ETH Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Institute of Computational Science

Ivo F. Sbalzarini

Diffusion in the Endoplasmic Reticulum

Theoretical Foundation, Computer Simulations, Models

Diploma thesis at the

Institute of Computational Science

Principal Adviser: Prof. Dr. Petros Koumoutsakos, Computational Science
Co-Adviser: Prof. Dr. Ari Helenius, Institute of Biochemistry, ETHZ

February 2002

Abstract

Diffusion processes in the lumen of the endoplasmic reticulum of tissue derived cells are investigated using a computational approach based on the method of particle strength exchange. 3D reconstructions of the shape of the endoplasmic reticulum are made from micrograph sections and are then used to conduct direct numerical simulations of fluorescence recovery experiments. The influences of the local geometry (i.e. the restriction of diffusion to it) on the time behavior of diffusion are discussed. Using concepts from fractal geometry, these influences are described with a single parameter that is proven to be sufficient. Based on this theory and physical principles, novel mathematical models for fluorescence recovery curves are developed and validated. They are shown to approximate the recovery curve at least one order of magnitude better than current models on the same geometry.

Zusammenfassung

Diffusionsprozesse im Lumen des Endoplasmatischen Retikulums lebender Zellen werden mittels Rechnersimulationen untersucht, wobei die “particle strength exchange”-Methode verwendet wird. Basierend auf mikroskopischen Schnittbildern werden 3D-Rekonstruktionen der Geometrie des Endoplasmatischen Retikulums gemacht, welche dann benutzt werden, um direkte numerische Simulationen von “fluorescence recovery”-Experimenten zu machen. Die Einflüsse der lokalen Geometrie (d.h. der Einschränkung der Diffusion auf diese) auf das zeitliche Verhalten der Diffusion werden diskutiert. Unter Nutzung von Konzepten aus der fraktalen Geometrie werden diese Einflüsse dann mittels eines einzigen Parameters beschrieben, welcher als dafür hinreichend bewiesen wird. Basierend auf dieser Theorie und physikalischen Grundgesetzen werden neue mathematische Modelle für “fluorescence recovery”-Kurven hergeleitet und validiert. Es wird gezeigt, dass diese die Kurven für die selbe Geometrie mindestens eine Grössenordnung besser wiedergeben als die heute verwendeten Modelle.

to my Mother

List of symbols

a	First edge vector of a triangle, $a = P_2 - P_1$ / a coefficient / a point
(a, c)	Lower-left corner coordinates of bleached box
$\hat{a}$	Least squares estimate of the slope
a_0	A coefficient
A	A set / integration constant/ bleached area
b	Second edge vector of a triangle, $b = P_3 - P_1$ / a point
(b, d)	Upper-right corner coordinates of bleached box
b_x, b_y, b_z	Size of a bin or cell in all three directions
$\hat{b}$	Least squares estimate of the y-axis intersection
B	The bleached box / a set / integration constant
$B_{i,j}$	Bin with indices (i, j)
c	Concentration / centroid of a triangle / a constant
c_c	Concentration at centroid of a grid cell
c_V	Centroid of a voxel
$C_{i,j,k}$	Cell with indices (i, j, k)
$C(E)$	Set of all continuous functions on E
D	Diffusion constant / a bounded open region
D_B	Box counting dimension
D_H	Hausdorff's dimension
D_M	Minkovski dimension
D_{xx}, $D_{xy}, \ldots, D_{zz}$	Components of the anisotropic diffusion tensor
d_s	Spectral dimension
d_w	Dimension of the walk
$e(n)$	Parallel speed-up efficiency on n processors
E	A set, geometrical object / quadratic error
E_k	Pre-fractal Euclidean approximations of order k
E_{RMS}	RMS error
E_δ	δ-neighborhood of E, δ-parallel body to E
$\mathsf{E}(\cdot)$	Expectation value of a random variable
f	A function
F	A function / FRAP value
F_0	Pre-bleach FRAP value
F_a	FRAP value just after bleaching
F_∞	Steady-state FRAP value for $t \to \infty$
f_-	Negative part of a function
f_+	Positive part of a function
g	A function
G	Green's function / a function / number of grid points
h	Grid spacing / inter-particle spacing
$H_{i,j,k}$	Head of chain list for cell (i, j, k)
$\mathcal{H}^s$	The s-dimensional Hausdorff measure
i, j, k	Indices, index counters / Integer grid coordinates in simulation
$\mathbb{I}$	Identity matrix
k	A constant / particle index
K	Heat Kernel (time dependent Green's function) / convolution kernel
l	Particle index
ℓ	Linked list / load distribution coefficient
L	Edge length of cubic domain
$L_{i,j}$	Triangle list associated to bin (i, j)
$L_{i,j,k}$	Triangle list associated to cell (i, j, k)
$\mathcal{L}^n$	The n-dimensional Lebesgue measure
m	Index / a matrix / an element / mass
m_t	Total mass in the domain
M	Number of triangles in surface / a system matrix number of terms in a finite series expansion

$\mathcal{M}$	Family of μ-measurable sets
n	Topological dimension of a space / outer normal vector / index number of measurement points / number of processors
N	Total number of particles in a simulation
N_B	Number of particles inside the bleached box, area
N_i	Number of intersections
N_{Iter}	Number of iteration steps
N_δ	Measured length in multiples of δ
N_g	Number of grid points inside the bleached box
$N(\lambda)$	Eigenvalue distribution function
N_x, N_y, N_z	Number of grid points, bins or cells in each direction
Nc_x, Nc_y, Nc_z	Number of mesh cubes in each direction
$\mathcal{N}$	Gaussian random variable (normal distribution)
$\mathbb{N}^n$	n-dimensional space of integer numbers
$\mathbb{N}_0^+$	Set of all non-negative integer numbers
overlap	Particle overlap parameter
p	Particle index / triangle index / mass per particle
$p(\cdot)$	A positive periodic function
p_t	Transition density at time t
P	A point / a particle / probability of an event
P_1, P_2, P_3	Vertex points of a triangle
q	Particle index / source term
Q	A point
Q_ϵ	Anisotropic integral operator for diffusion
$\overline{Q}_\epsilon^h$	Discrete particle approximation to Q_ϵ
r	Ratio of anisotropy (ratio of diffusion resistances in x and y)
	Radial spherical coordinate / distance / order of a particle method
r_c	Cut-off radius
R	Radius of a tube / mobile fraction
$R_n^x(r)$, $R_n^y(r)$	Effective diffusion resistances in x and y
$\mathbb{R}^n$	Euclidean space of dimension n
$\mathbb{R}^+$	Set of all positive real numbers
s	Dimension (of a Hausdorff measure) / random walk step vector
s_t	Time stretch factor
$S(n)$	Parallel speed-up on n processors
t	Time
$t_E(n)$	Execution time on n processors
t_i	Time spent on processor i
$t_{1/2}$	Half-time of fluorescence recovery
t_{final}	Final time of simulation
T	Number of time steps / time constant
TOL	Tolerance
$\mathcal{U}$	Uniformly distributed random variable
u	Unknown variable of a differential equation / velocity vector
v	A vector / volume of a particle
V	Volume
V_k	Vertex with index k
$V_{i,j,k}$	Voxel with indices (i, j, k)
w	Vertex index / quadrature weight
x	Location vector / position / a point / argument to a function
$\overline{x}$	Mean displacement
ξ	Location vector / position
x_p	Position of particle p
X	A set
$X_k(t)$	Random walk with step length 2^{-k}
$X(t)$	Brownian motion
x, y, z	Coordinates in a cartesian system
ξ, η, ζ	Coordinates in a cartesian system

y	Position / Argument to a function
Y	Intersection point / vector of measured values
α	Order of continuity / model parameter / line parameter
α_k	Time between subsequent steps in a random walk of order k
β	Model parameter / line parameter
γ	Order of continuity
δ	Size of length scale / uniform grid spacing
δ_{ij}	The Kronecker delta symbol
$\delta(\cdot)$	Dirac's delta distribution
δt	Time step size
δx, δy, δz	Cartesian grid spacings in each direction
Δ	Difference / determinant
Δ_k	Triangle k
Δz	Distance between subsequent slices
ϵ	Kernel core size for particle method
ε	Residual / error
η	Number of voxels in a voxel representation / kernel function
η_ϵ	Regularized kernel function for a particle method
η_α, η_β	Learning rates for gradient descent
φ	Eigenfunction / azimuthal angle / angular coordinate on tube
ϕ_0	Volume-filling coefficient
Φ	Regression matrix
ψ	Spherically symmetric matrix cut-off function
ψ^ϵ	Regularized matrix cut-off function
κ	Dimensionless constant
λ	Eigenvalue / line parameter / ratio of a similarity transformation
μ	A measure / Micro
ν	Restriction of a measure / a distribution
π	Ratio of circumference to diameter of a circle ($= 3.14159265359\ldots$)
σ	Standard deviation / voxel edge length / source term / anisotropic kernel
σ_ϵ	Regularized anisotropic kernel
τ	Time / time constant
ϑ	Polar angle for spherical coordinates
Θ	Vector of unknown parameters
$\overline{\Theta}$	Scalar cut-off function
$\widehat{\Theta}_{LS}$	Least squares estimation of unknown parameters
ζ	Particle mollifier function / axial coordinate on a tube
ζ_ϵ	Regularized particle mollifier function
χ_A	Indicator function of set A
ω	Particle strength (in general)
Ω	Domain of solution
$\aleph$	Triangulated surface, i.e. its set of triangles
∂	Partial derivative / boundary of domain
∇	Nabla Operator
∇^2	Laplace Operator
∇^2_ϵ	Continuous integral approximation to the Laplace operator
$\nabla^2_{\epsilon,h}$	Discrete sum approximation to the Laplace operator
∞	Infinity
$\lvert\cdot\rvert$	Absolute value / diameter of a set
$\lVert\cdot\rVert_\infty$	Infinity norm of a function
$\overline{(\cdot)}$	Upper dimension
$\underline{(\cdot)}$	Lower dimension
$\widetilde{(\cdot)}$	Finite sum approximation of an infinite series
$(\cdot)^\top$	Matrix transpose
$(\cdot)^h$	Particle approximation with inter-particle spacing h
$(\cdot)^*$	True value (as opposed to model value)
$(\cdot)'$	Derivative with respect to the function's only variable
$(\cdot)_0$	Initial value, property at time zero

