmp(kv->value,"FALSE")==0) {
                simP->wrPDB = FALSE;
        }
}
if(strcmp(kv->key,"wrReport")==0){
        if (strcmp(kv->value,"TRUE")==0){
                simP->wrReport = TRUE;
        }
        else if (strcmp(kv->value,"FALSE")==0) {
                simP->wrReport = FALSE;
        }
}
if(strcmp(kv->key,"coordFile")==0){
        strcpy(simP->coordFile,kv->value);
}
if(strcmp(kv->key,"selectedType")==0){
        if(strcmp(kv->value,"AL")==0)           // Al
                simP->selectedType = AL;
        else if(strcmp(kv->value,"SC")==0)     // Sc
                simP->selectedType = SC;
        else if(strcmp(kv->value,"VACANCY")==0) // Vacancy
                simP->selectedType = VACANCY;
        else if(strcmp(kv->value,"ALSC")==0)  // Al and Sc
                simP->selectedType = ALSC;
        else if(strcmp(kv->value,"ALV")==0)   // Al and Vacancy
                simP->selectedType = ALV;
        else if(strcmp(kv->value,"SCV")==0)   // Sc and Vacancy
                simP->selectedType = SCV;
        else if(strcmp(kv->value,"ALL")==0)   // All
                simP->selectedType = ALL;
}
if(strcmp(kv->key,"typeVTKdata")==0){
        if(strcmp(kv->value,"OnlyAtoms")==0)   // Only atoms
                simP->typeVTKdata = OnlyAtoms;
        else if(strcmp(kv->value,"AtomsCubes")==0) // Atoms and cubes
                simP->typeVTKdata = AtomsCubes;
}

// Complete the 1st and 2nd nearest neighbor pair effective energies:
//   energyP->E_AlSc_1 = energyP->E_AlSc_1 + 21.0e-6*simP->T;  // [Clouet 2004]
//   energyP->E_AlSc_2 = energyP->E_AlSc_2 - 33.4e-6*simP->T;  // [Clouet 2004]

if(strcmp(kv->key,"E_AlAl_1")==0){
        energyP->E_AlAl_1 = atof(kv->value);
}
if(strcmp(kv->key,"E_ScSc_1")==0){
        energyP->E_ScSc_1 = atof(kv->value);
}
if(strcmp(kv->key,"E_AlSc_1")==0){
        energyP->E_AlSc_1 = atof(kv->value)+21.0e-6*simP->T;  // [Clouet 2004]
}
if(strcmp(kv->key,"E_AlSc_2")==0){
        energyP->E_AlSc_2 = atof(kv->value)-33.4e-6*simP->T;  // [Clouet 2004]
```

```
}
if(strcmp(kv->key,"E_AlAl_2")==0){
	energyP->E_AlAl_2 = atof(kv->value);  // ZERO in [Clouet 2004]
}
if(strcmp(kv->key,"E_ScSc_2")==0){
	energyP->E_ScSc_2 = atof(kv->value);  // ZERO in [Clouet 2004]
}
if(strcmp(kv->key,"E_AlV_1")==0){
	energyP->E_AlV_1 = atof(kv->value);
}
if(strcmp(kv->key,"E_ScV_1")==0){
	energyP->E_ScV_1 = atof(kv->value);
}
if(strcmp(kv->key,"e_spAl")==0){
	energyP->e_spAl = atof(kv->value);
}
if(strcmp(kv->key,"e_spSc")==0){
	energyP->e_spSc = atof(kv->value);
}
if(strcmp(kv->key,"vAl")==0){
	energyP->vAl = atof(kv->value);
}
if(strcmp(kv->key,"vSc")==0){
	energyP->vSc = atof(kv->value);
}

if(strcmp(kv->key,"eps")==0){
	clustP->eps = atof(kv->value);
}
if(strcmp(kv->key,"minPts")==0){
	clustP->minPts = atoi(kv->value);
}
if(strcmp(kv->key,"storeDistances")==0){
	if (strcmp(kv->value,"TRUE")==0){
		clustP->storeDistances = TRUE;
	}
	else if (strcmp(kv->value,"FALSE")==0) {
		clustP->storeDistances = FALSE;
	}
}
if(strcmp(kv->key,"minClusterSize")==0){
	clustP->minSize = atoi(kv->value);
	if(clustP->minSize<clustP->minPts) // it is not acceptable a value smaller than 'minPts'
		clustP->minSize = clustP->minPts;
}

if(strcmp(kv->key,"stablePrecipitatesAnalysis")==0){
	if (strcmp(kv->value,"TRUE")==0){
		clustP->stablePrecipitates = TRUE;
	}
	else if (strcmp(kv->value,"FALSE")==0) {
		clustP->stablePrecipitates = FALSE;
	}
}
if(strcmp(kv->key,"smallClustersAnalysis")==0){
	if (strcmp(kv->value,"TRUE")==0){
		clustP->smallClusters = TRUE;
	}
```

```
                else if (strcmp(kv->value,"FALSE")==0) {
                        clustP->smallClusters = FALSE;
                }
        }
}

// *********************************************************************
//  Read configuration file.
//
//  Parameters:
//                  fName    Configuration file name.
//                  simP     Pointer to the simulation parameters.
//                  energyP  Pointer to the energy parameters.
//                  alScP    Pointer to the Al and Sc parameters.
//                  latP     Pointer to the lattice parameters.
//                  clustP   Pointer to the clustering parameters.
//
//  Return: TRUE (if reading file succeeds), FALSE (if reading file fails).
//
booleano readConfigFile (char *fName, SimulationParameters *simP, EnergyParameters *energyP,
        AlScParameters *alScP, LatticeParameters *latP, ClusteringParameters *clustP) {

        char line[MAX_LINE_SIZE];
        FILE *fp;
        booleano continueReading;
        KeyValue kv;

        if((fp = fopen(fName,"r"))==NULL){
                printf("Error when trying to open configuration file %s\n", fName);
                return FALSE;
        };

        do {
                continueReading = readLine(fp, line);
                if(strlen(line)>1) {
                        if(splitLine(line, &kv) == FALSE)
                                printf("Error on parsing configuration line!\n");
                        else {
                                identifyParameter(&kv, alScP, latP,     simP, energyP, clustP);
                        }
                }
        } while (continueReading == TRUE);

        fclose(fp);
        return TRUE;
}

// *********************************************************************
// Define the parameters for the type of atoms used in simulation.
//
void defAlScParameters (AlScParameters *alScParam){
        alScParam->al_Radius        = 1.18;             // Aluminum atom radius (Aº)
        alScParam->sc_Radius        = 1.84;             // Scandium atom radius (Aº)
        alScParam->lc               = 4.05;             // Aluminum lattice constant (Aº)
}
```

```
// ********************************************************************
// Print the parameters for the type of atoms used in simulation.
//
void printAlScParameters (AlScParameters *alScParam){
        printf(" * Aluminum atom radius          = %f\n",alScParam->al_Radius);
        printf(" * Scandium atom radius          = %f\n",alScParam->sc_Radius);
        printf(" * Aluminum lattice constant = %f\n",alScParam->lc);
}

// ********************************************************************
// Define the lattice parameters.
//
void defLatticeParameters (LatticeParameters *latP) {
        latP->nbx=3;
        latP->nby=3;
        latP->nbz=3;
        latP->num_Al_Sites=0;
        latP->num_Sc_Sites=0;
        latP->num_V_Sites=0;
        latP->lc=4.05;
        latP->ScC=0.5;
}

// ********************************************************************
// Print the lattice parameters.
//
void printLatticeParameters (LatticeParameters *latP) {
        printf(" * Number of unit cells in the z direction  = %d\n",latP->nbx);
        printf(" * Number of unit cells in the y direction  = %d\n",latP->nby);
        printf(" * Number of unit cells in the x direction  = %d\n",latP->nbz);
        printf(" * Number of Aluminum atoms/sites       = %d\n",latP->num_Al_Sites);
        printf(" * Number of Scandium atoms/sites       = %d\n",latP->num_Sc_Sites);
        printf(" * Number of Vacancy sites              = %d\n",latP->num_V_Sites);
        printf(" * Aluminum lattice constant            = %f\n",latP->lc);
        printf(" * Scandium percentage                  = %f\n",latP->ScC);
}

// ********************************************************************
// Define the parameters for the simulation.
//
void defSimulParameters (SimulationParameters *simParam){
        simParam->mcs           = 100ULL;       // Simulation Monte Carlo steps
        simParam->T             = 773.0;        // Simulation temperature (Kelvin)
        simParam->snap          = 4ULL;         // Number of snapshots to save during simulation
        simParam->kB            = 8.6173324e-5; // Boltzmann Constant (ElectronVolt/Kelvin)
        simParam->wrXYZ         = FALSE;        // Don't Write XYZ files
        simParam->wrVTK         = TRUE;         // Write VTK files
        simParam->wrPDB         = FALSE;        // Don't Write PDB files
        simParam->wrReport      = TRUE;         // Write simulation report file
        strcpy(simParam->coordFile, "MC_AlSc_precipitation"); // Base name of the output file(s)
        simParam->selectedType  = SC;           // Selected type of atoms to write into (VTK) file
        simParam->typeVTKdata   = OnlyAtoms;    // Type of data to write in VTK file: only atoms
}
```

```
//          ****************************************************************
//          Print the parameters for the simulation.
//
void printSimulParameters(SimulationParameters *simParam){
        printf(" * Simulation Monte Carlo steps                    = %lld\n",simParam->mcs);
        printf(" * Simulation temperature (Kelvin)                 = %f\n",simParam->T);
        printf(" * Number of snapshots to save during simulation   = %lld\n",simParam->snap);
        printf(" * Boltzmann Constant (ElectronVolt/Kelvin)        = %.7e\n",simParam->kB);
        printf(" * Write XYZ files  (Y/N)                          = %d\n",simParam->wrXYZ);
        printf(" * Write VTK files  (Y/N)                          = %d\n",simParam->wrVTK);
        printf(" * Write PDB files  (Y/N)                          = %d\n",simParam->wrPDB);
        printf(" * Write simulation report file  (Y/N)              = %d\n",simParam->wrReport);
        printf(" * Base name of the output file(s)                 = %s\n",simParam->coordFile);
        printf(" * Type(s) of atoms selected to write into (VTK) file: ");
        if (simParam->selectedType==AL)                 printf("Al\n");
        else if (simParam->selectedType==SC)            printf("Sc\n");
        else if (simParam->selectedType==VACANCY) printf("Vacancy\n");
        else if (simParam->selectedType==ALSC)      printf("Al and Sc\n");
        else if (simParam->selectedType==ALV)       printf("Al and Vacancy\n");
        else if (simParam->selectedType==SCV)       printf("Sc and Vacancy\n");
        else if (simParam->selectedType==ALL)       printf("All\n");
        printf(" * Type of data to write in VTK file: ");
        if (simParam->typeVTKdata==OnlyAtoms)     printf("Only atoms\n");
        else if (simParam->typeVTKdata==AtomsCubes) printf("Atoms and cubes\n");

}

// ********************************************************************
// Define the energy and kinetic parameters.
//
void defEnergyParameters (EnergyParameters *energyParam) {
        energyParam->E_AlAl_1 = -0.560;     // 1st nearest neighbor pair effective energies
        energyParam->E_ScSc_1 = -0.650;     // 1st nearest neighbor pair effective energies
        energyParam->E_AlSc_1 = -0.759;     // 1st nearest neighbor pair effective energies
        energyParam->E_AlSc_2 =  0.113;     // 2nd nearest neighbor pair effective energies
        energyParam->E_AlAl_2 =  0.0;       // 2nd nearest neighbor pair effective energies
        energyParam->E_ScSc_2 =  0.0;       // 2nd nearest neighbor pair effective energies
        energyParam->E_AlV_1  = -0.222;     // 1st nearest neighbor pair effective energies
        energyParam->E_ScV_1  = -0.757;     // 1st nearest neighbor pair effective energies
        energyParam->e_spAl    = -8.219;    // Aluminum saddle point energy
        energyParam->e_spSc    = -9.434;    // Scandium saddle point energy
        energyParam->vAl       = 1.36e14;   // Aluminum attempt frequency
        energyParam->vSc       = 4e15;      // Scandium attempt frequency
}

// ********************************************************************
// Print the energy and kinetic parameters.
//
void printEnergyParameters (EnergyParameters *energyParam) {
        printf(" * 1st nearest neighbor pair effective energies = %f\n",energyParam->E_AlAl_1);
        printf(" * 1st nearest neighbor pair effective energies = %f\n",energyParam->E_ScSc_1);
        printf(" * 1st nearest neighbor pair effective energies = %f\n",energyParam->E_AlSc_1);
        printf(" * 2nd nearest neighbor pair effective energies = %f\n",energyParam->E_AlSc_2);
        printf(" * 2nd nearest neighbor pair effective energies = %f\n",energyParam->E_AlAl_2);
        printf(" * 2nd nearest neighbor pair effective energies = %f\n",energyParam->E_ScSc_2);
        printf(" * 1st nearest neighbor pair effective energies = %f\n",energyParam->E_AlV_1);
        printf(" * 1st nearest neighbor pair effective energies = %f\n",energyParam->E_ScV_1);
        printf(" * Aluminum saddle point energy            = %f\n",energyParam->e_spAl);
```

```
	printf(" * Scandium saddle point energy		= %f\n",energyParam->e_spSc);
	printf(" * Aluminum attempt frequency		= %.3e\n",energyParam->vAl);
	printf(" * Scandium attempt frequency		= %.3e\n",energyParam->vSc);
}

// *********************************************************************
// Define the parameters for the clustering analysis.
//
void defClusterParameters (ClusteringParameters *clustP){
	clustP->eps		= 8.20;	// The radius used to define the neighborhood of
					// each atom (DBSCAN algorithm)
	clustP->minPts	= 3;	// Minimum number of neighbors that makes an atom
						// to be a core atom of a cluster (DBSCAN algorithm)
	clustP->storeDistances = FALSE;	// Store (TRUE) or do NOT store (FALSE) the distance
					// between each pair of atoms in advance (DBSCAN algorithm)
	clustP->minSize = 12;			// Minimum number of atoms to consider a cluster as valid
	clustP->stablePrecipitates = TRUE;	// Execute stable precipitates analysis
	clustP->smallClusters = FALSE;		// Execute small clusters analysis
}