$(\cdot)_\infty$	Steady-state value for $t \to \infty$
$(\cdot)_x$, $(\cdot)_1$	Cartesian x component or value in x direction
$(\cdot)_y$, $(\cdot)_2$	Cartesian y component or value in y direction
$(\cdot)_z$, $(\cdot)_3$	Cartesian z component or value in z direction
$(\cdot)_l$	Lower bound of a variable
$(\cdot)_h$	Upper bound of a variable
$(\cdot)_{min}$	Minimum value of a variable
$(\cdot)_{max}$	Maximum value of a variable
$(\cdot)_{opt}$	Optimum value of a variable
$(\cdot)_{mirror}$	Boundary mirror image
$(\cdot)_n$	At time step n
$(\cdot)_k$	At iteration step k
$O(\cdot)$	The Landau symbol
$\mathcal{O}(\cdot)$	Operation count at the order of
$\overrightarrow{AB}$	Vector pointing from A to B
$\times$	Cartesian product / vector cross product
$\forall$	For all
$(\cdot)!$	Faculty of a number
$\stackrel{!}{=}$	Must be equal (requirement)
$\asymp$	Asymptotically equal
$\in$	In a set, member of a set
$\notin$	Not in a set
$\setminus$	Set minus
$\emptyset$	Empty set
$spt(\cdot)$	Support of a measure
$int(\cdot)$	Interior of a closed set
$\mathrm{Vol}^n(\cdot)$	n-dimensional volume of an object or set
$[a, b]$	Interval between a and b including its boundaries
(a, b)	Interval between a and b excluding its boundaries
$P_{(n:m)}$	Sub-vector with components n to m of vector P
$\bigcup_{i=1}^{n}$	Multiset union of n sets
$\sum_{i=1}^{n}$	Sum of n terms
$\prod_{i=1}^{n}$	Product of n terms
inf	Infimum of a set (multidimensional minimum)
sup	Supremum of a set (multidimensional maximum)
min	Minimum of a set
max	Maximum of a set
lim	Limit
log	Natural logarithm (i.e. logarithm to the base e)
ceiling(x)	Smallest integer larger or equal x
floor(x)	Largest integer less or equal x
int(x)	Integer part truncation of x
$Tr(\cdot)$	Trace of a matrix
$\#\{\cdot\}$	Counting measure of a set (number of elements in the set)
$\Rightarrow$	Logical implication
$\Longleftrightarrow$	Only if (iff)
$\mapsto$	Maps to (mapping or transformation)
$\rightarrow$	To, towards
`<.>`	Mandatory argument
`[.]`	Optional argument

Further symbols may be defined in the text as they are encountered. Name conflicts may occur. However, in such cases the meaning of a symbol is always clearly defined in the text. Vectors are not printed in bold or otherwise marked as their occurrence depends on the dimension of the underlying space.

Acronyms

2D	two-dimensional
3D	three-dimensional
ASCII	American Standard Code for Information Interchange
C	The C programming language
CD	Compact Disc
CD-ROM	Compact Disc Read-Only Memory
CFL	Courant-Friedrichs-Levy (condition)
DNA	Desoxy-Ribonucleic Acid
DVI	Device Independent Interface file
EPS	Encapsulated Post-Script
ER	Endoplasmic Reticulum (an organelle of a biological cell)
ETH	Eidgenössische Technische Hochschule (Swiss Federal Institute of Technology)
FD	Finite Difference
FFT	Fast Fourier Transformation
FRAP	Fluorescence Recovery After Photobleaching
GB	Giga Byte
GFLOPS	Billion Floating Point Operations Per Second
GFP	Green Fluorescent Protein
ICoS	Institute of Computational Science
IRIX	The Silicon Graphics Operating System
kDa	Kilo Dalton (a protein molecular mass unit)
MB	Mega Byte
MBit	Mega bit
MFLOPS	Million Floating Point Operations Per Second
MIPS	Million Instructions Per Second
MOPS	Million Operations Per Second
MPI	Message Passing Interface
MPEG	Motion Picture Experts Group
ODE	Ordinary Differential Equation
OpenDX	A 3D data visualization tool
PDE	Partial differential equation
PDF	Portable Document Format
PS	Post-Script
PSE	Particle Strength Exchange
PSE3D	Three-dimensional particle strength exchange method
RMS	Root Mean Square
SGI	Silicon Graphics, Inc.
ssGFP-KDEL	ER resident signal sequence GFP
VERO	Cell from green monkey intestine epithelium

Names of simulation runs

bip2	erp574_1
clx	erp574_2
erp57	erp5722
erp572	erp5723
erp573_1	erp5724
erp572_2	8s
erp573_3	8.2

Contents

Foreword

This report describes the work and presents the results of a diploma project that has been hosted by the Institute of Computational Science (ICoS) at the Swiss Federal Institute of Technology (ETH) in Zürich, Switzerland. A close collaboration with the Institute of Biochemistry at ETH enabled the interdisciplinary and application-related character of the project. The work is a direct continuation of the semester project [Sbalzarini (2001)], to which frequent references will be made throughout the text. The reader will be assumed to be familiar with the basic concepts, ideas and results of [Sbalzarini (2001)]. Namely preliminary knowledge about the endoplasmic reticulum (ER), its function and shape, the concept of fluorescence recovery after photobleaching (FRAP), the green fluorescent protein (GFP) and the notions of random walk, Brownian motion and diffusion is expected. Readers not familiar with any of these subjects are advised to read chapters 2 (for the ER in general as well as its function and shape), 3 (for the concept of FRAP analysis and GFP) and 4 (for diffusion and random walk) of [Sbalzarini (2001)] as an introduction to the present work. It is electronically available from the e-collection of ETH library.[1]

The main purpose of this project is to further pursue the investigation of diffusion processes inside the endoplasmic reticulum of tissue derived cells. Particularly, the following goals are aimed at:

- Generation of 3D reconstructions of real ER geometries and representation as triangulated surfaces in the computer
- Development of a 3D geometry preprocessor for use with existing simulation codes
- Mathematical treatment of the theory of diffusion on fractal sets
- Identification of the relevant statistical geometry parameter and proof of its sufficiency
- Investigation of the usefulness of fractal models to describe protein diffusion in the ER
- Measurement of the fractal dimension of the ER surface in space and comparison to the findings in [Sbalzarini (2001)]
- Change of the simulation technique employed from random walk to the superior method of particle strength exchange
- Direct numerical simulations of FRAP experiments in full 3D reconstructions of real ER structures
- Development of a novel FRAP data model to fit simulation and experiments
- Comparison of simulations and models to experiments

[1] http://e-collection.ethbib.ethz.ch/browse/alph/s.html

Due to the interdisciplinary nature of the project, this report is targeted at both molecular biologists and applied mathematicians. In an effort to make it (at least partially) comprehensive to both groups, certain repetitions and somewhat lengthy-seeming explanations could not be avoided in the text. To keep it as clean and general as possible, vectors are not printed in bold or otherwise marked as their occurrence depends on the dimension of the underlying space. It is always clear from the context which variables are vectors (e.g. position in spaces of more than one dimension) and which are scalar (e.g. time).

Chapter 1 contains an overview of the current state and a survey of relevant literature. Moreover, it gives a concise introduction to the strategy that will be followed for the rest of the work. Chapter 2 starts with a description of the process of 3D reconstruction of ER shapes from a stack of parallel microscopical sections. Chapter 3 describes the algorithms that have been implemented in order to check the resulting triangulation for topological and syntactical validity as defined in [Sbalzarini (2001)].

Chapter 4 represents the first piece of the theoretical part of this work. It will introduce and summarize some basic theoretical concepts along with their notation and terminology for later use. Although all of this theory is commonly available from literature, it is concisely summarized here in a single place using a notation that is consistent with the rest of the work. However, readers are free to skip the chapter if familiar with the notions presented therein or to consult their own books of choice.

In chapter 5 the concepts previously introduced will be applied to measuring the fractal dimension of the ER surface in 3D space. As every work should be built on a sound theoretical foundation, chapter 6 will be concerned with the theory of diffusion on fractal sets. This chapter relies on the information presented in chapter 4 in order to identify the geometrical parameter that is relevant to the time behavior of diffusion and prove its sufficiency for two large classes of fractals.

This concludes the theoretical part of the report. Chapter 7 will describe the simulation methods that will be used to simulate protein diffusion in reconstructed ER shapes. Particularly, the method of particle strength exchange (PSE) and its anisotropic extension are presented. However, no anisotropic runs will be conducted in this project since biologists do not yet know anything about the anisotropy of the medium filling the ER lumen. However, all simulation algorithms are set up to handle anisotropic diffusion in order to be prepared for future work. Due to its many favorable properties, the PSE method will replace the random walk code developed in [Sbalzarini (2001)] for the rest of this project. Before applying it to real ER problems, it is validated on a test case against the analytic solution, a finite difference code and the random walk simulation in chapter 8.

Chapter 9 contains results from the actual FRAP simulations in 3D reconstructed ER samples from micrographs. Moreover, the influence of position and size of the bleached spot is investigated. In order to validate the simulations, their results are compared to experiments in chapter 10 and it is demonstrated how to obtain numerical values for the diffusion coefficient out of such simulations. However, it would be much too complex to run a full direct numerical simulation for every experiment one wishes to evaluate quantitatively. Therefore, chapter 11 starts looking into new mathematical models to describe FRAP curves with a small number of parameters. In chapter 12 the new models derived are then compared to existing ones for diffusion in a cubic box as well as diffusion in reconstructed ER samples and experimental data.

In order to allow easy re-use of the simulation programs, result data sets and resources of this work, all files are contained on two CD-ROMs that accompany this

report. Besides the complete program source codes, the CDs also contain the text files of this document (LaTeX source, EPS and PS figures, xfig drawings and all pixel images) as well as complete PostScript and PDF versions of it. Moreover, color images, movies of certain runs, all simulation and experimental data as well as the PowerPoint file of the final presentation can be found on them. Appendix A contains the complete table of contents.

Appendix B lists the numerical values of the model parameters found by fitting different FRAP data models to different ER samples. The Fortran source codes of all the programs entirely developed in this work are given in appendix C. It also contains basic information about the program structures and their usage. Since an existing code has been adapted and extended for the PSE simulations, its source code will not be given in the appendix but it is contained on CD #1 nevertheless.

Each algorithm presented in this report contains marginal numbers referring to the corresponding statement labels in the Fortran routine that implements it. This allows easy understanding of the Fortran programs and immediately links the algorithms to their respective implementation for convenient reference and easy reproduction. To keep the notation consistent, all algorithms and formulae in this report use Fortran index numbering starting at 1 rather than 0.

Acknowledgments

None of this work would have been possible without the generous support by various people that I hereby like to acknowledge. Particularly, I wish to thank my adviser Prof. Petros Koumoutsakos for hosting this project at his institute, for his continuing support and the various important contributions he made to this project as well as my co-adviser Prof. Ari Helenius for coaching the biological part of this project and supporting me with both information and laboratory infrastructure. Thanks to Anna Mezzacasa of Helenius' group for all the days and nights she spent in the laboratory to make the validation experiments and to prepare all the micrographs and to Dr. Alicia Smith for supplying the experiment's protocols as well as proofreading the biological part of this report. I am also indebted to ICoS' Dr. Nicol Schraudolph for helping me with model fitting and gradient descent techniques as well as to Dr. Jens Walther for his continuing support in using our super-computing facilities and codes. For developing the PSE simulation code and making it available to me as well as for proofreading parts of this thesis, I owe a debt of gratitude to Stephanie Zimmermann of the institute of hydromechanics and water resources management as well as to Dr. Jens Walther who developed the underlying framework and MPI parallelism of the PSE code. Great thanks also to Prof. Urs Stammbach, Prof. Christoph Schwab, Prof. Alain-Sol Sznitman, Prof. emer. Christian Blatter and Dr. Michail Loulakis, all of the department of mathematics, for their various discussions and literature pointers concerning the theory of fractals and diffusion on fractal sets. And last but not least, special thanks to BitPlane, Inc. and particularly to Dr. Patrick Schwarb for granting a free demo license for their 3D geometry reconstruction tool Imaris for the duration of this project.

Zürich, February 2002

Chapter 1

Introduction

The investigation of diffusion in complex-shaped organelles has become an important area of research in recent biology and fluorescence recovery analysis is the tool of choice (see [Lippincott-Schwartz, Snapp & Kenworthy (2001)] for a comprehensive review of the methods available). While early work applied 2D and even 1D models of homogeneous and isotropic diffusion to the problem of fluorescence recovery after photobleaching (FRAP), doubts about their validity and a potential influence of the underlying geometrical structure of the ER arose soon (e.g. stated in [Ölveczky & Verkman (1998)] or [Ellenberg et al. (1997)]) and biologists started looking into mathematical models and computer simulations to capture the effects (see e.g. [Ölveczky & Verkman (1998)], [Dayel, Hom & Verkman (1999)] or [Siggia, Lippincott-Schwartz & Bekiranov (2000)]). However, the problem has been taken up in applied physics long before. [Axelrod et al. (1976)] started developing models for fluorescence recovery curves as early as 1976 and their publication might have fallen into oblivion by now. In lack of simulation algorithms and computing power however, they restricted themselves to two dimensional problems and simple geometries whereas the present work will deal with 3D problems in real ER geometries. Nevertheless, they discovered some important basics and were able to match experimental data with high accuracy. By application of single particle tracking methods, [Kusumi, Sako & Yamamoto (1993)] and recently [Dietrich et al. (2002)] proposed an alternative to fluorescence recovery experiments if small statistical samples are sufficient. A comparative summary of the methods available can for example be found in [Cheezum, Walker & Guilford (2001)].

In order to be able to investigate the influence of geometrical factors on the time behavior of diffusion (i.e. the eigenvalues of the Laplacian) of molecules inside the lumen of the endoplasmic reticulum[1], several building blocks from different scientific disciplines are needed:

- Cell biology
- Biochemistry and biotechnology
- Physics of transport phenomena
- Numerics and applied mathematics
- Computer simulation techniques
- Theoretical mathematics

[1] Readers not familiar with the endoplasmic reticulum or diffusion are advised to read chapters 2 through 4 of [Sbalzarini (2001)] or the corresponding chapters of [Alberts et al. (1997)]

- Fractal and Euclidean geometry
- Image processing and reconstruction methods
- Fluorescence microscopy
- Optimization algorithms

Each of them is a well-known subject on its own. For an introduction to cell biology and biochemistry, see for example [Alberts et al. (1997)]. The numerics and computer simulation methods that will be applied are for example described in [Hockney & Eastwood (1988)] or [Cottet & Koumoutsakos (2000)] as well as in [Degond & Mas-Gallic (1989a)] and [Degond & Mas-Gallic (1989b)]. Alternative methods can for example be found in [McCorquodale, Colella & Johansen (2001)] and [Beaudoin, Huberson & Rivoalen (2001)]. The field of fractal geometry and the theory of partial differential equations (PDE) involving it is subject to extensive and ongoing research in theoretical mathematics. A good introduction to fractal geometry in general can be found in [Mandelbrot (1982)]. More in-detail and mathematically profound information are given in [Falconer (1985)], [Falconer (1990)] and [Falconer (1997)], which also contains some theory about PDEs on fractal domains. A concise summary of theorems and definitions relevant for fractal geometry is given in [Monks (2001)]. In addition, several research papers contain related information. They will be cited in the text where relevant. Information about image processing and 3D reconstruction from parallel sections are for example contained in [Baldock & Graham (2000)].

Despite all those efforts in the different fields, little work has been done in trying to combine them. The three most noticeable projects are the very recent one on cellular fluid mechanics by [Kamm (2002)], the application of fractal theory to image compression ([Mitra, Murthy, Kundu & Bhattacharya (2001)]) or the first – however somewhat aimless – application of fractal geometry to biological cell image analysis by [Smith, Marks, Lang, Sheriff & Neal (1989)]. The present work aims at combining all of the above areas in order to find a better understanding of molecular diffusion in the endoplasmic reticulum which hopefully will lead to novel FRAP data models that can be used to determine diffusion constants from experimental data.

Figure 1.1 shows the global picture of how these different techniques and areas of science will be combined. The corresponding chapter numbers of where they are treated in this report are printed in *italics*. Boxes with black print symbolize steps of the present work whereas rounded boxes with blue print signify external inputs to the process. The final goal is highlighted using red letters. The left and bottom parts only need to be done once, the right top-down line describes the repeated application in daily laboratory work.