// *********************************************************************
// Print the parameters for the clustering analysis.
//
void printClusterParameters (ClusteringParameters *clustP){
	printf(" * Value of EPS in DBSCAN algorithm   = %f\n",clustP->eps);
	printf(" * Value of MINPTS in DBSCAN algorithm = %d\n",clustP->minPts);
	if (clustP->storeDistances==TRUE)
		printf(" * Store distance between pairs of atoms in DBSCAN algorithm = YES\n");
	else
		printf(" * Store distance between pairs of atoms in DBSCAN algorithm = NO\n");
	printf(" * Minimum number of atoms in a valid cluster = %d\n",clustP->minSize);
	if (clustP->stablePrecipitates == TRUE)
		printf(" * Execute stable precipitates analysis\n");
	if(clustP->smallClusters == TRUE)
		printf(" * Execute small clusters analysis\n");
}
```

xii. dbscan1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "main_includes.h"
#include "ArrayList.h"

// Date: 11 September 2012

// Compute the euclidean distance between the points 'pi' and 'pf', in a 3D space,
// considering Periodic Boundary Conditions (PBC).
//
//  Parameters:
//		pi	One point.
//		pf	The other point.
//
//  Return: The euclidean distance between the 2 points.
//
```

```
float euclideanDistancePBC (point3Dclassified pi, point3Dclassified pf, float lx, float ly, float lz) {

	float distAux, dist = 0.0;
	float lx_half, ly_half, lz_half, minus_lx_half, minus_ly_half, minus_lz_half;
	float dx, dy, dz;

	lx_half = lx/2.0;
	ly_half = ly/2.0;
	lz_half = lz/2.0;
	minus_lx_half = -lx_half;
	minus_ly_half = -ly_half;
	minus_lz_half = -lz_half;

	dx = pf.x - pi.x;
	if(dx > lx_half)
		dx = pi.x + lx - pf.x;
	else if(dx < minus_lx_half)
		dx = pf.x + lx - pi.x;
	else
		dx = pf.x - pi.x;

	dy = pf.y - pi.y;
	if(dy > ly_half)
		dy = pi.y + ly - pf.y;
	else if(dy < minus_ly_half)
		dy = pf.y + ly - pi.y;
	else
		dy = pf.y - pi.y;

	dz = pf.z - pi.z;
	if(dz > lz_half)
		dz = pi.z + lz - pf.z;
	else if(dz < minus_lz_half)
		dz = pf.z + lz - pi.z;
	else
		dz = pf.z - pi.z;

	dist = dz*dz;
	dist += dy*dy;
	dist += dx*dx;
	dist = sqrt(dist);

	return dist;
}

// ********************************************************************************
// Compute the neighborhood of all atoms on a radius of value 'eps'.
//
//  Parameters:
//		atoms	 Pointer to the array where the atoms (coordinates, type, classID)
//			are stored.
//		dist	Pointer to the array with euclidean distance between all pairs of atoms.
//		nAtoms	The number atoms.
//		eps	The radius used to define the neighborhood of each atom.
//		N	Pointer to the array of 'Arraylist' where atoms' neighboorhood
//			will be stored (ONLY atom's position NOT atom's data).
//
//  Return: ----
```

```
//
void getNeighbors_1 (point3Dclassified *atoms, float* dist, int nAtoms, float eps, ArrayList* N) {
	int	i, j;

	for(i=0;i<nAtoms;++i) {
		for(j=0;j<nAtoms;++j) {
			if (j != i) {
				if(*(dist+i*nAtoms+j) <= eps) {
					add(N+i, (Element){j});
					}
			}
		}
	}
}

void printNeighbors (ArrayList* N, int nAtoms) {
	int	i, j, value, sizeAL;
	Element *e;

	for(i=0;i<nAtoms;++i) {
		sizeAL = size(N+i);
		printf("Checked neighboors of atom %d: ",i);
		for(j=0;j<sizeAL;++j) {
			e = get(N+i,j);
			value = (e!=NULL) ? e->data : -1;
			printf("%d ", value);
		}
		printf("\n");
	}
}

void printClusters (point3Dclassified *atoms, int nAtoms) {
	int	i, j, nClusters=0, cid=0, *clusterSize;

	for(i=0;i<nAtoms;++i) {
		if (atoms[i].classId > nClusters)
			nClusters = atoms[i].classId;
	}
	printf("\nNumber of clusters is %d\n",nClusters);

	if ((clusterSize = (int *)malloc(nClusters*sizeof(int))) == NULL) {
		printf("Error on memory allocation for 'clusterSize'\n");
		return;
	}

	for(i=0;i<nClusters;++i) {
		clusterSize[i] = 0;
		for(j=0;j<nAtoms;++j) {
			if (atoms[j].classId == i)
				++clusterSize[i];
		}
		printf("Size of cluster %d is %d\n",i,clusterSize[i]);

	}

	free(clusterSize);
}
```

```
void printDistances (float *dist, int nAtoms) {
	int i,j;
	printf("DISTANCE BETWEEN ATOMS:\n");
	for(i=0;i<nAtoms;++i) {
		printf("%d -> ",i);
		for(j=0;j<nAtoms;++j) {
			printf("%f ",*(dist+i*nAtoms+j));
		}
	printf("\n");
	}
}

// ********************************************************************************
// Expand a cluster.
//
//  Parameters:
//			atoms	Pointer to the array where the atoms (coordinates, type, classID) are stored.
//			nAtoms	The number of atoms.
//			visited	Pointer to the array that idicates if atoms were yet visited or not.
//			N	Pointer to the array of 'Arraylist' where atoms' neighborhood are stored.
//			pid	 Position of the current atom in the array of all atoms.
//			cid	 Current cluster ID.
//			eps	 The radius used to define the neighborhood of each atom.
//			minPts	Minimum number of neighbors that makes an atom to be a core atom of a cluster.
//
//  Return: TRUE (when a new cluster was created), FALSE (no clusters was created).
//
booleano ExpandCluster(point3Dclassified *atoms, int nAtoms, booleano *visited,
		ArrayList *N, int pid, int cid, float eps, int minPts) {

	int	 i, nid, nn, nnid, sizeN, sizeNN, sizeUnclustered;
	Element *e;

	sizeN  = size(N+pid);
	sizeUnclustered = sizeN;

	for(i=0;i<sizeN;++i) { // Count the unclustered neighbors of atom 'pid'
		e   = get(N+pid,i);
		nid = (e!=NULL) ? e->data : -1;
		if( (atoms[nid].classId != NOISE) && (atoms[nid].classId != UNCLASSIFIED) ) {
			--sizeUnclustered;
		}
	}

	// There are NOT enough unclustered neighbors to make atom 'pid' a core atom of a new cluster
	if (sizeUnclustered < minPts) {
		atoms[pid].classId = NOISE; // Mark atom 'pid' as NOISE
		return FALSE;
	}

	// There are enough unclustered neighbors to make atom 'pid' a core atom of a new cluster
	else {
		atoms[pid].classId = cid; // Add atom 'pid' to cluster 'cid'

		for(i=0;i<sizeN;++i) {
			e   = get(N+pid,i);
			nid = (e!=NULL) ? e->data : -1;
```

```
                        if (visited[nid] == FALSE) { // atom 'nid' is not yet visited
                                visited[nid] = TRUE;
                                sizeNN = size(N+nid);
                                if(sizeNN >= minPts) {
                                        for(nn=0;nn<sizeNN;++nn) {
                                                e       = get(N+nid,nn);
                                                nnid  = (e!=NULL) ? e->data : -1;
                                                add(N+pid, (Element){nnid});
                                                ++sizeN;
                                        }
                                }
                        }

                        // Atom 'nid' is not yet member of any cluster
                        if( (atoms[nid].classId == NOISE) || (atoms[nid].classId == UNCLASSIFIED) ) {
                                atoms[nid].classId = cid;
                        }
                }
        }
        return TRUE;
}

// ********************************************************************************
//  DBSCAN clustering algorithm (with the distance between all pairs of points
//  computed and stored before they are needed).
//
//  Parameters:
//              atoms     Pointer to the array where the atoms (coordinates, type, classID) are stored.
//              nAtoms    The number atoms.
//              eps       The radius used to define the neighborhood of each atom.
//              minPts    Minimum number of neighbors that makes an atom to be a core atom.
//              latP      Pointer to the lattice parameters.
//
//  Return: The number of identified clusters.
//
int DBSCAN_1 (point3Dclassified *atoms, int nAtoms, float eps, int minPts,
              LatticeParameters *latP) {
        float                   *dist;
        int                     i, j, cid, pid, sizeN;
        ArrayList               *N;
        point3Dclassified       p;
        float                   lx, ly, lz;
        long long unsigned int  sizeDist;
        booleano                *visited;

        // Create an array to store a flag that idicates if an atom was yet visited or not

        if ((visited = (booleano *)malloc(sizeof(booleano)*nAtoms)) == NULL) {
                printf("Error on memory allocation for 'visited'\n");
                return -1;
        }

        // Initialize all the array 'visited' to FALSE

        for(i=0;i<nAtoms;++i) {
                visited[i] = FALSE;
        }
```

```
	// Compute the euclidean distance between all pairs of atoms ----------
	lx = (float)(latP->nbx * latP->lc);
	ly = (float)(latP->nby * latP->lc);
	lz = (float)(latP->nbz * latP->lc);

	sizeDist = nAtoms*nAtoms*sizeof(float);

	if ((dist = (float *)malloc(nAtoms*nAtoms*sizeof(float))) == NULL) {
		printf("Error on memory allocation for 'dist'\n");
		free(visited);
		return (-1);
	}
	for(i=0;i<nAtoms;++i) {
		for(j=i+1;j<nAtoms;++j) {
			*(dist+i*nAtoms+j) = euclideanDistancePBC(*(atoms+i),*(atoms+j),lx,ly,lz);
			*(dist+j*nAtoms+i) = *(dist+i*nAtoms+j);
		}
	}

	// Create an array of 'Arraylist' necessary to store all atoms' neighboorhood

	N = (ArrayList *) initArrayWithSizeAndIncrementRate(nAtoms, INITIAL_CAPACITY,
		INCREMENT_CAPACITY);

	// Compute the neighborhood of each atom on a radius of value 'eps' --------

	getNeighbors_1(atoms, dist, nAtoms, eps, N);

	cid = 0;	// cid = current cluster ID
		// pid = atom position on array of atoms

	for(pid=0;pid<nAtoms;++pid) { // cycle over all atoms ----------
		p			= atoms[pid];
		visited[pid]		= TRUE;
		sizeN			= size(N+pid);

		if(sizeN<minPts){
			atoms[pid].classId = NOISE;
		}
		else {
			if(ExpandCluster(atoms, nAtoms, visited, N, pid, cid, eps, minPts) == TRUE)
				++cid;
		}
	} // (END OF) cycle over all atoms ----------

	// free memory space ----------

	free(visited);
	freeMemoryArray(N, nAtoms);
	free(dist);

	return (cid);
}
```

xiii. dbscan2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "main_includes.h"
#include "ArrayList.h"

// Date: 11 September 2012

// ***************************************************************************
//  Compute the neighborhood of all atoms on a radius of value 'eps'.
//
//  Parameters:
//                      atoms       Pointer to the array where the atoms (coordinates, type, classID) are stored.
//                      nAtoms      The number atoms.
//                      eps         The radius used to define the neighborhood of each atom.
//                      N           Pointer to the array of 'Arraylist' where atoms' neighboorhood
//                                  will be stored (ONLY atom's position NOT atom's data).
//                      latP        Pointer to the lattice parameters.
//
//  Return: ----
//
void getNeighbors_2 (point3Dclassified *atoms, int nAtoms, float eps, ArrayList* N,
                     LatticeParameters *latP) {
            int          i, j;
            float   dist, lx, ly, lz;

            // Simulation cell size ----------------------------------------------------

            lx = (float)(latP->nbx * latP->lc);
            ly = (float)(latP->nby * latP->lc);
            lz = (float)(latP->nbz * latP->lc);

            for(i=0;i<nAtoms;++i) {

                        for(j=0;j<nAtoms;++j) {
                                    if (j != i) {

                                                // Compute the euclidean distance between atoms i,j ------------
                                                dist = euclideanDistancePBC(*(atoms+i),*(atoms+j),lx,ly,lz);

                                                if(dist <= eps) {
                                                            add(N+i, (Element){j});
                                                }
                                    }
                        }
            }
}
```

```
// ********************************************************************************
//  DBSCAN clustering algorithm (with the distance between points computed
//  only when it is needed; distances are NOT computed and stored before).
//
//  Parameters:
//                      atoms       Pointer to the array where the atoms (coordinates, type, classID) are stored.
//                      nAtoms      The number atoms.
//                      eps         The radius used to define the neighborhood of each atom.
//                      minPts      Minimum number of neighbors that makes an atom to be a core atom.
//                      latP        Pointer to the lattice parameters.
//
//  Return: The number of identified clusters.
//
int DBSCAN_2 (point3Dclassified *atoms, int nAtoms, float eps, int minPts, LatticeParameters *latP) {
        int                     i, j, cid, pid, sizeN;
        ArrayList               *N;
        point3Dclassified       p;
        booleano                *visited;

        // Create an array to store a flag that idicates if an atom was yet visited or not

        if ((visited = (booleano *)malloc(sizeof(booleano)*nAtoms)) == NULL) {
                printf("Error on memory allocation for 'visited'\n");
                return -1;
        }

        // Initialize all the array 'visited' to FALSE

        for(i=0;i<nAtoms;++i) {
                visited[i] = FALSE;
        }

        // Create an array of 'Arraylist' necessary to store all atoms' neighboorhood

        N = (ArrayList *) initArrayWithSizeAndIncrementRate(nAtoms, INITIAL_CAPACITY,
                INCREMENT_CAPACITY);

        // Compute the neighborhood of each atom on a radius of value 'eps' --------

        getNeighbors_2(atoms, nAtoms, eps, N, latP);

        cid = 0;   // cid = current cluster ID
                // pid = atom position on array of atoms

        for(pid=0;pid<nAtoms;++pid) { // cycle over all atoms ----------------------

                if (visited[pid] == FALSE) {            // atom 'pid' was not yet visited
                        visited[pid] = TRUE;             // mark atom 'pid' as visited

                        sizeN      = size(N+pid);

                        if(sizeN<minPts){
                                atoms[pid].classId = NOISE;
                        }
                        else {
                                if(ExpandCluster(atoms, nAtoms, visited, N, pid, cid, eps, minPts) == TRUE)
                                        ++cid;
                        }
```

```
                }
        } // (END OF) cycle over all atoms ---------------------------------------

        // free memory space ----------------------------------------------------

        free(visited);
        freeMemoryArray(N, nAtoms);

        return (cid);
}
```

xiv. fcclattice.c

```
#include <math.h>
#include <stdlib.h>
#include "main_includes.h"

// *********************************************************************
// Computes the linear position of an atom with index 'offset' in cell (Z,Y,X,).
//
// nby      Number of unit cells in the y direction.
// nbx      Number of unit cells in the x direction.
// cz       Z value of the cell including atom.
// cy       Y value of the cell including atom.
// cx       X value of the cell including atom.
// offset   Index of atom inside the cell.
//
// Return: the linear position of an atom.
//
int BoxAndOffset_to_LinearIndex (int nby, int nbx, int cz, int cy, int cx, int offset) {
        return 4*(nbx*(cz*nby+cy)+cx)+offset;
}

// *********************************************************************
// Given the linear position of an atom compute its position in terms of cell 'offset' and
// cell postion (Z,Y,X,).
//
// Parameters:
//      index     Linear position of an atom.
//      nby       Number of unit cells in the y direction.
//      nbx       Number of unit cells in the x direction.
//      cz        Pointer to the variable where Z value of the cell (that includes the atom) will be stored.
//      cy        Pointer to the variable where Y value of the cell (that includes the atom) will be stored.
//      cx        Pointer to the variable where X value of the cell (that includes the atom) will be stored.
//      offset    Pointer to the variable where OFFSET inside the cell (that includes the atom) will be stored.
//
void LinearIndex_to_BoxAndOffse t(int index, int nby, int nbx, int *cz, int *cy, int *cx, int *offset) {

        *offset = index % FCC_ATOMS_CELL;
        index  -= *offset;
        *cz   = index/(nby*nbx*FCC_ATOMS_CELL);
        index  -= *cz*nby*nbx*FCC_ATOMS_CELL;
        *cy = index/(nbx*FCC_ATOMS_CELL);
        index  -= *cy*nbx*FCC_ATOMS_CELL;
        *cx  = index/FCC_ATOMS_CELL;
}
```

```
// ****************************************************************
// This function creates a list of coordinates of fcc lattice points.
//
//  Parameters:
//          latParam            Struct with the following fields previously setup:
//                  nbx                 Number of unit cells in the x direction.
//                  nby                 Number of unit cells in the y direction.
//                  nbz                 Number of unit cells in the z direction.
//                  lc                  Aluminum lattice constant.
//                  ScC                 Scandium percentage (0.0-100.0).
//          atomPosition        Pointer to the array with lattice atoms' coordinates.
//          atomType            Pointer to the array with lattice atoms' type.
//          vacancyPos          Pointer to save the position of the random vacancy.
//
//  Returns:
//                  ----
//
booleano fcclattice (LatticeParameters *latParam, point3d  atomPosition[],
          AtomType atomType[], int *vacancyPos) {

          int                 Natoms, cx, cy, cz, a, randNum, position;
          booleano            found = FALSE;
          FCCubeXYZ           cube;
          double              half_lc;
          int                 *ScAtom;

          Natoms  = latParam->nbx*latParam->nby*latParam->nbz*FCC_ATOMS_CELL;
          half_lc = latParam->lc/2.0;

          // define atoms' relative coordinates in a FCC
          cube.site[0].x=0.0;  cube.site[0].y=0.0; cube.site[0].z=0.0;
          cube.site[1].x=half_lc; cube.site[1].y=half_lc; cube.site[1].z=0.0;
          cube.site[2].x=half_lc; cube.site[2].y=0.0;          cube.site[2].z=half_lc;
          cube.site[3].x=0.0;  cube.site[3].y=half_lc; cube.site[3].z=half_lc;

          // Define atoms' absolute coordinates in all FCCs.
          // (By default) assign all atoms type to ALUMINIUM (Al).
          for(cz=0;cz<latParam->nbz;++cz) {
                    for(cy=0;cy<latParam->nby;++cy) {
                              for(cx=0;cx<latParam->nbx;++cx) {
                                        for(a=0;a<FCC_ATOMS_CELL;++a) {
                                                  position = FCC_ATOMS_CELL*(latParam->nbx*
                                                            (cz*latParam->nby+cy)+cx)+a;
                                                  atomPosition[position].z = cz*latParam->lc+cube.site[a].z;
                                                  atomPosition[position].y = cy*latParam->lc+cube.site[a].y;
                                                  atomPosition[position].x = cx*latParam->lc+cube.site[a].x;
                                                  atomType[position] = Al;
                                        }
                              }
                    }
          }

          // Number of Vacancy sites
          latParam->num_V_Sites = 1;

          // Number of Scandium sites/atoms
          latParam->num_Sc_Sites=(int)ceil(((double)Natoms*latParam->ScC)/100.0);
```

```
	// Select latParam->ScS % of atoms randomly and define its type to SCANDIUM (Sc)

	if ((ScAtom = (int *)malloc(latParam->num_Sc_Sites*sizeof(int *))) == NULL) {
		printf("Error on memory allocation in fcclattice()!\n");
		return FALSE;
	}

	// Generate 'latParam->num_Sc_Sites' random positions without repetition
	genNonRepeatedRandInteger(ScAtom,0,(Natoms-1),latParam->num_Sc_Sites);


	for (a=0;a<latParam->num_Sc_Sites;++a) {
		atomType[ScAtom[a]] = Sc;
	}

	// Select randomly ONE lattice site and define its type to 'Vacancy'

	randNum = genNotForbiddenRandInteger(ScAtom, latParam->num_Sc_Sites, 0, (Natoms-1));
	atomType[randNum] = Vacancy;
	*vacancyPos = randNum; // save the position of the vacancy in the lattice

	// Number of Aluminium sites/atoms
	latParam->num_Al_Sites=Natoms-latParam->num_Sc_Sites-latParam->num_V_Sites;

	printf("Natoms = %d  N_sc = %d   N_al = %d\n",Natoms, latParam->num_Sc_Sites,
		latParam->num_Al_Sites);

	free(ScAtom);

	return TRUE;