The starting point for all mathematical modeling are stacks of micrographic slices of stained ERs. These images are then used to perform a 3D geometry reconstruction in order to obtain a triangulated surface description (see [Sbalzarini (2001)]) of the real-world ER shape in the computer (see chapter 2). This triangulated representation then needs to be checked for different criteria of validity in order to make sure that it actually represents a valid ER structure (i.e. its interior needs to be a connected closed subset of $\mathbb{R}^3$). The topological criteria and algorithms used are described in chapter 3. The final and valid triangulated geometry can then be used as a computational domain for subsequent numerical simulations of diffusion. The simulation techniques employed and their validations are contained in chapters 7 and 8. Choosing appropriate initial conditions then allows realistic simulations of FRAP experiments. Since for the simulation runs, the corresponding diffusion

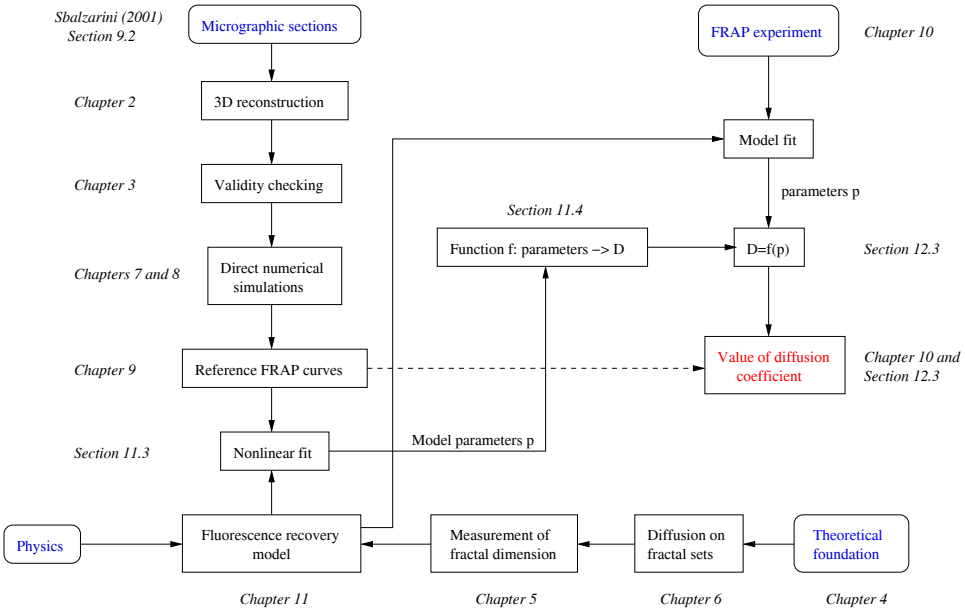


Figure 1.1: General research strategy outline

constant is known (it has to be passed to the simulation algorithm as a parameter), these FRAP curves serve as reference curves with no unknown properties. Fitting these curves to experimental data *in the same ER geometry* would in principle allow to directly determine the unknown real-world diffusion constant (dashed arrow). Since it would be too complicated to always take a stack of micrographs and run a full-fledged computer simulation for each cell one wishes to make a fluorescence recovery experiment, data models are looked for.

The starting point to develop such models is the theoretical foundation given by mathematical set theory, the physics of transport phenomena and fractal geometry, the most important concepts of which are summarized in chapter 4. Certain knowledge about the solution of the diffusion equation on fractal domains (cf. chapter 6) and the fractal properties of the specific ER geometries at hand (chapter 5) then allow to infer novel, physically motivated FRAP data models (see chapter 11). These models can be fitted to the simulated reference FRAP curves in order to get numerical values for their parameters. As both the model parameters and the underlying diffusion constant are known, it is in principle possible to find the physically given functional relationship f between them (section 11.4).

For productive laboratory use, one starts from an experimentally measured FRAP curve with unknown diffusion coefficient. Fitting the newly found fluorescence recovery model to such experimental data yields values for the model parameters. Using the functional relationship f finally translates them into the sought-after value of the diffusion constant.

Chapter 2

3D reconstruction of micrographs

Since it is the goal of this project to take the ideas in [Sbalzarini (2001)] to the third dimension, some 3D representation of the ER is needed. This should be in the form of a triangulated surface in $\mathbb{R}^3$ meeting the topological requirements stated in [Sbalzarini (2001)]. Fortunately, there exists a good commercial software package for 3D reconstruction from parallel sections called Imaris[1]. It not only handles the reconstruction but also does the triangulation of the resulting surface. However, the problems regarding aliasing due to insufficient resolution (see [Shannon (1948)]) and geometrical ambiguity[2] remain. Imaris "solves" these problems by making some best guesses such that the overall result is closed, connected and as smooth as possible. In addition, Imaris does a Gaussian filtering of the geometry to remove noise, artifacts and geometrical details below a certain level of resolution. The minimum size of the triangles can be controlled as well as the resolution and smoothness of the resulting reconstruction. For the reconstructions made in this work, the data are first filtered with a Gaussian filter of kernel size 2.208 in all three directions. The reconstruction is then done using voxels[3] of size $3.15 \times 3.15 \times 2.8$. Moreover, objects are required to be closed and contain more than 2000 triangles.

For illustration purposes, one example of the numerous reconstructions that have been conducted is shown hereafter. All reconstructions were made using Imaris 3.1.3 by BitPlane, Inc. running on a Silicon Graphics Octane workstation under IRIX 6.5. BitPlane kindly granted a demo license of Imaris for the duration of this project.

Figure 2.1 shows the 16 plane parallel slices of stained ER of an example VERO cell as they come from a Leica confocal microscope (see [Sbalzarini (2001)] chapter 9.2 for the detailed protocol). They are then fed to Imaris which does the reconstruction and triangulation. The result is saved as an OpenInventor file (see [SGI (1992a)] and [SGI (1992b)] for the file format specifications) which can be read by the simulation programs developed in this project. Figure 2.2 shows an enlarged part of the resulting triangulated surface; the view in figure 2.3 additionally has its hidden lines removed. Finally, figure 2.4 shows a sample view of the outcome as a 3D shaded surface.

[1] http://www.imaris.com

[2] two parallel lines on successive slices could for example stem from two parallel planes or from a tube intersected twice parallel to its axis

[3] The 3D analogue of pixels

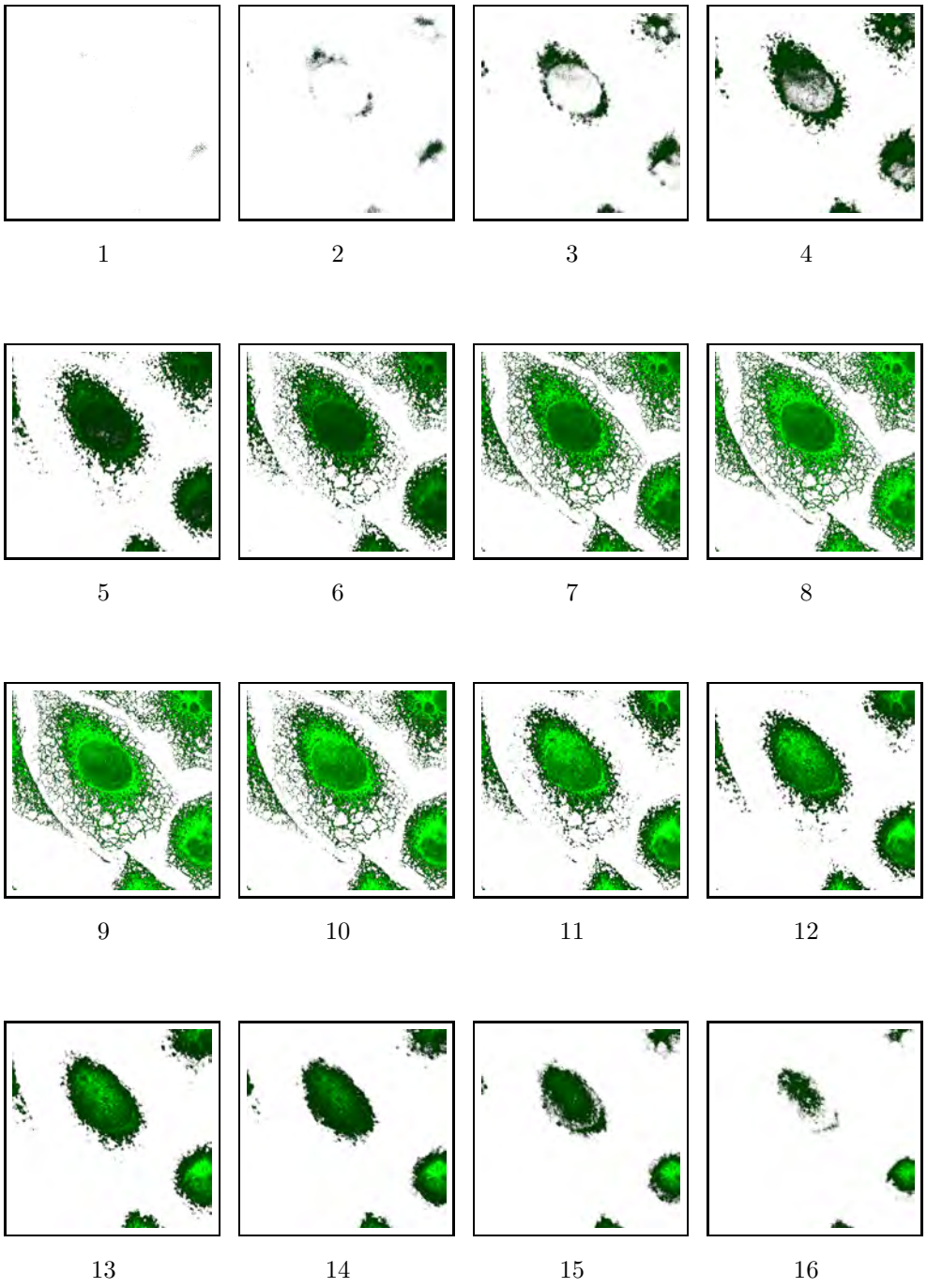


Figure 2.1: Sample micrograph stack of stained ER used for 3D reconstruction. The slices are taken at vertical distances of $\Delta z = 0.1\,\mu$m as described in [Sbalzarini (2001)]. (Courtesy of Anna Mezzacasa)

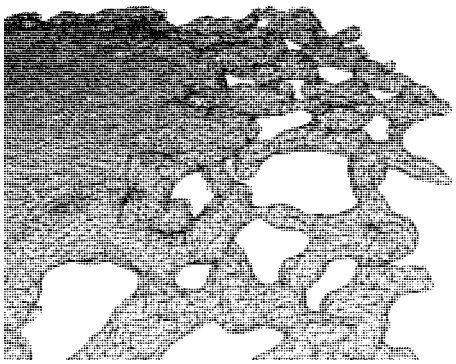


Figure 2.2: Surface triangulation

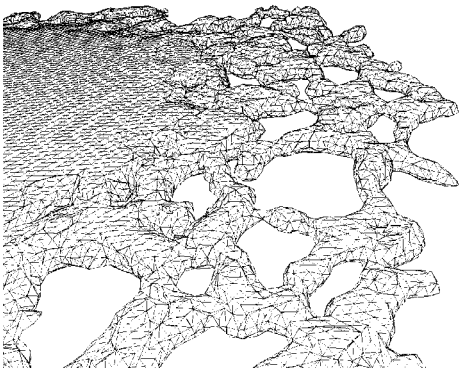


Figure 2.3: Surface triangulation with hidden lines removed

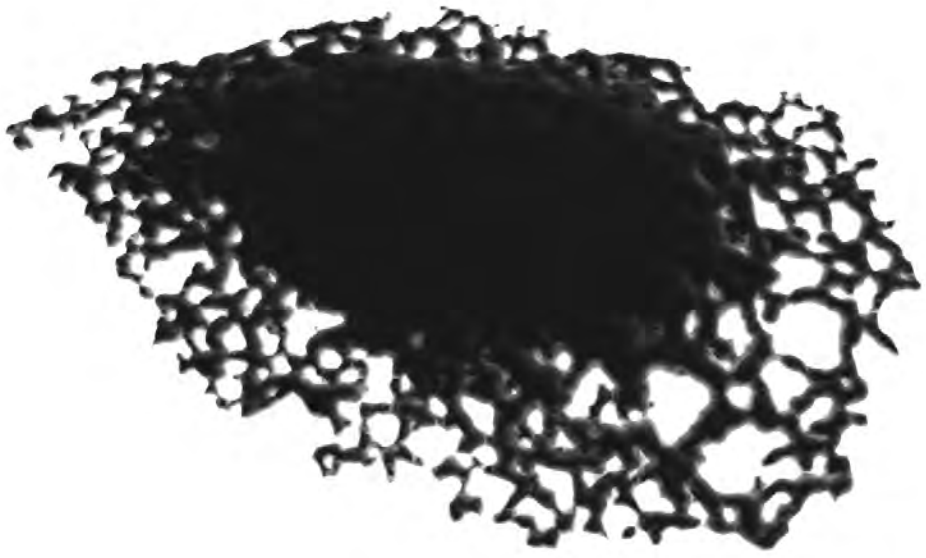


Figure 2.4: Shaded surface view of 3D reconstruction

Chapter 3

Validating the reconstruction

Now having a triangulated 3D reconstruction of the ER surface, the question arises whether this triangulation meets all the requirements stated in [Sbalzarini (2001)]. Therefore, a subroutine is written that performs some checks on the geometry. In order to be able to describe its algorithms, the requirements are first restated in a slightly different but topologically equivalent form.

3.1 Restatement of requirements

A triangulation is given by a set $\aleph$ of triangles Δ_i. Each triangle is described by the location vectors of its three vertices P_1, P_2, P_3. Since the ER membrane encloses a closed, connected and contiguous subspace of $\mathbb{R}^3$, the triangulation has to meet a minimal set of requirements. We call a triangulation $\aleph$ *topologically valid* if and only if it meets all of the following:

1. $\aleph$ describes a *closed, connected*[1] surface in $\mathbb{R}^3$
2. No Δ_i must contain any edge of length $\leqslant 0$
3. No $P_j(\Delta_i)$ must be inside any other triangle
4. Two neighboring triangles must have *exactly* two vertices P_j in common

We call it *syntactically valid* if it is topologically valid and additionally satisfies:

1. The vector $n = \overrightarrow{P1P2} \times \overrightarrow{P1P3}$, where "$\times$" is the vector cross product, must point outward (i.e. out of the domain enclosed by $\aleph$) for every Δ_i.

The first point of the topological validity is particularly difficult to check. The code therefore uses the following equivalent set of criteria:

A) All edges are of length > 0 (exclude degenerate cases).

B) No vertex lies inside any other triangle (only allow the minimum set).

C) Every triangle has exactly 3 neighbors sharing 2 vertices with each (ensure continuity of surface).

D) No edge of any triangle intersects the face of any other triangle (exclude overlaps).

[1] meaning that any point enclosed by $\aleph$ can be connected to any other point enclosed by $\aleph$ with a continuous curve that never intersects $\aleph$

The check for syntactical validity is straightforward and the criterion is not reformulated. However it can only be done for sets that are at least topologically valid which means that the two checks are split into different subroutines whereas the second can only be called when the first terminates without error. The two routines are briefly presented in turn below. It is worth mentioning that all floating point comparisons are performed using a global geometry tolerance TOL. Thus an edge is for example considered to be of length 0 if it is of length $< TOL$ and two numbers are considered equal if their difference is less than TOL. Typically, TOL is about 10^{-10} for double precision numbers on a 32bit machine which ensures sufficient robustness against round-off errors while still maintaining good accuracy. The geometry checking routines give us a tool to validate the triangulated 3D reconstructions made by Imaris (cf. chapter 2) in order to ensure proper functioning of the simulation codes that rely on certain properties of the triangulation. If any reconstruction fails the test, it is redone using different filter settings in Imaris until a valid reconstruction results.

3.2 Checking the topological validity

The algorithms described hereafter are implemented in the function `chk_topology` in `chktriang.f90`. See appendix C.5.4.7 for a complete listing of the source code. Marginal numbers in the following paragraphs conveniently refer to statement numbers of the listing in appendix C.5.4.7.

The check of requirement A) is trivial, the program just loops over all triangles and
17-37 calculates the lengths of all the edges. For requirement B), one uses the fact that for every point P inside a triangle, the following holds:

$$P_1 + \alpha a + \beta b = P \qquad \text{for } \alpha > 0,\ \beta > 0,\ \alpha + \beta < 1$$

where $a = P_2 - P_1$ and $b = P_3 - P_1$ are two edge vectors of the triangle. The corresponding linear set of equations

$$\begin{bmatrix} | & | \\ a & b \\ | & | \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} | \\ P - P_1 \\ | \end{bmatrix}$$

is over-defined and will not have a solution for general P. To check whether the system has a solution for a given P, the 2×2 subsystem

$$\begin{bmatrix} | & | \\ a_{(1:2)} & b_{(1:2)} \\ | & | \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} | \\ P_{(1:2)} - P_{1,(1:2)} \\ | \end{bmatrix}$$

is solved for α and β (the code directly contains the symbolic solution for per-
38-69 formance reasons) and then one checks if this solution also satisfies the omitted third equation. If this is the case, point P lies inside the triangle if $\alpha > 0$, $\beta > 0$ and $\alpha + \beta < 1$. This is checked for every vertex of every triangle against every other triangle which makes this part of the algorithm $\mathcal{O}(3M^2)$ when M is the total number of triangles. This is acceptable since the number of triangles is typically much smaller than the number of points and rarely exceeds 100'000. Moreover, the algorithm is only run once per triangulation and optimization is often a source of programming errors that should be avoided for a "checking" code.

Requirement C) is again easy to check. For every triangle, all the other triangles are
70-101 searched and the number of common vertices is counted. Then the number of its

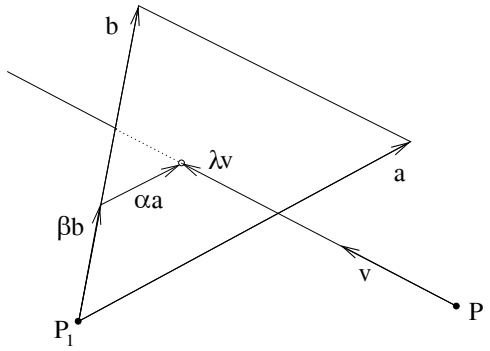


Figure 3.1: Geometry of a line intersecting a triangle

neighbors (defined as triangles that share exactly 2 vertices) is determined. Every triangle should have exactly 3 neighbors.