}

// Print the atoms' coordinates and type in all FCCs.
//
//  Parameters:
//		latParam		Struct with the following fields previously setup:
//			nbx:			number of unit cells in the x direction.
//			nby:			number of unit cells in the y direction.
//			nbz:			number of unit cells in the z direction.
//			lc:			Aluminum lattice constant.
//			ScC:			Scandium percentage (0.0-100.0).
//		atomPosition	Pointer to the array with lattice atoms' coordinates.
//		atomType		Pointer to the array with lattice atoms' type.
//
void printFccLattice (LatticeParameters *latParam, point3d  atomPosition[], AtomType atomType[]) {

	int  cx, cy, cz, a, position;

	for(cz=0;cz<latParam->nbz;++cz) {
		for(cy=0;cy<latParam->nby;++cy) {
			for(cx=0;cx<latParam->nbx;++cx) {
				for(a=0;a<FCC_ATOMS_CELL;++a) {
					position = FCC_ATOMS_CELL*(latParam->nbx*
						(cz*latParam->nby+cy)+cx)+a;

					printf("Atom in site %d or BOX_z,y,x=[%d][%d][%d] and in
						BoxPosition=%d has coordinates:\n",position,cz,cy,cx,a);
```

```
                                        printf("\t Z = %f  Y = %f  X = %f\n",atomPosition[position].z,
                                                atomPosition[position].y, atomPosition[position].x);
                                        printf("\t and is of type %d\n",atomType[position]);
                                }
                        }
                }
        }
}
```

xv. main.c

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "main_includes.h"

// Date: 29 September 2012

// *********************************************************************
// argument #1 -> configuration file
//
int main(int argc, char *argv[]) {

        LatticeParameters       latParam;
        AlScParameters          alScParam;
        SimulationParameters    simParam;
        EnergyParameters        energyParam;
        ClusteringParameters    clustParam;
        point3d                 *atomPosition;
        AtomType                *atomType, newType;
        NeighborsType           *atomN1N2;
        int                     latticeSize, seed;
        int                     posVacancy, newPosVacancy;
        double                  ts, tsCorrection;               // step/step correction time
        double                  Eact[NUM_1ST_NEIGHBORS];  // Activation energy for each 1st neighbor
        double                  vEF[NUM_1ST_NEIGHBORS]; // Exchange frequency for each 1st neighbor
        double                  sumVef[NUM_1ST_NEIGHBORS];    // Accumulated exchange frequency
                                                              // for each 1st neighbor
        time_t                  end, start;
        double                  dif = 0;
        TimeInterval            timeSim, timeRun, tsNewFormat;
        unsigned long long      mcs, deltaPrints, numSnap, *snapshots;
        char                    fConfig[200]="AlScPrecipitation_config.txt";

        int                     errorSteps=0;
        TimeInterval             *snapshotTime;   // array of snapshots' corrected time
        unsigned long long      aux, mcs_compAvgTime;
        double                  averageStepTime, rejectStepTime;

        printf("*********************************************************************\n");
        printf("*              Monte Carlo Simulation of Al3Sc Precipitation        *\n");
        printf("*********************************************************************\n");
```

```
if(argc < 2) {
        printf("Run program with the following arguments:\n");
        printf("  argument #1 -> configuration file\n");
        printf("*********************************************************************\n");
        return(-1);
}
else {
        if (readConfigFile(argv[1], &simParam, &energyParam, &alScParam, &latParam, &clustParam)
                == FALSE)
                return(-1);
}

printAlScParameters(&alScParam);
printLatticeParameters(&latParam);
printSimulParameters(&simParam);
printEnergyParameters(&energyParam);
printf("*********************************************************************\n");

if ((snapshots = (unsigned long long *)malloc((simParam.snap+1ULL)*sizeof(unsigned long long)))
        == NULL) {
        printf("Error on memory allocation for 'snapshots'\n");
        return (-2);
}

snapshotsNumber(&simParam, snapshots);

if ((snapshotTime = (TimeInterval *)malloc(simParam.snap*sizeof(TimeInterval))) == NULL) {
        printf("Error on memory allocation for 'snapshotTime'\n");
        free(snapshots);
        return (-3);
}

// compute the number of steps between sucessive prints of a "." on screen
if (simParam.mcs>NUM_DOTS_PRINT) {
        deltaPrints = simParam.mcs/NUM_DOTS_PRINT;
        printf("MCS                     = %lld\n",simParam.mcs);
        printf("NUM_DOTS_PRINT = %lld\n",NUM_DOTS_PRINT);
        printf("DELTA_PRINTS (#1)  = %lld\n",deltaPrints);
}
else {
        deltaPrints = simParam.mcs;
        printf("MCS                     = %lld\n",simParam.mcs);
        printf("NUM_DOTS_PRINT = %lld\n",NUM_DOTS_PRINT);
        printf("DELTA_PRINTS (#2)  = %lld\n",deltaPrints);
}

// Allocate memory space to store the lattice: atoms' coordinates, type and neighbors
latticeSize = FCC_ATOMS_CELL*latParam.nbx*latParam.nby*latParam.nbz;

printf("Lattice size is %d\n",latticeSize);
printf("Lattice 'atomPosition' array size is %d\n",latticeSize*sizeof(point3d));

if ((atomPosition = (point3d *)malloc(latticeSize*sizeof(point3d))) == NULL) {
        printf("Error on memory allocation for 'atomPosition'\n");
        free(snapshots);
        free(snapshotTime);
        return (-4);
}
```

```
if ((atomType = (AtomType *)malloc(latticeSize*sizeof(AtomType))) == NULL) {
        printf("Error on memory allocation for 'atomType'\n");
        free(snapshots);
        free(snapshotTime);
        free(atomPosition);
        return (-5);
}


if ((atomN1N2 = (NeighborsType *)malloc(latticeSize*sizeof(NeighborsType))) == NULL) {
        printf("Error on memory allocation for 'atomN1N2'\n");
        free(snapshots);
        free(snapshotTime);
        free(atomPosition);
        free(atomType);
        return (-6);
}

// Compute the coordinates of all FCC lattice sites

fcclattice(&latParam, atomPosition,     atomType, &posVacancy);

// printFccLattice(&latParam, atomPosition, atomType);  // DEBUG ONLY !!!!!!!!!

// Compute correction factor to apply to each MC step simulation time (ts)

tsCorrection = simVacancyConcentration(&latParam)/realVacancyConcentration(simParam.T);
printf("Correction factor to apply to each MC step simulation time is %e\n",tsCorrection);

// Compute the position of the 1st and 2nd nearest neighbors of all sites

printf("Compute the 1st and 2nd nearest neighbors of all sites ...\n");
allSitesNeighbors(latticeSize, &latParam, atomN1N2);

// Write initial configuration (atoms' position and type to file(s)

printf("Write initial configuration (positions and types) to file(s) ...\n");
if (simParam.wrXYZ == TRUE) {         // Write XYZ files
        writeXYZfile(&simParam, 0, atomPosition, atomType, latticeSize);
}
else if (simParam.wrVTK == TRUE) { // Write VTK files
        if (simParam.typeVTKdata == OnlyAtoms)          // write only atoms
                writeVTKfileNoLines(&simParam, &latParam, 0, atomPosition, atomType);
        else if (simParam.typeVTKdata == AtomsCubes) // write atoms and lattice cubes
                writeVTKfileWithLines(&simParam, &latParam, 0, atomPosition, atomType);
        }
else if (simParam.wrPDB == TRUE) { // Write PDB files
        writePDBfile(&simParam, 0, atomPosition, atomType, latticeSize);
}

printf("\nInitial Vacancy site is %d\n",posVacancy);
printf("*********************************************************************\n");
printf("Simulating \n");
fflush(stdout);
```

```
// ***************************************************************************
// Start Monte Carlo simulation

// Initialize simulated time to ZERO
timeSim.days    = 0ULL;
timeSim.hours   = 0;
timeSim.minutes = 0;
timeSim.seconds = 0.0;

numSnap = 1;
seed = time(NULL);
srand(seed);

time (&start);

// Run 'mcs_compAvgTime' MC steps to compute an average step time ----------

Aux = simParam.mcs/(2*simParam.snap);
mcs_compAvgTime = (aux < MCS_COMP_AVGTIME) ? aux : MCS_COMP_AVGTIME;
averageStepTime = 0.0;

for(mcs=0;mcs<mcs_compAvgTime;++mcs) {

        // Calculate the activation energy
        activationEnergy(&simParam, &energyParam, posVacancy, atomN1N2, atomType, Eact);

        // Calculate the vacancy exchange frequency and the real time of this MC step
        ts = vacancyExchangeFrequency(&simParam, &energyParam, posVacancy, atomN1N2,
        atomType, Eact, vEF, sumVef);

        // Calculate the corrected simulated time for current MCS
        ts *= tsCorrection;
        convertTime(ts, &tsNewFormat);

        // compute the accumulated simulated time (in format dd..d:hh:mm:ss)
        addTimes(&timeSim, timeSim, tsNewFormat);

        averageStepTime += ts; // accumulated time (in seconds) used to compute average step time

        // Select a 1st nearest neighbor for the new position of the vacancy
        newPosVacancy = randomNeighborSelection(vEF, sumVef, posVacancy, atomN1N2);

        // swap the vacancy with the selected neighbor
        newType                 = atomType[newPosVacancy];
        atomType[posVacancy]    = newType;
        atomType[newPosVacancy] = Vacancy;
        posVacancy              = newPosVacancy;
}
averageStepTime = averageStepTime / mcs_compAvgTime;  // compute average step time
rejectStepTime  = averageStepTime*(double)FACTOR_REJECT_MCS; // compute rejection step time

printf("Average step time after %lld steps is %e\n",mcs_compAvgTime, averageStepTime);
printf("Rejection step time is %e\n",rejectStepTime);
```

```
// Run the remaining steps of MC simulation --------------------------------

for(;mcs<simParam.mcs;++mcs) {

        // Calculate the activation energy
        activationEnergy(&simParam, &energyParam, posVacancy, atomN1N2, atomType, Eact);

        // Calculate the vacancy exchange frequency and the real time of this MC step
        ts = vacancyExchangeFrequency(&simParam, &energyParam, posVacancy, atomN1N2,
        atomType, Eact, vEF, sumVef);

        // Calculate the corrected simulation time for current MCS
        ts *= tsCorrection;

        // THRESHOLD VALUE THAT MAY INDICATE A STEP WITH ERRONEOUS TIME
        if (ts > rejectStepTime) {
                ++errorSteps;
                ts   = averageStepTime; // replace computed step time by average step time
        }

        convertTime(ts, &tsNewFormat);
        // compute the accumulated simulated time
        addTimes(&timeSim, timeSim, tsNewFormat);

        // Select a 1st nearest neighbor for the new position of the vacancy
        newPosVacancy = randomNeighborSelection(vEF, sumVef, posVacancy, atomN1N2);

        // swap the vacancy with the selected neighbor
        newType                 = atomType[newPosVacancy];
        atomType[posVacancy]    = newType;
        atomType[newPosVacancy] = Vacancy;
        posVacancy              = newPosVacancy;

        // save simulation data (atoms' type & position) to file at snapshop points
        if (mcs == snapshots[numSnap]) {

                if (simParam.wrXYZ == TRUE) {         // Write XYZ files
                        writeXYZfile(&simParam, numSnap, atomPosition, atomType, latticeSize);
                }
                else if (simParam.wrVTK == TRUE) { // Write VTK files
                        if (simParam.typeVTKdata == OnlyAtoms)         // write only atoms
                                writeVTKfileNoLines(&simParam, &latParam, numSnap,
                                        atomPosition, atomType);
                        else if (simParam.typeVTKdata == AtomsCubes) // write atoms+ lattice cubes
                                writeVTKfileWithLines(&simParam, &latParam, numSnap,
                                        atomPosition, atomType);
                }
                else if (simParam.wrPDB == TRUE) { // Write PDB files
                        writePDBfile(&simParam, numSnap, atomPosition, atomType, latticeSize);
                }

                // Save the snapshot simulated time (timeSim)
                snapshotTime[numSnap-1] = timeSim;

                printf("\nSnapshot %3lld time is %lld d : %d h : %d m : %.16e s.\n", numSnap,
                        snapshotTime[numSnap-1].days, snapshotTime[numSnap-1].hours,
                        snapshotTime[numSnap-1].minutes, snapshotTime[numSnap-1].seconds);
```

```
                ++ numSnap;
            }

            if(mcs%deltaPrints==(deltaPrints-1ULL)){
                printf(". ");
                fflush(stdout);
            }

        }

        //**********************************************************************
        time(&end);
        dif  = difftime(end,start);
        convertTime(dif, &timeRun);
        printf("\n");
        printf("The duration of simulation was  %lld d : %d h : %d m : %.0f s.\n",
            timeRun.days, timeRun.hours, timeRun.minutes, timeRun.seconds);

        //**********************************************************************
        printf("Simulated time was %lld d : %d h : %d m : %e s.\n",
            timeSim.days, timeSim.hours, timeSim.minutes, timeSim.seconds);

        //**********************************************************************
        printf("Number of steps with a suspictious erroneous duration was %d\n",errorSteps);

        //**********************************************************************
        // write a simulation report to file
        if(simParam.wrReport == TRUE){
            writeSimulationReport(&simParam, &energyParam, &latParam, &timeRun, &timeSim,
                snapshotTime);
        }

        // **********************************************************************
        // Free allocated memory space

        free(snapshots);
        free(snapshotTime);
        free(atomPosition);
        free(atomType);
        free(atomN1N2);

        return(0);
}
```