Condition D) is checked by explicitly calculating the intersection point of every edge of every triangle with every other triangle. If such an intersection point is found, the triangulation cannot be topologically valid. However, this method calls for a fast triangle/line intersection subroutine. Such a general routine is implemented in `intersect.f90` (see appendix C.5.4.8). The task is to intersect a line $P + \lambda v$ with a triangle, given by its first vertex P_1 and the two edge vectors a and b according to figure 3.1. The intersection point is given by the following equality: 102-129

$$P + \lambda v = \alpha a + \beta b + P_1$$

for some α and β. This corresponds to the linear system of equations:

$$\underbrace{\begin{bmatrix} | & | & | \\ a & b & v \\ | & | & | \end{bmatrix}}_{M} \begin{bmatrix} \alpha \\ \beta \\ -\lambda \end{bmatrix} = \begin{bmatrix} | \\ P - P_1 \\ | \end{bmatrix}$$

If $det(M) = 0$ (up to the global tolerance TOL) no intersection point exists. Else its position is given by $P + \lambda v$ and it lies in the triangle if $\alpha \geqslant 0$ and $\beta \geqslant 0$ and $\alpha + \beta \leqslant 1$. The routine returns -1 if no intersection point *in the triangle* exists and $|\lambda|$ if one is found. 16 31-32

3.3 Checking the syntactical validity

This check is implemented in the function `chk_orientation` in `chktriang.f90` (see appendix C.5.4.7). First, the normal $n = a \times b$ of each triangle is calculated or read from memory if this has been done earlier. The code then checks whether the point that is reached by following the normal a "small distance" starting at the triangle's centroid c is inside the domain or not. "Small distance" is chosen to be 5 times the geometrical tolerance, thus $c + 5TOL \cdot n$ has to lie outside of the domain for the triangle to be properly oriented. This is checked using the function `point_in_domain` which is discussed in section 7.2.5, algorithm 7.8. `chk_orientation` returns `.TRUE.` if all the triangles are oriented correctly, `.FALSE.` otherwise. 140-146

Chapter 4

Background on measures and dimensions

4.1 Measures and distributions

Measures and distributions are a central tool in the mathematics of fractals. Since some theoretical studies involving fractal sets will be conducted in later chapters, a few basic concepts are introduced here. However, the basics of fractal geometry and dimensionality as introduced in [Sbalzarini (2001)] will not be repeated but assumed to be known. Only measures and distributions on the Euclidean $\mathbb{R}^n$ are needed for the rest of this text. Note that what will be called "measure" hereafter is often termed "outer measure" in general texts on measure theory.

Simply put, a measure is a way of assigning a scalar numerical value to a set such that the principle "the whole is the sum of its parts" applies[1]. Readers not yet familiar with the concept of measures may find it helpful to imagine them as physical "mass distributions" or "charge distributions". By restricting this section to measures on $\mathbb{R}^n$, many of the awkward features that can occur in more general spaces are avoided.

Definition 4.1 (Measure). Let $X \subset \mathbb{R}^n$. We call $\mu : X \mapsto \mathbb{R}^+$ a *measure* on X if μ assigns a non-negative number (possibly ∞) to each subset of X such that:

(a) $\mu(\emptyset) = 0$

(b) $A \subset B \quad \Rightarrow \quad \mu(A) \leqslant \mu(B)$

(c) if $A_1, A_2, \ldots$ is a countable sequence of sets, then:

$$\mu\left(\bigcup_{i=1}^{\infty} A_i\right) \leqslant \sum_{i=1}^{\infty} \mu(A_i)$$

Thus (a) requires the empty set to have measure zero and (b) states that the bigger set has the larger measure. Property (c) ensures that the measure of any set is no more than the sum of the measures of the pieces in any countable decomposition. Exact equality holds for "nice" disjoint sets A_i, in particular (but not only) if A_i are disjoint *Borel sets*[2].

[1] Strictly speaking this is only true if the set can reasonably be decomposed into a finite or countable number of disjoint subsets.

[2] Every set that can be constructed starting with open or closed sets and taking countable unions or intersections a finite number of times is a Borel set. All sets encountered in this text are Borel sets.

This leads to the concept of *measurability*. Given a measure μ there is a family of subsets of X on which μ behaves in a nice additive way such that property (c) in definition 4.1 holds for equality. This family is called μ-*measurable* and termed $\mathcal{M}$. The exact definition is:

Definition 4.2 (Measurable). A set $A \subset X$ is called μ-*measurable* if

$$\mu(E) = \mu(E \cap A) + \mu(E \setminus A) \qquad \forall \quad E \subset X$$

μ is termed a *Borel measure* on $X \subset \mathbb{R}^n$ if all Borel subsets of X are μ-measurable. It may be shown that μ is a Borel measure if and only if $\mu(A \cup B) = \mu(A) + \mu(B)$ whenever $A, B \subset X$. A Borel measure is termed *Borel regular* if every subset of X is contained in a Borel set of the same measure. All measures that will be encountered in this text are Borel regular on $\mathbb{R}^n$ or the pertinent subset thereof. Thus, to simplify notation, the term "measure" will mean "Borel regular outer measure" and "set" will mean "Borel set" throughout this report.

A measure μ on X with $\mu(X) < \infty$ is called *finite*; if $\mu(A) < \infty$ for every bounded set A, it is *locally finite*. The *support* of μ is the set on which the "mass" of μ is concentrated, thus the smallest closed set X such that $\mu(\mathbb{R}^n \setminus X) = 0$. The support of any measure is always closed and termed $spt(\mu)$. Its definition is:

Definition 4.3 (Support). The *support* of a measure μ is the smallest closed set with complement of measure zero, thus:

$$spt(\mu) = X \setminus \cup\{U : U \text{ is open and } \mu(U) = 0\}$$

A measure on a bounded subset X of $\mathbb{R}^n$ for which $0 < \mu(\mathbb{R}^n) < \infty$ is called a *distribution*. Imagine for example a finite mass spread over the set X in some way. Then, the mass contained in any subset A of X is given by $\mu(A \subset X)$. As distributions are a subclass of measures, all properties for measures also hold for distributions.

Sometimes, a *restriction* of a measure to a set E is used. Let μ be a measure on $\mathbb{R}^n$ and E a Borel subset of $\mathbb{R}^n$. We can define a new measure ν on $\mathbb{R}^n$ through $\nu(A) = \mu(E \cap A)$ for all A. This new measure ν is called *restriction of μ to E* and it is a measure on $\mathbb{R}^n$ whose support is entirely enclosed in E. The restriction is thus a method of defining a derived measure with support in a certain region. Every μ-measurable set is also ν-measurable and, provided E is measurable and $\mu(E) < \infty$, then ν is a Borel regular measure.

Some examples of frequently used used measures are given hereafter. The corresponding proofs that the described objects really are measures in the sense of definition 4.1 are omitted to keep this section as compact as possible. They are well known and contained in every book on measure theory.

Example 4.1 (Counting measure). For each subset A of $\mathbb{R}^n$, let $\mu(A)$ be the number of points in A if A is finite and ∞ otherwise. μ is a measure on $\mathbb{R}^n$.

Example 4.2 (Point measure). Let a be a point in $\mathbb{R}^n$. Define $\mu(A) = 1$ if $a \in A$ and $\mu(A) = 0$ if $a \notin A$. Then, μ is a measure with support $\{a\}$ that we think of as unit point mass concentrated at a.

Example 4.3 (Lebesgue measure). The Lebesgue measure on $\mathbb{R}^n$ is the natural extension to the terms "length", "surface area" and "volume". It is based on the "n-dimensional Volume" of the coordinate parallelepiped $A = \{(x_1, \ldots, x_n) \in \mathbb{R}^n : a_i \leqslant x_i \leqslant b_i\}$ defined by:

$$\mathrm{Vol}^n(A) = \prod_{i=1}^{n}(b_i - a_i)$$

Then the *n-dimensional Lebesgue measure* $\mathcal{L}^n$ is defined by:

$$\mathcal{L}^n(A) = \inf\left\{\sum_{i=1}^{\infty}\mathrm{Vol}^n(A_i) : A \subset \bigcup_{i=1}^{\infty} A_i\right\}$$

where the infimum is over all coverings of A by countable collections of parallelepipeds. With some effort it may be shown that $\mathcal{L}^n$ is indeed a Borel regular measure on $\mathbb{R}^n$ such that $\mathcal{L}^n(A)$ equals the n-dimensional volume of A if A is a parallelepiped or any other set for which the volume can be calculated using the usual rules of mensuration.