xvi. main_includes.h

```
#include <stdio.h>
#include "ArrayList.h"

// Date: 11 october 2012

// CONSTANTS ***************************************************

#define PI                      3.14159265358979323846264338327
#define FCC_ATOMS_CELL          4       // Number of atoms per cell on a FCC lattice
#define NUM_1ST_NEIGHBORS       12      // Number of nearest neighbors in FCC
#define NUM_2ND_NEIGHBORS       6       // Number of 2nd nearest neighbors in FCC
```

```
#define MAX_LINE_SIZE		500	// Maximum size of a line to read from file
#define NUM_PARAMETERS		31	// Number of configuration parameters
#define ENTER			13
#define NEWLINE			10
#define NUM_DOTS_PRINT		10000ULL // Number of "." to print during simulation
#define AL_POS			0
#define SC_POS			1
#define V_POS			2
#define UNCLASSIFIED		-1
#define NOISE			-2
#define MCS_COMP_AVGTIME	1000  // Number of MC steps used to compute the average time of a MCS.
#define FACTOR_REJECT_MCS	100	// Multiplicative factor used to reject a MC step simulated time.
					// These 2 values will be used to reject the simulated time of
					// a MC step that is too far from the average value.

// DATA TYPES ************************************************

typedef enum {Al=13, Sc=21, Vacancy=0} AtomType;
typedef enum {AL=13, SC=21, VACANCY=0, ALSC=1000, ALV=2000, SCV=3000, ALL=4000} SelectedType;
typedef enum {FALSE, TRUE} booleano;
typedef enum {OnlyAtoms, AtomsCubes} OutputType; // write in VTK file only atoms

// or atoms and lattice cubes.

typedef struct key_value { // KEY:VALUE pair
	char key[MAX_LINE_SIZE];
	char value[MAX_LINE_SIZE];
} KeyValue;

typedef struct al_sc_parameters {
	double al_Radius;  // Aluminum atom radius (Aº)
	double sc_Radius;  // Scandium atom radius (Aº)
	double lc;		// Aluminum lattice constant (Aº)
} AlScParameters;

typedef struct point3D { // coordinates of a 3D point/atom
	double z,y,x;
} point3d;

typedef struct ffc { // coordinates of the 4 FCC unit cell points/atoms
	point3d site[FCC_ATOMS_CELL];
} FCCubeXYZ;

typedef struct ffc_type { // types of the 4 atoms in a FCC unit cell
	AtomType site[FCC_ATOMS_CELL];
} FCCubeType;

typedef struct point3D_classified { // 3D point/atom (coordinates, type, class)
	double   z, y, x;
	AtomType type;
	int	 classId;
} point3Dclassified;

typedef struct neighbors { // 1st and 2nd neighborhoods of an atom
	int N1[NUM_1ST_NEIGHBORS];
	int N2[NUM_2ND_NEIGHBORS];
} NeighborsType;
```

```
typedef struct lattice_parameters {
        int     nbz;            // Number of unit cells in the z direction
        int     nby;            // Number of unit cells in the y direction
        int     nbx;            // Number of unit cells in the x direction
        int     num_Al_Sites;   // Number of Aluminum atoms/sites
        int     num_Sc_Sites;   // Number of Scandium atoms/sites
        int     num_V_Sites;    // Number of Vacancy sites
        double lc;              // Aluminum lattice constant
        double ScC;             // Scandium percentage (0.0-100.0)
} LatticeParameters;

typedef struct simul_parameters {
        unsigned long long  mcs;            // Simulation Monte Carlo steps
        unsigned long long  snap;           // Number of snapshots to save during simulation
        double              T;              // Simulation temperature (Kelvin)
        double              kB;             // Boltzmann Constant (eV/K)
        booleano            wrXYZ;          // Indicates if we want to write XYZ files (TRUE) or not (FALSE)
        booleano            wrVTK;          // Indicates if we want to write VTK files (TRUE) or not (FALSE)
        booleano            wrPDB;          // Indicates if we want to write PDB files (TRUE) or not (FALSE)
        booleano            wrReport;       // Indicates if we want to write a simulation report to file
        char                coordFile[200]; // Base name of the output file(s) to write atoms' coordinates & type
        SelectedType        selectedType;   // Selected type of atoms to write into (VTK) file
        OutputType          typeVTKdata;    // Type of data to write in VTK file: only atoms OR atoms & cubes
} SimulationParameters;

typedef struct energy_parameters {
        double E_AlAl_1;  // 1st nearest neighbor pair effective energies
        double E_ScSc_1;  // 1st nearest neighbor pair effective energies
        double E_AlSc_1;  // 1st nearest neighbor pair effective energies
        double E_AlSc_2;  // 2nd nearest neighbor pair effective energies
        double E_AlAl_2;  // 2nd nearest neighbor pair effective energies
        double E_ScSc_2;  // 2nd nearest neighbor pair effective energies
        double E_AlV_1;   // 1st nearest neighbor pair effective energies
        double E_ScV_1;   // 1st nearest neighbor pair effective energies
        double e_spAl;    // Saddle point energy
        double e_spSc;    // Saddle point energy
        double vAl;       // Aluminum attempt frequency
        double vSc;       // Scandium attempt frequency
} EnergyParameters;

typedef struct time_interval {
        unsigned long long days;
        int                hours;
        int                minutes;
        double             seconds;
} TimeInterval;

typedef struct clustering_parameters {
        float   eps;            // The radius used to define the neighborhood of each atom
                                // (DBSCAN algorithm)
        int     minPts;         // Minimum number of neighbors that makes an atom to be
                                // a core atom of a cluster (DBSCAN algorithm)
        booleano storeDistances;// Store (TRUE) or do NOT store (FALSE) the distance
                                // between each pair of atoms in advance (DBSCAN algorithm)
        int     minSize;        // Minimum number of atoms to consider a cluster as valid
        booleano stablePrecipitates; // execute stable precipitates analysis (if TRUE)
        booleano smallClusters; // execute small clusters analysis (if TRUE)
} ClusteringParameters;
```

```
typedef struct analysis_report_param {
        char    *inputVTK;          // VTK file used as input in clustering analysis.
        char    *outputVTK;         // VTK file generated by the clustering analysis.
        char    *configFile;        // Configuration file used as input in clustering analysis.
        int      nClusters;         // Number of clusters identified by the clustering analysis.
        int      nPrecipitates;     // Number of stable precipitates identified by the clustering analysis.
        int      *sizeCluster;      // Size of each cluster (in atoms).
        int      avgSize;           // Average size among all clusters (in atoms).
        double   *radiusCluster;    // Approximated radius of each cluster (in Angstrom).
        double   avgRadius;         // Average radius among all clusters (in Angstrom).
        double   percentClustered;      // Percentage of Sc atoms in precipitates (clusters)
        double   percentNotClustered; // Percentage of Sc atoms in Al solid solution (not clustered)
        int      *nClustersSize;        // Number of clusters with the same size (clustParam->minSize)
} AnalysisReportParam;

// FUNCTIONS ***********************************************************

void      defAlScParameters(AlScParameters *);
void      defLatticeParameters(LatticeParameters *);
void      defSimulParameters(SimulationParameters *);
void      defEnergyParameters(EnergyParameters *);
void      defClusterParameters(ClusteringParameters *);
void      snapshotsNumber(SimulationParameters *, unsigned long long *);
void      printAlScParameters(AlScParameters *);
void      printLatticeParameters(LatticeParameters *);
void      printSimulParameters(SimulationParameters *);
void      printEnergyParameters(EnergyParameters *);
void      printClusterParameters(ClusteringParameters *);

booleano fcclattice(LatticeParameters *, point3d [], AtomType [], int *);
void      printFccLattice(LatticeParameters *, point3d  [], AtomType []);
int       BoxAndOffset_to_LinearIndex(int, int, int, int, int, int);
void      LinearIndex_to_BoxAndOffset(int, int, int, int *, int *, int *, int *);

int       roundDouble2integer(double);
void      genRandDouble(double *, double, double, int);
void      genRandInteger(int *, int, int, int);
void      genNonRepeatedRandInteger(int *, int, int, int);
int       genNotForbiddenRandInteger(int *, int, int, int);
void      genRandDoubleNoSeed(double *, double, double, int);
void      convertTime(double, TimeInterval *);
void      addTimes(TimeInterval *, TimeInterval, TimeInterval);

void      allSitesNeighbors(int, LatticeParameters *, NeighborsType []);
void      neighbors(int, LatticeParameters *, AtomType *, int *,
          AtomType *, int *, AtomType *);
void      activationEnergy(SimulationParameters *, EnergyParameters *, int,
          NeighborsType [], AtomType *, double *);
double    vacancyExchangeFrequency(SimulationParameters *, EnergyParameters *,
          int, NeighborsType [], AtomType *, double *, double *,     double *);
int       randomNeighborSelection(double *, double *,int, NeighborsType []);
double    realVacancyConcentration(double);
double    simVacancyConcentration(LatticeParameters *);

void      writeXYZfile(SimulationParameters *, int, point3d [],AtomType [], int);
void      writePDBfile(SimulationParameters *, int, point3d [],AtomType [], int);
```

```
void        writeVTKfileWithLines(SimulationParameters *, LatticeParameters *, int,
            point3d [],AtomType []);
void        writeVTKfileSimple(SimulationParameters *, int, point3d [],
            AtomType [], int , int );
void        writeVTKfileNoLines( SimulationParameters *, LatticeParameters *,
            int, point3d [], AtomType []);
void        writeSimulationReport(SimulationParameters *, EnergyParameters *,
            LatticeParameters *, TimeInterval *, TimeInterval *, TimeInterval *);
point3Dclassified  *readVtkFile(char *, int *);
void         writeVTKfileClusters(char *, point3Dclassified *, int);

booleano readConfigFile(char *, SimulationParameters *, EnergyParameters *,
            AlScParameters *, LatticeParameters *, ClusteringParameters *);
booleano readLine(FILE *, char *);

booleano ExpandCluster(point3Dclassified *, int, booleano *, ArrayList *,
            int, int, float, int);
void        printNeighbors(ArrayList *, int);
void        printClusters(point3Dclassified *, int);
float       euclideanDistancePBC(point3Dclassified, point3Dclassified, float, float, float);
int         DBSCAN_1(point3Dclassified *, int, float, int, LatticeParameters *);
int         DBSCAN_2(point3Dclassified *, int, float, int, LatticeParameters *);
void        organizeClusters(point3Dclassified *, int, ArrayList *);
void        getNeighborsInCluster(point3Dclassified *, int, ArrayList *, int, float,
            ArrayList*, LatticeParameters *);
void        mergeClusters(point3Dclassified *, int, ArrayList *, int, ArrayList *, LatticeParameters *);
int         removeSmallClusters(point3Dclassified *, int, ArrayList *, int, int);
int*        sizeClusters(point3Dclassified *, int, ArrayList *,          int);
void        printSizeClusters(int *, int);
int         averageSizeClusters(int *, int);
double*     radiusClusters(int *, int, double);
void        printRadiusClusters(double *, int);
double      averageRadiusClusters(double *, int);
void        writeAnalysisReport(SimulationParameters *, LatticeParameters *,
            ClusteringParameters *, AnalysisReportParam *);
void        writeSimpleClustersReport(SimulationParameters *, LatticeParameters *,
            ClusteringParameters *, AnalysisReportParam *);
void        countNumberSmallClusters(ArrayList *, int, ClusteringParameters *, AnalysisReportParam *);
```

xvii. main2.c

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "main_includes.h"

// Date: 09 October 2012

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

            LatticeParameters           latParam;
            AlScParameters              alScParam;
```

```
SimulationParameters        simParam;
EnergyParameters            energyParam;
ClusteringParameters        clustParam;
AnalysisReportParam         analysisRP;

point3Dclassified           *atoms;
int                         nAtoms, nAtomsSim, *sizeCluster, avgSize, n;
ArrayList                   *cluster, *N;
double                      *radiusCluster, avgRadius, num;

analysisRP.nPrecipitates    = 0;
analysisRP.nClusters        = 0;
cluster                     = NULL;
N                           = NULL;
sizeCluster                 = NULL;
radiusCluster               = NULL;
analysisRP.nClustersSize    = NULL;

printf("*****************************************************************************\n");
printf("*   Analysis of Monte Carlo Simulation of Al3Sc Precipitation               *\n");
printf("*****************************************************************************\n");
if(argc < 4) {
        printf("Run program with the following arguments:\n");
        printf("  argument #1 -> input  VTK file\n");
        printf("  argument #2 -> output VTK file\n");
        printf("  argument #3 -> configuration file\n");
        printf("*****************************************************************************\n");
        return(-1);
}
else {
        if (readConfigFile(argv[3], &simParam, &energyParam, &alScParam,
                &latParam, &clustParam) == FALSE)
                return(-1);

        // Derive input parameters used in simulation

        nAtomsSim = latParam.nbx*latParam.nby*latParam.nbz*FCC_ATOMS_CELL;
        latParam.num_V_Sites = 1;
        latParam.num_Sc_Sites=(int)ceil(((double)nAtomsSim*latParam.ScC)/100.0);
        latParam.num_Al_Sites=nAtomsSim-latParam.num_Sc_Sites-latParam.num_V_Sites;

        printf("Configuration parameters:\n");
        printAlScParameters(&alScParam);
        printLatticeParameters(&latParam);
        printSimulParameters(&simParam);
        printEnergyParameters(&energyParam);
        printClusterParameters(&clustParam);

        printf("*****************************************************************************\n");
        printf("Reading input VTK file ...");
        if ((atoms=readVtkFile(argv[1], &nAtoms)) == NULL)
                return(-1);
        printf(" done!\n");
}
```

```
// ------------------------------------------------------------------------
// apply the DBSCAN clustering algorithm

printf("Applying DBSCAN clustering algorithm ...");
if(clustParam.storeDistances == TRUE)
        analysisRP.nClusters = DBSCAN_1(atoms, nAtoms, clustParam.eps, clustParam.minPts,
                &latParam);
else if(clustParam.storeDistances == FALSE)
        analysisRP.nClusters = DBSCAN_2(atoms, nAtoms, clustParam.eps, clustParam.minPts,
                &latParam);
printf(" done!\n");

// Create an array of 'Arraylist' necessary to store all clusters' atoms

if (analysisRP.nClusters > 0) { // DBSCAN FOUND CLUSTERS ----------

        cluster = (ArrayList *) initArrayWithSizeAndIncrementRate(analysisRP.nClusters,
                INITIAL_CAPACITY, INCREMENT_CAPACITY);

        // ------------------------------------------------------------------------
        // Organize the array of clustered points (atoms) in an array of ArrayList's (cluster),
        // with the atoms of each cluster stored in a different position of the array.

        organizeClusters(atoms, nAtoms, cluster);

        // Create an array of Arraylist's necessary to store all atoms' neighboorhood

        N = (ArrayList *) initArrayWithSizeAndIncrementRate(nAtoms, INITIAL_CAPACITY,
                INCREMENT_CAPACITY);

        // Compute the neighborhood (inside a cluster) of each atom

        getNeighborsInCluster(atoms, nAtoms, cluster, analysisRP.nClusters, clustParam.eps, N,
                &latParam);

        // ------------------------------------------------------------------------
        // Merge the clusters that are splited in several parts in a sigle spatial
        // region per cluster

        printf("Number of clusters is %d\n", analysisRP.nClusters);

        printf("Merging splited clusters ...");
        mergeClusters(atoms, nAtoms, cluster, analysisRP.nClusters, N, &latParam);
        printf(" done!\n");

        // ------------------------------------------------------------------------
        // Execute a small clusters analysis
        // ------------------------------------------------------------------------

        if (clustParam.smallClusters == TRUE) {
                printf("Counting the small clusters with the same size...", analysisRP.nClusters);
                if ((analysisRP.nClustersSize = (int *)malloc((clustParam.minSize+1)*sizeof(int)))
                        == NULL) {
                        printf("  Error on memory allocation for 'analysisRP.nClustersSize'\n");
                }
                else {
                        countNumberSmallClusters(cluster, analysisRP.nClusters, &clustParam,
                                &analysisRP);
```

```
		}
		printf(" done!\n");
}

// ---------------------------------------------------------------------
// Execute a stable precipitates analysis
// ---------------------------------------------------------------------

if (clustParam.stablePrecipitates == TRUE) {
		// Remove the clusters that have a size smaller than a given threshold

		printf("Removing the small sized clusters ...\n", analysisRP.nClusters);
		analysisRP.nPrecipitates = removeSmallClusters(atoms, nAtoms, cluster,
			analysisRP.nClusters, clustParam.minSize);

		printf("  The number of stable precipiates is now %d\n", analysisRP.nPrecipitates);

		if (analysisRP.nPrecipitates>0) { // THERE ARE STABLE PRECIPITATES

			// Compute and store in an array the size of all precipitates

			sizeCluster = sizeClusters(atoms, nAtoms, cluster, analysisRP.nPrecipitates);
			printSizeClusters(sizeCluster, analysisRP.nPrecipitates);

			// Compute and store in an array the (approximated) radius of all precipitates

			radiusCluster = radiusClusters(sizeCluster, analysisRP.nPrecipitates,
				latParam.lc);
			printRadiusClusters(radiusCluster, analysisRP.nPrecipitates);

			// Compute the average size among all precipitates

			avgSize = averageSizeClusters(sizeCluster, analysisRP.nPrecipitates);
			printf("*****************************************************\n");
			printf("Average size among all clusters is %d\n",avgSize);

			// Compute the average radius among all precipitates

			avgRadius = averageRadiusClusters(radiusCluster, analysisRP.nPrecipitates);
			printf("Average radius among all clusters is %.2f\n",avgRadius);

			// Compute the percentage of Sc atoms in precipitates (Al3Sc)

			for (n=0, num=0.0;n<analysisRP.nPrecipitates;++n) {
				num += sizeCluster[n];
			}
			analysisRP.percentClustered =
				(num*latParam.ScC)/(double)latParam.num_Sc_Sites;

			// Compute the percentage of Sc atoms in Al solid solution

			analysisRP.percentNotClustered = latParam.ScC –
				analysisRP.percentClustered;

			printf("Percentage of Sc atoms in Al solid solution is %.3f %%\n",
				analysisRP.percentNotClustered);
			printf("Percentage of Sc atoms in precipitates is %.3f %%\n",
				analysisRP.percentClustered);
```

```
                                // ************************************************************
                                // Prepare data to write the clustering analysis report.

                                analysisRP.inputVTK = argv[1];
                                analysisRP.outputVTK = argv[2];
                                analysisRP.configFile = argv[3];
                                analysisRP.sizeCluster   = sizeCluster;
                                analysisRP.avgSize   = avgSize;
                                analysisRP.radiusCluster = radiusCluster;
                                analysisRP.avgRadius = avgRadius;

                                // ------------------------------------------------------------
                                // Write a VTK file with clusters data: (x,y,z,clusterId) for each atom

                                printf("\nWriting output VTK file ...");
                                writeVTKfileClusters(argv[2], atoms, nAtoms);
                                printf(" done!\n");
                                printf("*********************************************************\n");

                        } // (end of) THERE ARE STABLE PRECIPITATES ------------------------

                } // (end of) Execute a stable precipitates analysis --------------------

        } // (end of) DBSCAN FOUND CLUSTERS ---------------------------------------

        // DBSCAN DID NOT FOUND ANY STABLE PRECIPITATE -----------------------------
        if (analysisRP.nPrecipitates==0) {
                // ************************************************************************
                // Prepare data to write the clustering analysis report.

                printf("Setup default data for analysis report...");

                analysisRP.inputVTK                 = argv[1];
                analysisRP.outputVTK                = argv[2];
                analysisRP.configFile               = argv[3];
                analysisRP.percentNotClustered      = 100.0;
                analysisRP.percentClustered         = 0.0;
                analysisRP.sizeCluster              = 0;
                analysisRP.avgSize                  = 0;
                analysisRP.radiusCluster            = 0;
                analysisRP.avgRadius                = 0;
                printf(" done!\n");
        }

        // ************************************************************************
        // * Write (to file) a report about clustering analysis: some data about simulation,
        //   input VTK file, output VTK file, configuratyion file, number of (stable)
        //   precipitates identified, precipitates' size, average precipitates' size,
        //   precipitates' radius, average precipitates' radius, number of small clusters
        //   with the same size.

        writeAnalysisReport(&simParam, &latParam, &clustParam, &analysisRP);

        // free memory space ------------------------------------------------

        if(analysisRP.nClustersSize != NULL)
                free(analysisRP.nClustersSize);
```

```
	if (radiusCluster != NULL)
		free (radiusCluster);
	if (sizeCluster != NULL)
		free (sizeCluster);
	if (N != NULL)
		freeMemoryArray(N, nAtoms);
	if (cluster != NULL)
		freeMemoryArray(cluster, analysisRP.nPrecipitates);
	free(atoms);

	return(0);
}
```

xviii. neighbors.c

```
#include "main_includes.h"

// ***************************************************************************
// Compute the position of the first and second nearest neighbors of all sites.
//
// Parameters:
//		latSize		Lattice size.
//		latP		Lattice parameters.
//		atomN1N2	Pointer to the array where the position of the 1st and 2nd
//				nearest neighbors will be saved.
//
//  Compute: atomN1N2
//
void allSitesNeighbors (int latSize, LatticeParameters *latP, NeighborsType atomN1N2[]) {

	int i=0, pos;
	int cz, cy, cx, offset, nbz, nby, nbx;
	int CzPlus,  CyPlus,  CxPlus;
	int CzMinus, CyMinus, CxMinus;

	nbz = latP->nbz;
	nby = latP->nby;
	nbx = latP->nbx;

	// Iterate over all lattice sites ---------------------------------------
	for (pos=0;pos<latSize;++pos) {

		// compute the coordinates of the cube that includes the site (cz, cy, cx)
		// and the position inside that cube (offset)

		LinearIndex_to_BoxAndOffset(pos, nby, nbx, &cz, &cy, &cx, &offset);

		CzMinus = cz-1; if(CzMinus<0) CzMinus=nbz-1;  // (1)
		CyMinus = cy-1; if(CyMinus<0) CyMinus=nby-1;  // (2)
		CxMinus = cx-1; if(CxMinus<0) CxMinus=nbx-1;  // (3)
		CzPlus  = cz+1; if(CzPlus==nbz) CzPlus=0;	// (4)
		CyPlus  = cy+1; if(CyPlus==nby) CyPlus=0;	// (5)
		CxPlus  = cx+1; if(CxPlus==nbx) CxPlus=0;	// (6)
```

```
// Compute the position and type of the 12 1st neighbors ***************

// ********************************************************************
if (offset == 0) {
        // 3 neighbors in same cube (cz,cy,cx) and offset={1,2,3}
        atomN1N2[pos].N1[0]=pos+1;
        atomN1N2[pos].N1[1]=pos+2;
        atomN1N2[pos].N1[2]=pos+3;

        // 2 neighbors in cube (cz,cy,cx-1) and offset={1,2}
        atomN1N2[pos].N1[3]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, cy, CxMinus, 1);
        atomN1N2[pos].N1[4]=atomN1N2[pos].N1[3]+1;


        // 2 neighbors in cube (cz-1,cy,cx) and offset={2,3}
        atomN1N2[pos].N1[5]=BoxAndOffset_to_LinearIndex(nby, nbx, CzMinus, cy, cx, 2);
        atomN1N2[pos].N1[6]=atomN1N2[pos].N1[5]+1;

        // 1 neighbor in cube (cz-1,cy,cx-1) and offset={2}
        atomN1N2[pos].N1[7]=BoxAndOffset_to_LinearIndex(nby, nbx, CzMinus, cy,
                CxMinus, 2);

        // 1 neighbor in cube (cz,cy-1,cx-1) and offset={1}
        atomN1N2[pos].N1[8]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, CyMinus,
                CxMinus, 1);

        // 2 neighbors in cube (cz,cy-1,cx) and offset={1,3}
        atomN1N2[pos].N1[9]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, CyMinus, cx, 1);
        atomN1N2[pos].N1[10]=atomN1N2[pos].N1[9]+2;

        // 1 neighbor in cube (cz-1,cy-1,cx) and offset={3}
        atomN1N2[pos].N1[11]=BoxAndOffset_to_LinearIndex(nby, nbx, CzMinus, CyMinus,
                cx, 3);
}

// ********************************************************************
else if (offset == 1) {
        // 3 neighbors in same cube (cz,cy,cx) and offset={0,2,3}
        atomN1N2[pos].N1[0]=pos-1;
        atomN1N2[pos].N1[1]=pos+1;
        atomN1N2[pos].N1[2]=pos+2;

        // 2 neighbors in cube (cz,cy,cx+1) and offset={0,3}
        atomN1N2[pos].N1[3]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, cy, CxPlus, 0);
        atomN1N2[pos].N1[4]=atomN1N2[pos].N1[3]+3;

        // 2 neighbors in cube (cz-1,cy,cx) and offset={2,3}
        atomN1N2[pos].N1[5]=BoxAndOffset_to_LinearIndex(nby, nbx, CzMinus, cy, cx, 2);
        atomN1N2[pos].N1[6]=atomN1N2[pos].N1[5]+1;

        // 2 neighbors in cube (cz,cy+1,cx) and offset={0,2}
        atomN1N2[pos].N1[7]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, CyPlus, cx, 0);
        atomN1N2[pos].N1[8]=atomN1N2[pos].N1[7]+2;

        // 1 neighbor in cube (cz,cy+1,cx+1) and offset={0}
        atomN1N2[pos].N1[9]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, CyPlus,
                CxPlus, 0);
```

```
		// 1 neighbor in cube (cz-1,cy,cx+1) and offset={3}
		atomN1N2[pos].N1[10]=BoxAndOffset_to_LinearIndex(nby, nbx, CzMinus, cy,
			CxPlus, 3);

		// 1 neighbor in cube (cz-1,cy+1,cx) and offset={2}
		atomN1N2[pos].N1[11]=BoxAndOffset_to_LinearIndex(nby, nbx, CzMinus,
			CyPlus, cx, 2);
}

// ************************************************************************
else if (offset == 2) {
		// 3 neighbors in same cube (cz,cy,cx) and offset={0,1,3}
		atomN1N2[pos].N1[0]=pos-2;
		atomN1N2[pos].N1[1]=pos-1;
		atomN1N2[pos].N1[2]=pos+1;

		// 2 neighbors in cube (cz,cy,cx+1) and offset={0,3}
		atomN1N2[pos].N1[3]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, cy, CxPlus, 0);
		atomN1N2[pos].N1[4]=atomN1N2[pos].N1[3]+3;

		// 2 neighbors in cube (cz+1,cy,cx) and offset={0,1}
		atomN1N2[pos].N1[5]=BoxAndOffset_to_LinearIndex(nby, nbx, CzPlus, cy, cx, 0);
		atomN1N2[pos].N1[6]=atomN1N2[pos].N1[5]+1;

		// 2 neighbors in cube (cz,cy-1,cx) and offset={1,3}
		atomN1N2[pos].N1[7]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, CyMinus, cx, 1);
		atomN1N2[pos].N1[8]=atomN1N2[pos].N1[7]+2;

		// 1 neighbor in cube (cz+1,cy,cx+1) and offset={0}
		atomN1N2[pos].N1[9]=BoxAndOffset_to_LinearIndex(nby, nbx, CzPlus, cy,
			CxPlus, 0);

		// 1 neighbor in cube (cz+1,cy-1,cx) and offset={1}
		atomN1N2[pos].N1[10]=BoxAndOffset_to_LinearIndex(nby, nbx, CzPlus,
			CyMinus, cx, 1);

		// 1 neighbor in cube (cz,cy-1,cx+1) and offset={3}
		atomN1N2[pos].N1[11]=BoxAndOffset_to_LinearIndex(nby, nbx, cz,
			CyMinus, CxPlus, 3);
}

// ************************************************************************
else if (offset == 3) {
		// 3 neighbors in same cube (cz,cy,cx) and offset={0,1,2}
		atomN1N2[pos].N1[0]=pos-3;
		atomN1N2[pos].N1[1]=pos-2;
		atomN1N2[pos].N1[2]=pos-1;

		// 2 neighbors in cube (cz+1,cy,cx) and offset={0,1}
		atomN1N2[pos].N1[3]=BoxAndOffset_to_LinearIndex(nby, nbx, CzPlus, cy, cx, 0);
		atomN1N2[pos].N1[4]=atomN1N2[pos].N1[3]+1;

		// 2 neighbors in cube (cz,cy+1,cx) and offset={0,2}
		atomN1N2[pos].N1[5]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, CyPlus, cx, 0);
		atomN1N2[pos].N1[6]=atomN1N2[pos].N1[5]+2;

		// 2 neighbors in cube (cz,cy,cx-1) and offset={1,2}
```

```
                        atomN1N2[pos].N1[7]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, cy, CxMinus, 1);
                        atomN1N2[pos].N1[8]=atomN1N2[pos].N1[7]+1;

                        // 1 neighbor in cube (cz+1,cy+1,cx) and offset={0}
                        atomN1N2[pos].N1[9]=BoxAndOffset_to_LinearIndex(nby, nbx, CzPlus,
                                CyPlus, cx, 0);

                        // 1 neighbor in cube (cz+1,cy,cx-1) and offset={1}
                        atomN1N2[pos].N1[10]=BoxAndOffset_to_LinearIndex(nby, nbx, CzPlus,
                                cy, CxMinus, 1);

                        // 1 neighbor in cube (cz,cy+1,cx-1) and offset={2}
                        atomN1N2[pos].N1[11]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, CyPlus,
                                CxMinus, 2);
                }

                // Compute the position of the six 2nd neighbors ************************

                // *********************************************************************
                // one 2nd neighbor in cube (cz,cy,cx-1) and offset
                atomN1N2[pos].N2[0]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, cy, CxMinus, offset);

                // one 2nd neighbor in cube (cz,cy,cx+1) and offset
                atomN1N2[pos].N2[1]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, cy, CxPlus, offset);

                // one 2nd neighbor in cube (cz,cy-1,cx) and offset
                atomN1N2[pos].N2[2]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, CyMinus, cx, offset);

                // one 2nd neighbor in cube (cz,cy+1,cx) and offset
                atomN1N2[pos].N2[3]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, CyPlus, cx, offset);

                // one 2nd neighbor in cube (cz-1,cy,cx) and offset
                atomN1N2[pos].N2[4]=BoxAndOffset_to_LinearIndex(nby, nbx, CzMinus, cy, cx, offset);

                // one 2nd neighbor in cube (cz+1,cy,cx) and offset
                atomN1N2[pos].N2[5]=BoxAndOffset_to_LinearIndex(nby, nbx, CzPlus, cy, cx, offset);
        }
}

// ***************************************************************************
// Find the first and second nearest neighbors of the vacancy site.
//
//      Parameters:
//              posV            The vacancy position (0..#atoms-1).
//              latP            Lattice parameters.
//              atomType        Pointer to the array with all lattice sites (type of atom).
//              posNN1          Pointer to the array where the 1st nearest neighbors list
//                              (position) will be saved.
//              NN1             Pointer to the array where the 1st nearest neighbors list
//                              (type of atom) will be saved.
//              posNN2          Pointer to the array where the 2nd nearest neighbors list
//                              (position) will be saved.
//              NN2             Pointer to the array where the 2nd nearest neighbors list
//                               (type of atom) will be saved.
//
//  Compute: posNN1, NN1, posNN2, NN2
//
```

```
void neighbors(int posV, LatticeParameters *latP, AtomType *atomType, int *posNN1,
        AtomType *NN1, int *posNN2, AtomType *NN2) {

        int i=0;
        int cz, cy, cx, offset, nbz, nby, nbx;
        int CzPlus,  CyPlus,  CxPlus;
        int CzMinus, CyMinus, CxMinus;

        nbz = latP->nbz;
        nby = latP->nby;
        nbx = latP->nbx;

        // compute the coordinates of the cube that includes the vacancy (cz, cy, cx)
        // and the position inside that cube (offset)

        LinearIndex_to_BoxAndOffset(posV, nby, nbx, &cz, &cy, &cx, &offset);

        CzMinus = cz-1; if(CzMinus<0) CzMinus=nbz-1; // (1)
        CyMinus = cy-1; if(CyMinus<0) CyMinus=nby-1; // (2)
        CxMinus = cx-1; if(CxMinus<0) CxMinus=nbx-1; // (3)
        CzPlus = cz+1; if(CzPlus==nbz) CzPlus=0;        // (4)
        CyPlus = cy+1; if(CyPlus==nby) CyPlus=0;        // (5)
        CxPlus = cx+1; if(CxPlus==nbx) CxPlus=0;        // (6)

        // Compute the position and type of the 12 1st neighbors **************

        // ********************************************************************
        if (offset == 0) {
                // 3 neighbors in same cube (cz,cy,cx) and offset={1,2,3}
                posNN1[0]=posV+1;
                posNN1[1]=posV+2;
                posNN1[2]=posV+3;

                // 2 neighbors in cube (cz,cy,cx-1) and offset={1,2}
                posNN1[3]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, cy, CxMinus, 1);
                posNN1[4]=posNN1[3]+1;

                // 2 neighbors in cube (cz-1,cy,cx) and offset={2,3}
                posNN1[5]=BoxAndOffset_to_LinearIndex(nby, nbx, CzMinus, cy, cx, 2);
                posNN1[6]=posNN1[5]+1;

                // 1 neighbor in cube (cz-1,cy,cx-1) and offset={2}
                posNN1[7]=BoxAndOffset_to_LinearIndex(nby, nbx, CzMinus, cy, CxMinus, 2);

                // 1 neighbor in cube (cz,cy-1,cx-1) and offset={1}
                posNN1[8]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, CyMinus, CxMinus, 1);

                // 2 neighbors in cube (cz,cy-1,cx) and offset={1,3}
                posNN1[9]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, CyMinus, cx, 1);
                posNN1[10]=posNN1[9]+2;

                // 1 neighbor in cube (cz-1,cy-1,cx) and offset={3}
                posNN1[11]=BoxAndOffset_to_LinearIndex(nby, nbx, CzMinus, CyMinus, cx, 3);
        }
```

```
// ********************************************************************
else if (offset == 1) {
		// 3 neighbors in same cube (cz,cy,cx) and offset={0,2,3}
		posNN1[0]=posV-1;
		posNN1[1]=posV+1;
		posNN1[2]=posV+2;

		// 2 neighbors in cube (cz,cy,cx+1) and offset={0,3}
		posNN1[3]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, cy, CxPlus, 0);
		posNN1[4]=posNN1[3]+3;

		// 2 neighbors in cube (cz-1,cy,cx) and offset={2,3}
		posNN1[5]=BoxAndOffset_to_LinearIndex(nby, nbx, CzMinus, cy, cx, 2);
		posNN1[6]=posNN1[5]+1;

		// 2 neighbors in cube (cz,cy+1,cx) and offset={0,2}
		posNN1[7]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, CyPlus, cx, 0);
		posNN1[8]=posNN1[7]+2;

		// 1 neighbor in cube (cz,cy+1,cx+1) and offset={0}
		posNN1[9]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, CyPlus, CxPlus, 0);

		// 1 neighbor in cube (cz-1,cy,cx+1) and offset={3}
		posNN1[10]=BoxAndOffset_to_LinearIndex(nby, nbx, CzMinus, cy, CxPlus, 3);

		// 1 neighbor in cube (cz-1,cy+1,cx) and offset={2}
		posNN1[11]=BoxAndOffset_to_LinearIndex(nby, nbx, CzMinus, CyPlus, cx, 2);
}

// ********************************************************************
else if (offset == 2) {
		// 3 neighbors in same cube (cz,cy,cx) and offset={0,1,3}
		posNN1[0]=posV-2;
		posNN1[1]=posV-1;
		posNN1[2]=posV+1;

		// 2 neighbors in cube (cz,cy,cx+1) and offset={0,3}
		posNN1[3]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, cy, CxPlus, 0);
		posNN1[4]=posNN1[3]+3;

		// 2 neighbors in cube (cz+1,cy,cx) and offset={0,1}
		posNN1[5]=BoxAndOffset_to_LinearIndex(nby, nbx, CzPlus, cy, cx, 0);
		posNN1[6]=posNN1[5]+1;

		// 2 neighbors in cube (cz,cy-1,cx) and offset={1,3}
		posNN1[7]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, CyMinus, cx, 1);
		posNN1[8]=posNN1[7]+2;

		// 1 neighbor in cube (cz+1,cy,cx+1) and offset={0}
		posNN1[9]=BoxAndOffset_to_LinearIndex(nby, nbx, CzPlus, cy, CxPlus, 0);

		// 1 neighbor in cube (cz+1,cy-1,cx) and offset={1}
		posNN1[10]=BoxAndOffset_to_LinearIndex(nby, nbx, CzPlus, CyMinus, cx, 1);

		// 1 neighbor in cube (cz,cy-1,cx+1) and offset={3}
		posNN1[11]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, CyMinus, CxPlus, 3);
}
```

```
// ********************************************************************
else if (offset == 3) {
        // 3 neighbors in same cube (cz,cy,cx) and offset={0,1,2}
        posNN1[0]=posV-3;
        posNN1[1]=posV-2;
        posNN1[2]=posV-1;

        // 2 neighbors in cube (cz+1,cy,cx) and offset={0,1}
        posNN1[3]=BoxAndOffset_to_LinearIndex(nby, nbx, CzPlus, cy, cx, 0);
        posNN1[4]=posNN1[3]+1;

        // 2 neighbors in cube (cz,cy+1,cx) and offset={0,2}
        posNN1[5]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, CyPlus, cx, 0);
        posNN1[6]=posNN1[5]+2;

        // 2 neighbors in cube (cz,cy,cx-1) and offset={1,2}
        posNN1[7]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, cy, CxMinus, 1);
        posNN1[8]=posNN1[7]+1;

        // 1 neighbor in cube (cz+1,cy+1,cx) and offset={0}
        posNN1[9]=BoxAndOffset_to_LinearIndex(nby, nbx, CzPlus, CyPlus, cx, 0);

        // 1 neighbor in cube (cz+1,cy,cx-1) and offset={1}
        posNN1[10]=BoxAndOffset_to_LinearIndex(nby, nbx, CzPlus, cy, CxMinus, 1);

        // 1 neighbor in cube (cz,cy+1,cx-1) and offset={2}
        posNN1[11]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, CyPlus, CxMinus, 2);
}

// save 1st neighbors type
for(i=0;i<NUM_1ST_NEIGHBORS;++i) {
        NN1[i]=atomType[posNN1[i]];
}

// Compute the position and type of the six 2nd neighbors ************

// ********************************************************************
// one 2nd neighbor in cube (cz,cy,cx-1) and offset
posNN2[0]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, cy, CxMinus, offset);
NN2[0]=atomType[posNN2[0]];

// one 2nd neighbor in cube (cz,cy,cx+1) and offset
posNN2[1]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, cy, CxPlus, offset);
NN2[1]=atomType[posNN2[1]];

// one 2nd neighbor in cube (cz,cy-1,cx) and offset
posNN2[2]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, CyMinus, cx, offset);
NN2[2]=atomType[posNN2[2]];