Later, integration with respect to a measure μ will also be used. Let $f : X \mapsto \mathbb{R}^+ \cup \{0\}$ be a non-negative *simple function*, i.e. a function that only takes a finite number of different values $a_1, \ldots, a_k \in \mathbb{R}^+ \cup \{0\}$. Then integration with respect to the measure μ on X is defined as:

$$\int f\, d\mu = \sum_{i=1}^{k} a_i \mu\{x : f(x) = a_i\}$$

Integration of more general functions is defined using approximation by simple functions. We term $f : X \mapsto \mathbb{R}$ a *measurable function* if for all $c \in \mathbb{R}$ the set $\{x \in X : f(x) < c\}$ is a measurable set. In particular for a Borel measure μ all continuous functions are measurable. The integral of any non-negative measurable function $f : X \mapsto \mathbb{R}^+ \cup \{0\}$ is defined by:

$$\int f\, d\mu = \sup\left\{\int g\, d\mu : g \text{ is simple, } 0 \leqslant g \leqslant f\right\}$$

This value may also be infinite. Finally, for a measurable function $f : X \mapsto \mathbb{R}$ that can also take negative values, the definition is completed writing $f_+(x) = \max\{f(x), 0\}$ and $f_-(x) = \max\{-f(x), 0\}$, so that $f(x) = f_+(x) - f_-(x)$, and defining:

$$\int f\, d\mu = \int f_+\, d\mu - \int f_-\, d\mu$$

provided both $\int f_+\, d\mu$ and $\int f_-\, d\mu$ are finite. For integrals over measures, the usual properties of integrals including their linearity and convergence theorems hold unchanged. If A is a Borel subset of X, the definite integral over $A \subset \mathbb{R}^n$ is defined by:

$$\int_A f\, d\mu = \int f\chi_A\, d\mu$$

where $\chi_A : \mathbb{R}^n \mapsto \mathbb{R}$ is the *indicator function* of A with $\chi_A(x) = 1$ if $x \in A$ and $\chi_A(x) = 0$ otherwise.

4.2 Hausdorff's measure and dimension

Having now the basic knowledge of measures and distributions at hand, the special Hausdorff measure can be introduced. This will lead to the definition of Hausdorff's dimension ([Hausdorff (1919)]) which is the oldest and most important metric dimension when studying fractal sets. This is due to the fact that it is defined for any set and is mathematically practical as it is based on a comparatively simple measure. Although [Sbalzarini (2001)] already gave a "free hanging" definition of Hausdorff's dimension, it is restated here based on the more fundamental and solid background of measure theory since it is important to be familiar with Hausdorff's measure and dimension to understand the mathematics of fractals.

First, recall the following two definitions from basic set theory:

Definition 4.4 (Diameter). The *diameter* of a non-empty subset U of the n-dimensional Euclidean space $\mathbb{R}^n$ is defined as:

$$|U| = \sup\{|x-y| : x, y \in U\}$$

This means that the diameter is the largest occurring Euclidean distance between any two points in U.

Definition 4.5 (δ-cover). A finite or countable collection of subsets $\{U_i\}$ of $\mathbb{R}^n$ is called a *δ-cover* of a set $E \subset \mathbb{R}^n$ if $0 < |U_i| \leqslant \delta \ \forall i$ and $E \subset \bigcup_{i=1}^{\infty} U_i$.

This means that E is completely covered by a countable collection of subsets of diameter less or equal δ. Now let E be a subset of $\mathbb{R}^n$ and s a non-negative number. For all δ we define the quantity:

$$\mathcal{H}^s_\delta(E) = \inf\left\{\sum_{i=1}^{\infty} |U_i|^s : \{U_i\} \text{ is a } \delta\text{-cover of } E\right\} \tag{4.1}$$

As δ decreases, the class of valid δ-covers of E is reduced. The infimum therefore increases and approaches a limit as $\delta \downarrow 0$. We write:

$$\mathcal{H}^s(E) = \lim_{\delta \to 0} \mathcal{H}^s_\delta(E) \tag{4.2}$$

This limit exists for all $E \subset \mathbb{R}^n$, although its value can be 0 or ∞. We term $\mathcal{H}^s(E)$ the *s-dimensional Hausdorff measure* of E. With considerable effort is has been shown that the Hausdorff measure is a measure in the sense of definition 4.1. It is even a Borel regular measure on $\mathbb{R}^n$.

Hausdorff's measure generalizes Lebesgue's measure, so that $\mathcal{H}^0(E)$ gives the number of points in E, $\mathcal{H}^1(E)$ the length of a smooth curve E, $\mathcal{H}^2(E)$ the (normalized) surface area of a region E, etc. In general, for every Borel subset E of $\mathbb{R}^n$, is holds:

$$\mathcal{H}^n(E) = c_n \mathcal{L}^n(E) \qquad \text{where} \qquad c_n = \frac{\pi^{\frac{n}{2}}}{2^n\left(\frac{n}{2}\right)!}$$

Often, the Hausdorff measure of the image of a set under a Hölder mapping is needed. Let $E \subset \mathbb{R}^n$ and $f : E \mapsto \mathbb{R}^m$ a *Hölder continuous*[3] function of order α, so that

$$|f(x) - f(y)| \leqslant c\,|x-y|^\alpha$$

for all $x, y \in E$ and some $c, \alpha > 0$.

[3] A function is called *Hölder continuous* of order α if $|f(x) - f(y)| \leqslant c\,|x-y|^\alpha \ \forall x, y$

Then

$$\mathcal{H}^{\frac{s}{\alpha}}(f(E)) \leqslant c^{\frac{s}{\alpha}}\mathcal{H}^s(E)$$

for all s. The proof of this can for example be found in [Falconer (1997)]. The special case of a *Lipschitz mapping* f (i.e. $\alpha = 1$) is particularly interesting. It also comprises the case when f is a regular *similarity transformation* of ratio λ. Then:

$$\mathcal{H}^s(\lambda E) = \lambda^s \mathcal{H}^s(E)$$

which generalizes the familiar scaling properties of length, area, volume, etc. to the fractal case. Also notice that Hausdorff's measure is invariant under isomeries[4], translation and rotation as could be expected for a generalized "volume".

It it easy to show from equations 4.1 and 4.2 that for all sets $E \subset \mathbb{R}^n$ there is a critical number D_H such that $\mathcal{H}^s(E) = \infty$ if $s < D_H$ and $\mathcal{H}^s(E) = 0$ if $s > D_H$. This number D_H is called *Hausdorff dimension* of the set E and can be compactly defined as follows:

Definition 4.6 (Hausdorff dimension). The number:

$$D_H = \inf\{s : \mathcal{H}^s(E) = 0\} = \sup\{s : \mathcal{H}^s(E) = \infty\}$$

is called *Hausdorff dimension* of E.

Thus D_H is the value of s for which the Hausdorff measure jumps from ∞ to 0 (see figure 4.1). When $s = D_H$, the Hausdorff measure can be either 0, ∞ or, in the nicest situation, $0 < \mathcal{H}^{D_H}(E) < \infty$.

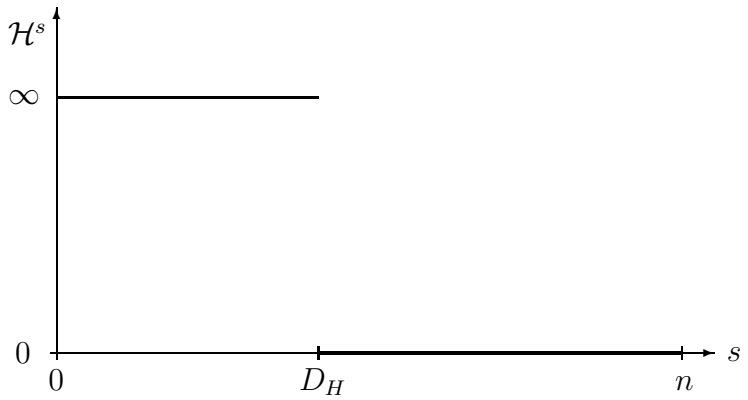


Figure 4.1: Behavior of the Hausdorff measure in the vicinity of D_H

Definition 4.7 (s-set). A set $E \subset \mathbb{R}^n$ of Hausdorff dimension D_H with $0 < \mathcal{H}^{D_H}(E) < \infty$ is called an *s-set*.

As an example, imagine a flat circular disk in $\mathbb{R}^3$ with radius 1. From known properties of length, area and volume, we get: $\mathcal{H}^1 = \text{length} = \infty$, $0 < \mathcal{H}^2 = \frac{\pi}{4}\text{area} < \infty$ and $\mathcal{H}^3 = \frac{\pi}{6}\text{volume} = 0$. Thus the disk is an s-set with Hausdorff dimension $D_H = 2$ which is consistent with the usual Euclidean definition of dimension.

[4]Mappings for which $|f(x) - f(y)| = |x - y| \ \forall x, y$

The transformation properties of Hausdorff's dimension follow from the ones of Hausdorff's measure. In particular, for a Hölder mapping $f : E \mapsto \mathbb{R}^m$ of order α we get:

$$D_H(f(E)) \leqslant \frac{1}{\alpha} D_H(E)$$

A fundamental property of Hausdorff's dimension is its invariance under *Bi-Lipschitz mappings*[5] as can easily be seen from above transformation property. This means that if two sets have different Hausdorff dimensions, there is no Bi-Lipschitz mapping between them. Analogous to topology theory where two homeomorphic sets are considered equivalent, two fractals are considered *equivalent* if they have the same Hausdorff dimension, i.e. there exists a Bi-Lipschitz continuous mapping between them.