// one 2nd neighbor in cube (cz,cy+1,cx) and offset
posNN2[3]=BoxAndOffset_to_LinearIndex(nby, nbx, cz, CyPlus, cx, offset);
NN2[3]=atomType[posNN2[3]];

// one 2nd neighbor in cube (cz-1,cy,cx) and offset
posNN2[4]=BoxAndOffset_to_LinearIndex(nby, nbx, CzMinus, cy, cx, offset);
NN2[4]=atomType[posNN2[4]];
```

```
        // one 2nd neighbor in cube (cz+1,cy,cx) and offset
        posNN2[5]=BoxAndOffset_to_LinearIndex(nby, nbx, CzPlus, cy, cx, offset);
        NN2[5]=atomType[posNN2[5]];
}
```

xix. readFiles.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "main_includes.h"

// Date: 07 September 2012

// *****************************************************************
// Read and parse a VTK file.
//
//  Parameters:
//                  fName           VTK file name.
//                  numAtoms        Pointer to the variable where the number atoms read from file
//                                  will be stored.
//
//  Return:         Pointer to the array where the atoms (coordinates & type) read from
//                  file are stored (on success), NULL (on failure).
//
point3Dclassified *readVtkFile (char *fName, int *numAtoms) {

        char     line[MAX_LINE_SIZE], aux1[MAX_LINE_SIZE], aux2[MAX_LINE_SIZE];
        char    aux3[MAX_LINE_SIZE];
        FILE    *fp;
        booleano continueReading;
        int     nPoints, nAtoms, nLines, i, j, id, tmp, tmp2;
        int     *atomsPos;
        point3Dclassified *allPoints, *atoms;

        if((fp = fopen(fName,"r"))==NULL){
                printf("Error when trying to open VTK file %s\n", fName);
                return NULL;
        };

        // Read the file header ************************************************
        //
        // # vtk DataFile Version 2.0
        // Generated by Alfredo de Moura and Antonio Esteves
        // ASCII
        // DATASET POLYDATA

        continueReading = readLine(fp, line);
        if( (continueReading != TRUE) || (strcmp(line,"# vtk DataFile Version 2.0")!=0) )
                return NULL;

        continueReading = readLine(fp, line);
        if (continueReading != TRUE)
                return NULL;

        continueReading = readLine(fp, line);
```

```
if( (continueReading != TRUE) || (strcmp(line,"ASCII")!=0) )
        return NULL;

continueReading = readLine(fp, line);
if( (continueReading != TRUE) || (strcmp(line,"DATASET POLYDATA")!=0) )
        return NULL;

// Read points ************************************************************
//
// POINTS <nPoints> <"float">
// <X_1> <Y_1> <Z_1>
// ...
// <X_nPoints> <Y_nPoints> <Z_nPoints>

continueReading = readLine(fp, line);
if (continueReading != TRUE)
        return NULL;
sscanf(line,"%s %d %s", aux1, &nPoints, aux2);
if( (strcmp(aux1,"POINTS")!=0) || (strcmp(aux2,"float")!=0) )
        return NULL;

if ((allPoints = (point3Dclassified *)malloc(nPoints*sizeof(point3Dclassified))) == NULL) {
        printf("Error on memory allocation for 'allPoints'\n");
        return NULL;
}

for(i=0;i<nPoints;++i) {
        continueReading = readLine(fp, line);
        // printf("LINE: %s\n", line);
                                        // DEBUG ONLY !!!!!!!!!
        sscanf(line,"%lf %lf %lf ", &allPoints[i].x, &allPoints[i].y, &allPoints[i].z);
}

// Read atoms (vtk vertices) ***********************************************
//
// VERTICES <nAtoms> <nAtoms*2>
// 1 <ID_1>
// ...
// 1 <ID_nAtoms>

continueReading = readLine(fp, line);
if (continueReading != TRUE) {
        free(allPoints);
        return NULL;
}
sscanf(line,"%s %d %d", aux1, &nAtoms, &i);
if (strcmp(aux1,"VERTICES")!=0) {
        free(allPoints);
        return NULL;
}
*numAtoms = nAtoms;
if ((atoms = (point3Dclassified *)malloc(nAtoms*sizeof(point3Dclassified))) == NULL) {
        printf("Error on memory allocation for 'atoms'\n");
        free(allPoints);
        return NULL;
}
if ((atomsPos = (int *)malloc(nAtoms*sizeof(int))) == NULL) {
        printf("Error on memory allocation for 'atomsPos'\n");
```

```
		free(atoms);
		free(allPoints);
		return NULL;
	}
	for(i=0;i<nAtoms;++i) {
		continueReading = readLine(fp, line);
		if (continueReading != TRUE) {
			free(atomsPos);
			free(atoms);
			free(allPoints);
			return NULL;
		}
		sscanf(line,"%d %d", &tmp, &id);


		atomsPos[i] = id; // save atoms' postion for later usage in lookup table
		atoms[i].x  = allPoints[id].x; // copy coordinates of the 'id' atom
		atoms[i].y  = allPoints[id].y;
		atoms[i].z  = allPoints[id].z;
	}

	continueReading = readLine(fp, line);
	if (continueReading != TRUE) {
		free(atomsPos);
		free(atoms);
		free(allPoints);
		return NULL;
	}
	sscanf(line,"%s", aux1);

	if (strcmp(aux1,"LINES")==0) {
		// Read lines (optionaly) ***************************************************
		//
		// LINES <nLines> <nLines*3>
		// 2 <ID_1_a> <ID_1_b>
		// ...
		// 2 <ID_nLines_a> <ID_nLines_b>

		sscanf(line,"%s %d %d", aux1, &nLines, &i);
		for(i=0;i<nLines;++i) {
			continueReading = readLine(fp, line);
			if (continueReading != TRUE) {
				free(atomsPos);
				free(atoms);
				free(allPoints);
				return NULL;
			}
		}

		continueReading = readLine(fp, line);
		if (continueReading != TRUE) {
			free(atomsPos);
			free(atoms);
			free(allPoints);
			return NULL;
		}
		sscanf(line,"%s", aux1);
	}
```

```
if (strcmp(aux1,"POINT_DATA")==0) { //--------------------------------
        // Read atoms' type (vtk footer & lookup table ******************************
        //
        // POINT_DATA <nPoints>
        // SCALARS atom_type int 1
        // LOOKUP_TABLE default
        // <V_1>  <V_2>  ...  <V_nPoints>

        sscanf(line,"%s %d", aux1, &tmp);
        if(tmp != nPoints) {
                printf("Error: lookup table with wrong size!\n");
                free(atomsPos);
                free(atoms);
                free(allPoints);
                return NULL;
        }

        continueReading = readLine(fp, line);
        if (continueReading != TRUE) {
                free(atomsPos);
                free(atoms);
                free(allPoints);
                return NULL;
        }
        sscanf(line,"%s %s %s %d", aux1, aux2, aux3, &tmp);
        if ( (strcmp(aux1,"SCALARS")!=0) || (strcmp(aux2,"atom_type")!=0) || (strcmp(aux3,"int")!=0) ||
                        (tmp!=1) ) {
                printf("Error: SCALARS line with wrong format!\n");
                free(atomsPos);
                free(atoms);
                free(allPoints);
                return NULL;
        }
}
else { // Error ------------------------------------------------------------
        free(atomsPos);
        free(atoms);
        free(allPoints);
        return NULL;
}

continueReading = readLine(fp, line);
if (continueReading != TRUE) {
        free(atomsPos);
        free(atoms);
        free(allPoints);
        return NULL;
}
sscanf(line,"%s %s", aux1, aux2);
if ( (strcmp(aux1,"LOOKUP_TABLE")!=0) || (strcmp(aux2,"default")!=0) ) {
        printf("Error: LOOKUP_TABLE line with wrong format!\n");
        free(atomsPos);
        free(atoms);
        free(allPoints);
        return NULL;
}
```

```
	for(i=0,j=0;i<nPoints;++i) {
		fscanf(fp,"%d  ",&tmp);
		atoms[j].type	= (AtomType)tmp;
		atoms[j].classId = UNCLASSIFIED;
		++j;
	}

	fclose(fp);
	free(atomsPos);
	free(allPoints);
	return atoms;
}
```

xx. utils.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "main_includes.h"

// Date: 07 July 2012

// Convert time from format 'time_t' to DAYS:HOURS:MINUTES:SECONDS.
//
//	Parameters:
//		ti	Time to be converted.
//		to	Pointer to the structure where the converted time will be stored.
//
void convertTime(double ti, TimeInterval *to) {
	double	remainder;

	to->days   = (unsigned long long)(floor(ti/86400.0));
	remainder   = ti-(double)(to->days*86400);
	to->hours= (int)(floor(remainder/3600.0));
	remainder  -= (double)(to->hours*3600);
	to->minutes  = (int)(floor(remainder/60.0));
	to->seconds  = remainder - (double)(to->minutes*60);
}

// Add two times in format DAYS:HOURS:MINUTES:SECONDS.
//
//	Parameters:
//		to	Pointer to the structure where the added times will be stored.
//		t1, t2	Times to be added.
//
//	Compute:
//			The addition of 2 times in format DAYS:HOURS:MINUTES:SECONDS.
//
void addTimes(TimeInterval *to, TimeInterval t1, TimeInterval t2) {
	to->days  = 0ULL;
	to->hours   = 0;
	to->minutes = 0;
	to->seconds = 0.0;

	to->seconds = t1.seconds + t2.seconds;
```

```
	if(to->seconds>=60.0) {
		to->seconds -= 60.0;
		++to->minutes;
	}

	to->minutes += t1.minutes + t2.minutes;
	if(to->minutes>=60) {
		to->minutes -= 60;
		++to->hours;
	}
	to->hours += t1.hours + t2.hours;
	if(to->hours>=24) {
		to->hours -= 24;
		++to->days;
	}
	to->days += t1.days + t2.days;
}

// Round a double number to the closest integer.
//
int roundDouble2integer (double num) {
	double	floorNum;
	int	numInt;

	floorNum = floor(num);
	if((num-floorNum)>=0.5)
		numInt = (int)num+1;
	else
		numInt = (int)num;
	return numInt;
}

// ********************************************************************
// Generate 'numRand' double random numbers in the interval [min:max].
//
//	Parameters:
//		outRand		Pointer to the array where random numbers will be stored.
//		min		Minimum value of the random numbers to generate.
//		max		Maximum value of the random numbers to generate.
//		numRand		Total of random numbers to generate.
//
void genRandDoubleNoSeed (double *outRand, double min, double max, int numRand) {
	double x;
	int	i;

	// Generate random numbers uniformly distributed in range [min : max]
	for (i = 0; i < numRand; i++) {
		x = (double) rand() / (double) RAND_MAX;
		x = min + (max-min)*x;
		*(outRand+i)=x;
	}
}
```

```
// ****************************************************************
// Generate 'numRand' double random numbers in the interval [min:max].
//
//		Parameters:
//			outRand		Pointer to the array where random numbers will be stored.
//			min		Minimum value of the random numbers to generate.
//			max		Maximum value of the random numbers to generate.
//			numRand		Total of random numbers to generate.
//
void genRandDouble (double *outRand, double min, double max, int numRand) {
	int	seed;
	double x;
	int	i;

	// Seed random number generator
	seed = time(NULL);
	srand(seed);



	// Generate random numbers uniformly distributed in range [min : max]
	for (i = 0; i < numRand; i++) {
		x = (double) rand() / (double) RAND_MAX;
		x = min + (max-min)*x;
		*(outRand+i)=x;
	}
}

// ****************************************************************
// Generate 'numRand' integer random numbers in the interval [min:max].
//
//		Parameters:
//			outRand		Pointer to the array where random numbers will be stored.
//			min		Minimum value of the random numbers to generate.
//			max		Maximum value of the random numbers to generate.
//			numRand		Total of random numbers to generate.
//
void genRandInteger (int *outRand, int min, int max, int numRand) {
	int	seed;
	double x;
	int	i;

	// Seed random number generator
	seed = time(NULL);
	srand(seed);

	// Generate random numbers uniformly distributed in range [min : max]
	for (i = 0; i < numRand; i++) {
		x = (double) rand() / (double) RAND_MAX;
		x = min + (max-min)*x;
		*(outRand+i)=roundDouble2integer(x);
	}
}
```

```
// ****************************************************************
// Generate 'numRand' integer random numbers in the interval [min:max] without repetitions.
//
//          Parameters:
//                    outRand          Pointer to the array where random numbers will be stored.
//                    min              Minimum value of the random numbers to generate.
//                    max              Maximum value of the random numbers to generate.
//                    numRand          Total of random numbers to generate.
//
void genNonRepeatedRandIntege r(int *outRand, int min, int max, int numRand) {
          int       seed;
          double    x;
          int        i, j;
          booleano found;

          // Seed random number generator
          seed = time(NULL);
          srand(seed);

          // Generate random numbers uniformly distributed in range [min : max]
          // without repetitions
          for (i = 0; i < numRand; i++) {
                    x = (double) rand() / (double) RAND_MAX;
                    x = min + (max-min)*x;
                    *(outRand+i)=roundDouble2integer(x);
                    found = FALSE;

                    for(j=0;j<i;++j) {
                              if (*(outRand+j) == *(outRand+i)) {
                                        found = TRUE;
                              }
                    }
                    if (found == TRUE) {
                              --i;
                    }
          }
}

// ****************************************************************
// Generate an integer random number, in the interval [min:max],
//  that is different from any number in the 'forbiddenRand' array.
//
//          Parameters:
//                    forbiddenRand    Pointer to the array containing not allowed numbers.
//                    forbiddenSize    Size of the array of not allowed numbers.
//                    min              Minimum value of the random numbers to generate.
//                    max              Maximum value of the random numbers to generate.
//
//          Return:
//                    The new generated random number.
//
int genNotForbiddenRandInteger (int *forbiddenRand, int forbiddenSize, int min, int max) {
          double   x;
          int        seed, j, newRand;
          booleano found;

          // Seed random number generator
          seed = time(NULL);
```

```
	srand(seed);

	// Generate a random number in range [min : max] that is
	// different from any number in 'forbiddenRand' array
	do {
		x = (double) rand() / (double) RAND_MAX;
		x = min + (max-min)*x;
		newRand = roundDouble2integer(x);
		found   = FALSE;

		for(j=0;j<forbiddenSize;++j) {
			if (*(forbiddenRand+j) == newRand) {
				found = TRUE;
			}
		}
	} while (found == TRUE);

	return newRand;
}
```

xxi. writeFiles.c

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <time.h>
#include <stdlib.h>
#include "main_includes.h"

// Date: 11 October 2012

// ********************************************************************
// Define the simulation step where each snapshot will be saved.
//
//	Parameters:
//		simParam		Simulation parameters
//		snapshots		Array with the step where each snapshost must be saved
//
void snapshotsNumber (SimulationParameters *simParam,unsigned long long *snapshots) {
	unsigned long long i;
	double		aux;

	snapshots[0] = 0ULL;
	for (i=1ULL;i<=simParam->snap;++i) {
		if(i==simParam->snap) // to be sure that last snapshot = last simulation step
			snapshots[i]=simParam->mcs-1ULL;
		else {
			aux = (double)i*((double)simParam->mcs/(double)simParam->snap);
			snapshots[i]=(unsigned long long)floor(aux)-1ULL;
			}
	}

	for (i=1;i<=simParam->snap;++i)
		printf("Snapshot[%lld] = %lld\n",i,snapshots[i]);
}
```

```
// ****************************************************************
// Write atoms' coordinates and type to a XYZ file.
//
//  Parameters:
//                      simP        Pointer to the simulation parameters.
//                      numSnap  Number of the snapshot to write to file.
//                      atomP      Pointer to the array with atoms's coordinates.
//                      atomT      Pointer to the array with atoms's type.
//                      nAtoms   Number of atoms in the lattice to write in file.
//
void writeXYZfile (SimulationParameters *simP, int numSnap, point3d atomP[],
        AtomType atomT[], int nAtoms) {

        char fName[200], str[20];
        FILE *fp;
        int  i;

        strcpy(fName,simP->coordFile); // base file name
        strcat(fName,"_");                      // append "_"
        sprintf(str,"%d",numSnap);
        strcat(fName,str);                      // append the number of the snapshot
        strcat(fName,".xyz");                    // append the file extension ".xyz"

        i=0;
        do {
                if((fp = fopen(fName,"w"))==NULL){
                        printf("\nError when tryng to open XYZ file for writing (%d)\n",i);
                };
        } while (fp == NULL);

        fprintf(fp,"%d\n",nAtoms);
        fprintf(fp,"%s\n","Atoms");

        // write "ATOM_TYPE  X_COORD  Y_COORD  Z_COORD" for each atom
        for(i=0;i<nAtoms;++i) {
                fprintf(fp,"%d %f %f %f\n",atomT[i],atomP[i].x,atomP[i].y,atomP[i].z);
        }

        fclose(fp);
}

// ****************************************************************
// Write atoms' coordinates and type to a PDB (Protein Data Bank) file.
//
//  Parameters:
//                      simP        Pointer to the simulation parameters.
//                      numSnap  Number of the snapshot to write to file.
//                      atomP      Pointer to the array with atoms's coordinates.
//                      atomT      Pointer to the array with atoms's type.
//                      nAtoms   Number of atoms in the lattice to write in file.
//
void writePDBfile (SimulationParameters *simP, int numSnap, point3d atomP[],
        AtomType atomT[], int nAtoms) {

        char fName[200], str[20], SC[3]="Sc", AL[3]="Al";
        FILE *fp;
        int  i,j;
```

```
        strcpy(fName,simP->coordFile);          // base file name
        strcat(fName,"_");                      // append "_"
        sprintf(str,"%d",numSnap);
        strcat(fName,str);                      // append the number of the snapshot
        strcat(fName,".pdb");                   // append the file extension ".pdb"

        i=0;
        do {
                if((fp = fopen(fName,"w"))==NULL){
                        printf("\nError when tryng to open PDB file for writing (%d)\n",i);
                };
        } while (fp == NULL);

        // write "ATOM  ATOM_NUMBER ATOM_TYPE 1 X_COORD  Y_COORD  Z_COORD" for each atom
        for(i=0, j=1;i<nAtoms;++i) {
                if(atomT[i]==Al) {
                        fprintf(fp,"ATOM    %d  %s                 1          %.3f %.3f %.3f  1.00  0.00\n", j,
                                AL,atomP[i].x,atomP[i].y,atomP[i].z);
                        ++j;
                }

                else if(atomT[i]==Sc) {
                        fprintf(fp,"ATOM    %d  %s                 1          %.3f %.3f %.3f  1.00  0.00\n", j, SC,
                                atomP[i].x,atomP[i].y,atomP[i].z);
                        ++j;
                }
        }

        fprintf(fp,"END\n");
        fclose(fp);
}

// ***********************************************************************
// For a selected type of atoms (Al, Sc, Vacancy), write their coordinates
//  and type to a VTK file. The lattice cubes are drawn too.
//
//  Parameters:
//                  simP        Pointer to the simulation parameters.
//                  latP        Pointer to the lattice parameters.
//                  numSnap  Number of the snapshot to write to file.
//                  atomP      Pointer to the array with atoms's coordinates.
//                  atomT      Pointer to the array with atoms's type.
//
void writeVTKfileWithLines (SimulationParameters *simP, LatticeParameters *latP, int numSnap,
        point3d atomP[], AtomType atomT[]) {

        char        fName[200], str[20];
        FILE       *fp;
        int         i, j, cx, cy, cz, nbx, nby, nbz, nAtoms, nPoints, nVertexes, nlines;
        int         firstPt, secondPt, initPosition;
        double   lc;
        booleano wasSelected[3]={FALSE,FALSE,FALSE}; // to indicate if (Al, Sc or V were selected)

        lc          = latP->lc;
        nbx         = latP->nbx;
        nby         = latP->nby;
        nbz         = latP->nbz;
        nAtoms   = latP->num_Al_Sites + latP->num_Sc_Sites + latP->num_V_Sites;
```

```
nPoints   = nAtoms+(nbz+1)*(nby+1)+nbx*nbz+(nby+1)*nbx;
strcpy(fName,simP->coordFile); // base file name

switch(simP->selectedType) {
        case AL:
                wasSelected[AL_POS]=TRUE;
                nVertexes = latP->num_Al_Sites;
                strcat(fName,"_Al_");                           // append "_Al_" to file name
                break;
        case SC:
                wasSelected[SC_POS]=TRUE;
                nVertexes = latP->num_Sc_Sites;
                strcat(fName,"_Sc_");                           // append "_Sc_" to file name
                break;
        case VACANCY:
                wasSelected[V_POS]=TRUE;
                nVertexes = latP->num_V_Sites;
                strcat(fName,"_V_");                            // append "_V_" to file name
                break;

        case ALSC:
                wasSelected[AL_POS]=TRUE;
                wasSelected[SC_POS]=TRUE;
                nVertexes = latP->num_Al_Sites+latP->num_Sc_Sites;
                strcat(fName,"_AlSc_");                         // append "_AlSc_" to file name
                break;
        case ALV:
                wasSelected[AL_POS]=TRUE;
                wasSelected[V_POS] =TRUE;
                nVertexes = latP->num_Al_Sites+latP->num_V_Sites;
                strcat(fName,"_AlV_");                          // append "_AlV_" to file name
                break;
        case SCV:
                wasSelected[SC_POS]=TRUE;
                wasSelected[V_POS] =TRUE;
                nVertexes = latP->num_Sc_Sites+latP->num_V_Sites;
                strcat(fName,"_ScV_");                          // append "_ScV_" to file name
                break;
        case ALL:
                wasSelected[AL_POS]=TRUE;
                wasSelected[SC_POS]=TRUE;
                wasSelected[V_POS] =TRUE;
                nVertexes = nAtoms;
                strcat(fName,"_All_");                          // append "_All_" to file name
                break;
        default:
                printf("Incorrect type(s) of atoms selected to write in VTK file.\n");
                return;
}

sprintf(str,"%d",numSnap);
strcat(fName,str);                      // append the number of the snapshot to file name
strcat(fName,".vtk");                   // append the file extension ".vtk" to file name


i=0;
```

```
do {
	if((fp = fopen(fName,"w"))==NULL){
		printf("\nError when tryng to open VTK file for writing (%d)\n",i);
	};
} while (fp == NULL);

//*****************************************************************
// Write the file header

fprintf(fp,"# vtk DataFile Version 2.0\n");
fprintf(fp,"Generated by Alfredo de Moura and Antonio Esteves\n");
fprintf(fp,"ASCII\n");
fprintf(fp,"DATASET POLYDATA\n");
fprintf(fp,"POINTS %d float \n", nPoints);

//***************************************************************** POINTS
for(i=0;i<nAtoms;++i) {
	fprintf(fp,"%.3f %.3f %.3f \n",atomP[i].x,atomP[i].y,atomP[i].z);
}

//*****************************************************************
// Extra points to complete all lattice cube vertexes

// RIGHT FACE of the lattice [initPosition = nAtoms]
for(cz=0;cz<=nbz;++cz) {
	for(cy=0;cy<=nby;++cy) {
		fprintf(fp,"%.3f %.3f %.3f \n",nbx*lc,cy*lc,cz*lc);
	}
}
// BACK FACE of the lattice [initPosition = nAtoms + (nbz+1)*(nby+1)]
for(cz=0;cz<nbz;++cz) {
	for(cx=0;cx<nbx;++cx) {
		fprintf(fp,"%.3f %.3f %.3f \n",cx*lc,nby*lc,cz*lc);
	}
}
// TOP FACE of the lattice [initPosition = nAtoms+(nbz+1)*(nby+1) + nbx*nbz]
for(cy=0;cy<=nby;++cy) {
	for(cx=0;cx<nbx;++cx) {
		fprintf(fp,"%.3f %.3f %.3f \n",cx*lc,cy*lc,nbz*lc);
	}
}

//***************************************************************** VERTICES
fprintf(fp,"VERTICES %d %d \n", nVertexes, 2*nVertexes);

for(i=0, j=0;i<nAtoms;++i) {

	if( ((atomT[i]==Al) && (wasSelected[AL_POS]==TRUE)) || ((atomT[i]==Sc) &&
			(wasSelected[SC_POS]==TRUE)) || ((atomT[i]==Vacancy) &&
			(wasSelected[V_POS]==TRUE)) ) {
		fprintf(fp,"1 %d \n",i);
		++j;
	}
}

//***************************************************************** LINES
// nlines = (nbz+1)*(nby+1)*nbx + 2*nbz*nby*nbx + (nbz+1+nbx)*nby + nbz*(nby+1) + nbz*nbx;
```

```
nlines = 3*nbz*nby*nbx + 2*(nbz*nby+nbz*nbx+nby*nbx) + (nbz+nby+nbx);

fprintf(fp,"LINES %d %d\n", nlines, 3*nlines);

// lines on X direction, on the [nbz][nby][nbx] cubes of the lattice
for(cz=0;cz<nbz;++cz) {
	for(cy=0;cy<nby;++cy) {
		for(cx=0;cx<nbx;++cx) {
			firstPt  = FCC_ATOMS_CELL*(nbx*(cz*nby+cy)+cx);
			if(cx==(nbx-1))
				secondPt = nAtoms+cz*(nby+1)+cy ; // extra point in RIGHT face
			else
				secondPt = firstPt+FCC_ATOMS_CELL;
			fprintf(fp,"2 %d %d\n",firstPt,secondPt);
		}
	}
}

// lines on X direction in the BACK FACE of the lattice
initPosition = nAtoms+(nbz+1)*(nby+1);
for(cz=0;cz<nbz;++cz) {
	for(cx=0;cx<nbx;++cx) {
		firstPt  = initPosition+cz*nbx+cx;
		if(cx==(nbx-1))
		    secondPt = nAtoms+cz*(nby+1)+nby ; // extra point in RIGHT face, back edge
		else
		    secondPt = firstPt+1;
		fprintf(fp,"2 %d %d\n",firstPt,secondPt);
	}
}

// lines on X direction in the TOP FACE of the lattice
initPosition = nAtoms+(nbz+1)*(nby+1)+nbx*nbz;
for(cy=0;cy<=nby;++cy) {
	for(cx=0;cx<nbx;++cx) {
		firstPt  = initPosition+cy*nbx+cx;
		if(cx==(nbx-1))
			secondPt = nAtoms+(nby+1)*nbz+cy; // extra point in RIGHT face, top edge
		else
			secondPt = firstPt+1;
		fprintf(fp,"2 %d %d\n",firstPt,secondPt);
	}
}

// lines on Y direction, on the [nbz][nby][nbx] cubes of the lattice
initPosition = nAtoms + (nbz+1)*(nby+1);
for(cz=0;cz<nbz;++cz) {
	for(cx=0;cx<nbx;++cx) {
		for(cy=0;cy<nby;++cy) {
			firstPt  = FCC_ATOMS_CELL*(nbx*(cz*nby+cy)+cx);
			if(cy==(nby-1))
				secondPt = initPosition+cz*nbx+cx; // extra point in BACK face
			else
				secondPt = firstPt+FCC_ATOMS_CELL*nbx;
			fprintf(fp,"2 %d %d\n",firstPt,secondPt);
		}
	}
}
```

```
// lines on Y direction in the RIGHT FACE of the lattice
initPosition = nAtoms;
for(cz=0;cz<=nbz;++cz) {
	for(cy=0;cy<nby;++cy) {
		firstPt  = initPosition+cz*(nby+1)+cy;
		fprintf(fp,"2 %d %d\n",firstPt,firstPt+1);
	}
}

// lines on Y direction in the TOP FACE of the lattice
initPosition = nAtoms+(nbz+1)*(nby+1) + nbx*nbz;
for(cy=0;cy<nby;++cy) {
	for(cx=0;cx<nbx;++cx) {
		firstPt  = initPosition+cy*nbx+cx;
		fprintf(fp,"2 %d %d\n",firstPt,firstPt+nbx);
	}
}

// lines on Z direction, on the [nbz][nby][nbx] cubes of the lattice
initPosition = nAtoms+(nbz+1)*(nby+1) + nbx*nbz;
for(cz=0;cz<nbz;++cz) {
	for(cy=0;cy<nby;++cy) {
		for(cx=0;cx<nbx;++cx) {
			firstPt  = FCC_ATOMS_CELL*(cx+nbx*(cy+cz*nby));
			if(cz==(nbz-1))
				secondPt = initPosition+cy*nbx+cx;// extra point in TOP face
			else
				secondPt = firstPt+FCC_ATOMS_CELL*nby*nbx;
			fprintf(fp,"2 %d %d\n",firstPt,secondPt);
		}
	}
}

// lines on Z direction in the RIGHT FACE of the lattice
initPosition = nAtoms;
for(cz=0;cz<nbz;++cz) {
	for(cy=0;cy<=nby;++cy) {
		firstPt  = initPosition+cz*(nby+1)+cy;
		secondPt = firstPt+nby+1;
		fprintf(fp,"2 %d %d\n",firstPt,secondPt);
	}
}

// lines on Z direction in the BACK FACE of the lattice
initPosition = nAtoms + (nbz+1)*(nby+1);
for(cz=0;cz<nbz;++cz) {
	for(cx=0;cx<nbx;++cx) {
			firstPt  = initPosition+cz*nbx+cx;
			if(cz==(nbz-1))
				// extra point in TOP face, back edge
				secondPt = initPosition+nbz*(nbx+nby)+cx;
			else
				secondPt = firstPt+nbx;
		fprintf(fp,"2 %d %d\n",firstPt,secondPt);
	}
}
```

```
	//*****************************************************************
	fprintf(fp,"POINT_DATA %d \n", nPoints);
	fprintf(fp,"SCALARS atom_type int 1 \n");
	fprintf(fp,"LOOKUP_TABLE default \n");

	for(i=0;i<nAtoms;++i) {
		fprintf(fp,"%d  ",atomT[i]);
	}
	// write "0" on the LOOKUP_TABLE positions relative to extra points
	for(;i<nPoints;++i) {
		fprintf(fp,"%d  ",0);
	}
	fprintf(fp,"\n");

	fclose(fp);
}


// ********************************************************************
// For a selected type of atoms (Al, Sc, Vacancy), write their coordinates
//  and type to a VTK file. the lattice cubes are NOT drawn.
//
//  Parameters:
//		simP	Pointer to the simulation parameters.
//		latP	Pointer to the lattice parameters.
//		numSnap	Number of the snapshot to write to file.
//		atomP	Pointer to the array with atoms's coordinates.
//		atomT	Pointer to the array with atoms's type.
//
void writeVTKfileNoLines (SimulationParameters *simP, LatticeParameters *latP,
	int numSnap, point3d atomP[], AtomType atomT[]) {

	char fName[200], str[20];
	FILE *fp;
	int  i, na_wr, na2_wr, nAtoms;
	booleano wasSelected[3]={FALSE,FALSE,FALSE}; // to indicate if (Al, Sc or V were selected)

	nAtoms = latP->num_Al_Sites+latP->num_Sc_Sites+latP->num_V_Sites;

	strcpy(fName,simP->coordFile);				// base file name

	switch(simP->selectedType) {
		case AL:
			wasSelected[AL_POS]=TRUE;
			na_wr  = latP->num_Al_Sites;
			strcat(fName,"_Al_");				// append "_Al_" to file name
			break;
		case SC:
			wasSelected[SC_POS]=TRUE;
			na_wr  = latP->num_Sc_Sites;
			strcat(fName,"_Sc_");				// append "_Sc_" to file name
			break;
		case VACANCY:
			wasSelected[V_POS]=TRUE;
			na_wr  = latP->num_V_Sites;
			strcat(fName,"_V_");				// append "_V_" to file name
			break;
```

```
		case ALSC:
			wasSelected[AL_POS]=TRUE;
			wasSelected[SC_POS]=TRUE;
			na_wr  = latP->num_Al_Sites+latP->num_Sc_Sites;
			strcat(fName,"_AlSc_");				// append "_AlSc_" to file name
			break;
		case ALV:
			wasSelected[AL_POS]=TRUE;
			wasSelected[V_POS] =TRUE;
			na_wr  = latP->num_Al_Sites+latP->num_V_Sites;
			strcat(fName,"_AlV_");				// append "_AlV_" to file name
			break;
		case SCV:
			wasSelected[SC_POS]=TRUE;
			wasSelected[V_POS] =TRUE;
			na_wr  = latP->num_Sc_Sites+latP->num_V_Sites;
			strcat(fName,"_ScV_");				// append "_ScV_" to file name
			break;
		case ALL:
			wasSelected[AL_POS]=TRUE;
			wasSelected[SC_POS]=TRUE;
			wasSelected[V_POS] =TRUE;
			na_wr  = nAtoms;
			strcat(fName,"_All_");				// append "_All_" to file name
			break;
		default:
			printf("Incorrect type(s) of atoms selected to write in VTK file.\n");
			return;
	}

	sprintf(str,"%d",numSnap);
	strcat(fName,str);			// append the number of the snapshot to file name
	strcat(fName,".vtk");			// append the file extension ".vtk" to file name

	na2_wr = 2*na_wr;

	i=0;
	do {
		if((fp = fopen(fName,"w"))==NULL){
			printf("\nError when tryng to open VTK file for writing (%d)\n",i);
		};
	} while (fp == NULL);

	// Write the file header

	fprintf(fp,"# vtk DataFile Version 2.0\n");
	fprintf(fp,"Generated by Alfredo de Moura and Antonio Esteves\n");
	fprintf(fp,"ASCII\n");
	fprintf(fp,"DATASET POLYDATA\n");
	fprintf(fp,"POINTS %d float \n", na_wr);

	for(i=0;i<nAtoms;++i) {
		if( ((atomT[i]==Al) && (wasSelected[AL_POS]==TRUE)) || ((atomT[i]==Sc) &&
				(wasSelected[SC_POS]==TRUE)) || ((atomT[i]==Vacancy) &&
				(wasSelected[V_POS]==TRUE)) ) {
			fprintf(fp,"%.3f %.3f %.3f \n",atomP[i].x,atomP[i].y,atomP[i].z);
		}
	}
```

```
	fprintf(fp,"VERTICES %d %d \n", na_wr, na2_wr);

	for(i=0;i<na_wr;++i) {
		fprintf(fp,"1 %d \n",i);
	}

	fprintf(fp,"POINT_DATA %d \n", na_wr);
	fprintf(fp,"SCALARS atom_type int 1 \n");
	fprintf(fp,"LOOKUP_TABLE default \n");

	for(i=0;i<nAtoms;++i) {
		if( ((atomT[i]==Al) && (wasSelected[AL_POS]==TRUE)) || ((atomT[i]==Sc) &&
				(wasSelected[SC_POS]==TRUE)) || ((atomT[i]==Vacancy) &&
				(wasSelected[V_POS]==TRUE)) ) {
			fprintf(fp,"%d  ",atomT[i]);
		}
	}
	fprintf(fp,"\n");

	fclose(fp);
}

// ********************************************************************
// Write atoms' coordinates and type to a VTK file (simplified version).
//
//  Parameters:
//		simP	 Pointer to the simulation parameters.
//		numSnap  Number of the snapshot to write to file.
//		atomP	Pointer to the array with atoms's coordinates.
//		atomT	Pointer to the array with atoms's type.
//		nAtoms   Number of atoms in the lattice to write in file.
//		nVacancies Number of vacancies in the lattice, which will not be written to file.
//
void writeVTKfileSimple (SimulationParameters *simP, int numSnap, point3d atomP[],
	AtomType atomT[], int nAtoms, int nVacancies) {

	char fName[200], str[20];
	FILE *fp;
	int  i, na_wr, na2_wr;

	na_wr  = nAtoms-nVacancies;
	na2_wr = 2*na_wr;

	strcpy(fName,simP->coordFile);		// base file name
	strcat(fName,"_");			// append "_"
	sprintf(str,"%d",numSnap);
	strcat(fName,str);			// append the number of the snapshot
	strcat(fName,".vtk");			// append the file extension ".vtk"

	i=0;
	do {
		if((fp = fopen(fName,"w"))==NULL){
			printf("\nError when tryng to open VTK file for writing (%d)\n",i);
		};
	} while (fp == NULL);
```

```
	// Write the file header

	fprintf(fp,"# vtk DataFile Version 2.0\n");
	fprintf(fp,"Generated by Alfredo de Moura and Antonio Esteves\n");
	fprintf(fp,"ASCII\n");
	fprintf(fp,"DATASET POLYDATA\n");
	fprintf(fp,"POINTS %d float \n", na_wr);

	for(i=0;i<nAtoms;++i) {
		if((atomT[i]==Al) || (atomT[i]==Sc)) {
			fprintf(fp,"%.3f %.3f %.3f \n",atomP[i].x,atomP[i].y,atomP[i].z);
		}
	}

	fprintf(fp,"VERTICES %d %d \n", na_wr, na2_wr);

	for(i=0;i<na_wr;++i) {
		fprintf(fp,"1 %d \n",i);
	}

	fprintf(fp,"POINT_DATA %d \n", na_wr);
	fprintf(fp,"SCALARS atom_type int 1 \n");
	fprintf(fp,"LOOKUP_TABLE default \n");

	for(i=0;i<nAtoms;++i) {
		if((atomT[i]==Al) || (atomT[i]==Sc)) {
			fprintf(fp,"%d  ",atomT[i]);
		}
	}
	fprintf(fp,"\n");

	fclose(fp);
}

// ********************************************************************
// Write a report of the simulation in a file.
//
//  Parameters:
//		simP		Pointer to the simulation parameters.
//		energyP		Pointer to the energy parameters.
//		latticeP	Pointer to the lattice parameters.
//		timeRun		Pointer to simulation execution time (D..D:HH:mm:ss).
//		timeSim		Pointer to the simulated time (D..D:HH:mm:ss).
//		snapTime	Pointer to the array of snapshots' corrected simulation
//				time (D..D:HH:mm:ss).
//
void writeSimulationReport (SimulationParameters *simP, EnergyParameters *energyP,
	LatticeParameters *latticeP, TimeInterval *timeRun, TimeInterval *timeSim,
	TimeInterval *snapTime) {

	char	fName[200], str[20];
	FILE	*fp;
	int	 i;
	time_t	 curtime;
	struct tm *loctime;
	char	data[100];

	// Get the current time
```

```
curtime = time (NULL);
// Convert it to local time representation
loctime = localtime (&curtime);
// convert date and time to the standard format
sprintf(data,"%s",asctime (loctime));

strcpy(fName,simP->coordFile); // base file name
strcat(fName,"_report.txt");   // append "_report.txt"

i=0;
do {
        if((fp = fopen(fName,"w"))==NULL){
                printf("\nError when tryng to open Report file for writing (%d)\n",i);
        };
} while (fp == NULL);

// Report Header

fprintf(fp,"*********************************************************************************\n");
fprintf(fp,"Simulation Report\n");
fprintf(fp,"Made in: %s\n",data);
fprintf(fp,"Machine: %s Cores of type %s running %s\n",

getenv("NUMBER_OF_PROCESSORS"),getenv("PROCESSOR_IDENTIFIER"),getenv("OS"));
fprintf(fp,"MSc Thesis 2012\n");
fprintf(fp,"*********************************************************************************\n\n");

fprintf(fp,"*********************************************************************************\n");
fprintf(fp,"Simulation material information\n");
fprintf(fp,"---------------------------------------------------------------------\n");
fprintf(fp,"Aluminum-Aluminum 1st nearest neighbor pair effective energy: %f [ eV ]\n",
        energyP->E_AlAl_1);
fprintf(fp,"Scandium-Scandium 1st nearest neighbor pair effective energy: %f [ eV ]\n",
        energyP->E_ScSc_1);
fprintf(fp,"Aluminum-Scandium 1st nearest neighbor pair effective energy: %f [ eV ]\n",
        energyP->E_AlSc_1);
fprintf(fp,"Aluminum-Scandium 2nd nearest neighbor pair effective energy: %f [ eV ]\n",
        energyP->E_AlSc_2);
fprintf(fp,"Aluminum-Aluminum 2nd nearest neighbor pair effective energy: %f [ eV ]\n",
        energyP->E_AlAl_2);
fprintf(fp,"Scandium-Scandium 2nd nearest neighbor pair effective energy: %f [ eV ]\n",
        energyP->E_ScSc_2);
fprintf(fp,"Aluminum-Vacancy 1st nearest neighbor pair effective energy:  %f [ eV ]\n",
        energyP->E_AlV_1);
fprintf(fp,"Scandium-Vacancy 1st nearest neighbor pair effective energy:  %f [ eV ]\n",
        energyP->E_ScV_1);
fprintf(fp,"Aluminum Saddle point energy: %f [ eV ]\n",energyP->e_spAl);
fprintf(fp,"Scandium Saddle point energy: %f [ eV ]\n",energyP->e_spSc);
fprintf(fp,"Aluminum attempt frequency:   %.4e [ eV ]\n",energyP->vAl);
fprintf(fp,"Scandium attempt frequency:   %.4e [ eV ]\n",energyP->vSc);

fprintf(fp,"*********************************************************************************\n");
fprintf(fp,"Simulation input information\n");
fprintf(fp,"---------------------------------------------------------------------\n");
fprintf(fp,"Number of Aluminum atoms:                    %d\n",latticeP->num_Al_Sites);
fprintf(fp,"Number of Scandium atoms:                    %d\n",latticeP->num_Sc_Sites);
fprintf(fp,"Number of Vacancy sites:                    %d\n",latticeP->num_V_Sites);
fprintf(fp,"Scandium percentage:                                %.3f\n",latticeP->ScC);
```

```
	fprintf(fp,"Number of Monte Carlo Steps:                %lld\n", simP->mcs);
	fprintf(fp,"Number of snapshots saved to files: %d\n", simP->snap);
	fprintf(fp,"Temperature Applied:                                %.3f [ Kelvin ]\n", simP->T);

	fprintf(fp,"*********************************************************************************\n");
	fprintf(fp,"Simulation output information\n");
	fprintf(fp,"---------------------------------------------------------------------\n");
	for(i=0;i<simP->snap;++i){
		fprintf(fp,"Snapshot %3d simulated time:        %lld d : %d h : %d m : %.16e s\n",
			i+1, snapTime[i].days, snapTime[i].hours, snapTime[i].minutes, snapTime[i].seconds);
	}
	fprintf(fp,"Simulated time:                                %lld d : %d h : %d m : %.16e s\n",
		timeSim->days, timeSim->hours, timeSim->minutes, timeSim->seconds);
	fprintf(fp,"Simulation duration:                        %lld d : %d h : %d m : %.0f s\n",
		timeRun->days, timeRun->hours, timeRun->minutes, timeRun->seconds);
	fprintf(fp,"*********************************************************************************\n");

	fclose(fp);
}

// ********************************************************************
// Write the coordinates and clusterId of atoms, generated by a clustering algorithm, to a VTK file.
//
//  Parameters:
//		atoms	Pointer to the array where the atoms (coordinates, type, classId) are stored.
//		nAtoms	The number atoms.
//		fName	Name of the VTK file to be written.
//
void writeVTKfileClusters (char *fName, point3Dclassified *atoms, int nAtoms) {

	FILE *fp;
	int  i, naWr, na2Wr, notWr=0;

	// count atoms classified as NOISE or UNCLASSIFIED
	for(i=0;i<nAtoms;++i) {
		if( (atoms[i].classId==NOISE) || (atoms[i].classId==UNCLASSIFIED) )
			++notWr;
	}

	naWr	= nAtoms-notWr;
	na2Wr = 2*naWr;

	i=0;
	do {
		if((fp = fopen(fName,"w"))==NULL){
			printf("\nError when tryng to open VTK file for writing (%d)\n",i);
		};
	} while (fp == NULL);

	// Write the file header

	fprintf(fp,"# vtk DataFile Version 2.0\n");
	fprintf(fp,"Generated by Alfredo de Moura and Antonio Esteves\n");
	fprintf(fp,"ASCII\n");
	fprintf(fp,"DATASET POLYDATA\n");
	fprintf(fp,"POINTS %d float \n",naWr);
```

```
	for(i=0;i<nAtoms;++i) {
		if( (atoms[i].classId!=NOISE) && (atoms[i].classId!=UNCLASSIFIED) ) {
			fprintf(fp,"%.3f %.3f %.3f \n",atoms[i].x,atoms[i].y,atoms[i].z);
		}
	}

	fprintf(fp,"VERTICES %d %d \n",naWr, na2Wr);

	for(i=0;i<naWr;++i) {
		fprintf(fp,"1 %d \n",i);
	}

	fprintf(fp,"POINT_DATA %d \n",naWr);
	fprintf(fp,"SCALARS atom_type int 1 \n");
	fprintf(fp,"LOOKUP_TABLE default \n");

	for(i=0;i<nAtoms;++i) {
		if( (atoms[i].classId!=NOISE) && (atoms[i].classId!=UNCLASSIFIED) ) {
			fprintf(fp,"%d  ",atoms[i].classId);
		}
	}
	fprintf(fp,"\n");

	fclose(fp);
}

// ********************************************************************
// Write a report of the simulation/clustering analysis in a file.
//
//  Parameters:
//		simP	Pointer to the simulation parameters.
//		lattice	Pointer to the lattice parameters.
//		clustP	Pointer to the clustering analysis parameters.
//		reportP	Pointer to data to be included in the clustering analysis report.
//
void writeAnalysisReport (SimulationParameters *simP, LatticeParameters *latticeP,
		ClusteringParameters *clustP, AnalysisReportParam *reportP) {

	char	fName[200], str[20];
	FILE	*fp;
	int	 i;
	time_t	curtime;
	struct tm *loctime;
	char	data[100];

	// Get the current time
	curtime = time (NULL);
	// Convert it to local time representation
	loctime = localtime (&curtime);
	// convert date and time to the standard format
	sprintf(data,"%s",asctime (loctime));

	// copy input VTK name without extension
	//strncpy(fName,reportP->inputVTK, strlen(reportP->inputVTK)-4);
	strcpy(fName,reportP->inputVTK);
	i =  strlen(reportP->inputVTK)-4;
	fName[i]='\0';
	strcat(fName,"_analysisReport.txt");   // append "_analysisReport.txt"
```

```
i=0;
do {
	if((fp = fopen(fName,"w"))==NULL){
		printf("\nError when trying to open Analysis Report file for writing (%d)\n",i);
	};
} while (fp == NULL);

// Report Header

fprintf(fp,"*********************************************************************************\n");
fprintf(fp,"Simulation Analysys Report\n");
fprintf(fp,"Made in: %s\n",data);
fprintf(fp,"Machine: %s Cores of type %s running %s\n",

getenv("NUMBER_OF_PROCESSORS"),getenv("PROCESSOR_IDENTIFIER"),getenv("OS"));
fprintf(fp,"MSc Thesis 2012\n");
fprintf(fp,"*********************************************************************************\n\n");

fprintf(fp,"*********************************************************************************\n");
fprintf(fp,"Simulation information\n");
fprintf(fp,"---------------------------------------------------------------------------\n");
fprintf(fp,"Number of Aluminum atoms:                    %d\n",latticeP->num_Al_Sites);
fprintf(fp,"Number of Scandium atoms:                    %d\n",latticeP->num_Sc_Sites);
fprintf(fp,"Number of Vacancy sites:                    %d\n",latticeP->num_V_Sites);
fprintf(fp,"Scandium percentage:                              %.3f\n",latticeP->ScC);
fprintf(fp,"Number of Monte Carlo Steps:                %lld\n", simP->mcs);
fprintf(fp,"Number of snapshots saved to files: %d\n",   simP->snap);
fprintf(fp,"Temperature Applied:                              %.3f [ Kelvin ]\n\n", simP->T);

fprintf(fp,"*********************************************************************************\n");
fprintf(fp,"Cluster Analysis input information\n");
fprintf(fp,"---------------------------------------------------------------------------\n");
fprintf(fp,"Radius used to define atom's neighborhood (DBSCAN):          %.2f\n", clustP->eps);
fprintf(fp,"Minimum number of neighbors that turns an atom into a core atom of a cluster (DBSCAN):
	%d\n", clustP->minPts);
fprintf(fp,"Minimum number of atoms to consider a cluster as valid:       %d\n", clustP->minSize);
fprintf(fp,"Configuration file:   %s\n\n", reportP->configFile);

if (clustP->stablePrecipitates == TRUE) {
	fprintf(fp,"Input VTK file:       %s\n", reportP->inputVTK);
	fprintf(fp,"Generated VTK file:   %s\n", reportP->outputVTK);

	fprintf(fp,"*********************************************************************\n");
	fprintf(fp,"Stable Precipitates report:\n");
	fprintf(fp,"---------------------------------------------------------------------\n");

	fprintf(fp,"Percentage of Sc atoms in Al solid solution:          %.3f %%\n",
		reportP->percentNotClustered);
	fprintf(fp,"Percentage of Sc atoms in precipitates:                    %.3f %%\n",
		reportP->percentClustered);
	fprintf(fp,"Number of identified precipitates:                              %d\n",
		reportP->nPrecipitates);
	fprintf(fp,"Average size among all precipitates (in atoms):    %d\n",   reportP->avgSize);
	fprintf(fp,"Average radius among all precipitates (in Angstrom):  %.2f\n",   reportP->avgRadius);
	if(reportP->nPrecipitates>0) {
		fprintf(fp,"---------------------------------------------------------------------\n");
		fprintf(fp,"PrecipitateID   Precipitate Size (atoms)      Precipitate Radius (A) \n");
```

```
                        fprintf(fp,"-------------------------------------------------------------------------\n");
                        for(i=0;i<reportP->nPrecipitates;++i){
                                fprintf(fp,"%6d        %16d      %16.3f\n",i,*(reportP->sizeCluster+i),
                                        *(reportP->radiusCluster+i));
                        }
                }
                fprintf(fp,"-------------------------------------------------------------------------\n");
        }

        if (clustP->smallClusters == TRUE) {

        fprintf(fp,"*************************************************************************\n");
                fprintf(fp,"Small Clusters report:\n");
                fprintf(fp,"-------------------------------------------------------------------------\n");
                fprintf(fp,"Cluster Size (atoms)            Number of Clusters\n");
                fprintf(fp,"-------------------------------------------------------------------------\n");
                if (reportP->nClusters>0) {
                        for(i=clustP->minPts;i<=clustP->minSize;++i)
                                fprintf(fp,"%2d                    %8d\n",i,*(reportP->nClustersSize+i));
                }
                else {
                        for(i=clustP->minPts;i<=clustP->minSize;++i)
                                fprintf(fp,"%2d                    %8d\n",i,0);
                }

                fprintf(fp,"-------------------------------------------------------------------------\n");
        }
        fclose(fp);
}

// ****************************************************************
//  Write a simple/clean report of the simulation/clustering analysis in a file.
//  This report only includes two columns per line separared by ";" with:
//   * the size of the small clusters [1:13]
//   * the number of clusters with that size.
//
//  Parameters:
//                  simP      Pointer to the simulation parameters.
//                  latticeP  Pointer to the lattice parameters.
//                  clustP    Pointer to the clustering analysis parameters.
//                  reportP   Pointer to data to be included in the clustering analysis report.
//
void writeSimpleClustersReport (SimulationParameters *simP, LatticeParameters *latticeP,
                ClusteringParameters *clustP, AnalysisReportParam *reportP) {

        char                fName[200], str[20];
        FILE                *fp;
        int                 i;

        // copy input VTK name without extension
        // strncpy(fName,reportP->inputVTK, strlen(reportP->inputVTK)-4);
        strcpy(fName,reportP->inputVTK);
        i =  strlen(reportP->inputVTK)-4;
        fName[i]='\0';
        strcat(fName,"_analysisReport.txt");   // append "_analysisReport.txt"

        i=0;
```

```
	do {
		if((fp = fopen(fName,"w"))==NULL){
			printf("\nError when trying to open simple Analysis Report file for writing (%d)\n",i);
		};
	} while (fp == NULL);

	if (reportP->nClusters>0) {
		for(i=clustP->minPts;i<=clustP->minSize;++i)
			fprintf(fp,"%d ; %d\n",i,*(reportP->nClustersSize+i));
	}
	else {
		for(i=clustP->minPts;i<=clustP->minSize;++i)
			fprintf(fp,"%d ; %d\n",i,0);
	}
	fclose(fp);
}
```