Moreover, Hausdorff's dimension fulfills a whole set of properties that are natural to be expected from any meaningful definition of a dimension. [Falconer (1990)] for example contains an exhaustive list of them. However, their details are not relevant for this work and will not be stated here.

4.3 Other fractal dimensions

In addition to the very important Hausdorff dimension, there is an infinite number of other possible definitions of metric dimensions, the most important of whitch is the *box counting dimension* for this work. Although Hausdorff's dimension comes in very handy for theoretical considerations, it has a fundamental drawback: it is not measurable or computable in most cases. The box counting dimension however is quite easy to be estimated in computer simulations what makes it a valuable tool for the investigations to come.

It is however important to keep in mind that fractals are *dimensionally discordant sets* (cf. [Sbalzarini (2001)]), so that different meaningful definitions of dimension can yield different numerical values for the same set and even "very similar" definitions can have grossly different properties. A common property of all dimensions is that they are based on "measurements with respect to a scale δ" in a metric space. For a large class of dimensions, a power law between the measured length in multiples of δ (N_δ) and the metric size of δ is then assumed:

$$N_\delta(E) \sim c\delta^{-s}$$

The exponent s is called *dimension of* E and the constant c is referred to as *s-dimensional length of* E. Using log-log-plots or statistical fitting allows to estimate s from a number of measurements with different length scales δ. This way of defining dimension is not based on measure theory which makes it awkward for theoretical usage. It is however consistent with measure-based dimensions in the sense that the numerical values for dimensionally concordant sets are equal. It happens however, that the assumption of a power law is in fact wrong, in which case the points in a plot of $\log(N_\delta)$ vs. $\log(\delta)$ will not be on a straight line. Dimensions that are based on the power law assumption are then meaningless for the given set and should not be used. This emphasizes the need to not blindly apply linear regression fitting to such a data set but look at the distribution of points as well.

Moreover, when defining new dimensions, one should make sure that they meet all the requirements for "meaningful" dimensions such as monotony, stability, Bi-Lipschitz invariance, etc. as for example listed in [Falconer (1990)].

[5] A function $f : E \mapsto \mathbb{R}^m$ is called *Bi-Lipschitz* if $c_1|x-y| \leqslant |f(x)-f(y)| \leqslant c_2|x-y|$ for all $x, y \in E$ and certain $0 < c_1 \leqslant c_2 < \infty$.

The power law dimensions that will be used in this work are the box counting dimension and the interior Minkovski dimension that will be described hereafter. Another useful dimension is the *packing dimension* which is related to Hausdorff's dimension but instead of covering the set with δ-subsets, it is packed with disjoint δ-balls. It will however not be used in this text and therefore its description is omitted.

4.3.1 The box counting dimension

The box counting dimension has first been used in the 1930s. Due to its popularity, a large number of different names for it exist, among them Kolmogorov entropy, Pontrjagin-Kolmogorov dimension (as used in [Sbalzarini (2001)]), Entropy dimension, capacity dimension, logarithmic density or information dimension. To support intuition, the term "box counting dimension" will be used in this text.

For E being a non-empty bounded subset of $\mathbb{R}^n$ let $N_\delta(E)$ be the smallest number of sets of diameter $\leqslant \delta$ that can cover E. The *lower* and *upper box counting dimensions* of E are defined as:

$$\underline{D_B} = \liminf_{\delta \to 0} \frac{\log N_\delta(E)}{-\log \delta}$$

and

$$\overline{D_B} = \limsup_{\delta \to 0} \frac{\log N_\delta(E)}{-\log \delta}$$

respectively. If these are equal, the common value is simply called *box counting dimension* of E:

$$D_B = \lim_{\delta \to 0} \frac{\log N_\delta(E)}{-\log \delta} \tag{4.3}$$

Fortunately, there are a number of equivalent forms of this definition which are practically useful. The values of above limits namely remain unaltered if $N_\delta(E)$ is taken to be any of the following:

1. the smallest number of sets of diameter $\leqslant \delta$ that can cover E
2. the smallest number of closed balls of radius δ that can cover E
3. the smallest number of cubes of edge length δ that can cover E
4. the largest number of disjoint balls of radius δ with centers in E
5. the number of δ-mesh cubes[6] that intersect E

Variant 5 is the one that has been used in [Sbalzarini (2001)] and will also be used in the present work since the mesh needed for it is naturally given by the pixels in a 2D image or their 3D equivalent, the voxels, in a 3D volume.

[6] a δ-mesh cube is a cube of the form $[m_1\delta,\ (m_1+1)\delta) \times \ldots \times [m_n\delta,\ (m_n+1)\delta)$ where $m_1, \ldots, m_n$ are integers.

Although the box counting dimension is defined in a completely different way as the Hausdorff dimension, there is a deterministic relationship between them as stated in the following inequality:

$$D_H \leqslant \underline{D_B} \leqslant \overline{D_B} \tag{4.4}$$

This is due to the fact that the box counting dimension is (in variants 2 to 5) based on covering the set with elementary subsets (balls, cubes, etc.) of *equal and fixed* size whereas the Hausdorff dimension relies on a δ-cover with subsets of *different size smaller than* δ. Therefore, Hausdorff's dimension employs a finer covering with the biggest subset having the size of the ones used for the box counting.

In practice, most definitions of dimensions take values between the Hausdorff and upper box counting dimension, so if it can be shown that $\overline{D_B} = D_H$ then all common definitions of dimension will take this common value.

For a complete discussion of advantages and disadvantages of the box counting dimension, see [Falconer (1990)] and [Falconer (1997)]. For our purposes it is sufficient to state that the box counting dimension is unusable for sets that only consist of a countable number of isolated points[7]. Since the ER and its triangulated description is a continuous set with no isolated points, the box counting dimension can be used without danger.

4.3.2 The Minkovski dimension

A variant of the box counting dimension that is sometimes used for one-sided closed sets is the Minkovski dimension. It is based on the n-dimensional volume of the *δ-neighborhood* or *δ-parallel body* E_δ of E given by:

$$E_\delta = \{x \in \mathbb{R}^n : |x - y| \leqslant \delta \text{ for all } y \in \partial E\}$$

Then for $E \subset \mathbb{R}^n$:

$$\underline{D_M} = \underline{D_B} = n - \limsup_{\delta \to 0} \frac{\log \mathcal{L}^n(E_\delta)}{\log \delta}$$

$$\overline{D_M} = \overline{D_B} = n - \liminf_{\delta \to 0} \frac{\log \mathcal{L}^n(E_\delta)}{\log \delta}$$

and if they coincide:

$$D_M = D_B = n - \lim_{\delta \to 0} \frac{\log \mathcal{L}^n(E_\delta)}{\log \delta}$$

These definitions are actually equivalent to the ones of the box counting dimension. In the context of using the Lebesgue measure they are however referred to as *Minkovski dimensions*.

Important variants are the *inner* and *outer Minkovski dimension* for which only the inner or outer δ-neighborhood of a closed set is taken. This is for example useful to study the influence of the fractality of the boundary of a domain on the solution of a PDE solved in its interior (cf. section 6.2.7).

[7] this is due to the fact that the closure of a set E has the same box counting dimension as the set E itself

Chapter 5

Measuring the fractal dimension

Now having the necessary preliminaries on the notion of dimensions, the fractal dimension of the ER surface in $\mathbb{R}^3$ will be measured. As stated in [Sbalzarini (2001)], this dimension could be an important parameter for any geometry model. It will be shown later in this report that equality of the dimension is indeed a necessary condition for a geometrical model to be valid. In principle, one would wish to measure Hausdorff's dimension. However, this is not possible for arbitrary geometries (cf. section 4.3) and therefore the box counting dimension as defined in section 4.3.1 will be used instead.

A box counting algorithm similar to the one described in [Sbalzarini (2001)] requires the data to be in some sort of "box oriented form". For 2D images, these "box units" are naturally given by the image's pixels. The triangulated 3D objects dealt with now however pose some problems. According to section 4.3.1 there are various equivalent ways of defining the box counting dimension. One possibility would for example be to successively discard vertices of the triangulation at hand and retriangulate the surface at every step. Due to topological limitations (recall that the ER's surface has to be contiguous) this would be not only cumbersome but also dangerous and geometry checking and possibly fixing would be needed at every step. It is therefore easier to use, again, the "pixel" oriented approach. The 3D analogue of a pixel is called a *voxel* and can be defined as a finite parallelepiped in $\mathbb{R}^3$, thus:

$$V_{i,j,k} = \{(x,y,z) : (i-1)\delta x \leqslant x - x_l < i\delta x,\ (j-1)\delta y \leqslant y - y_l < j\delta y,\\ (k-1)\delta z \leqslant z - z_l < k\delta z\}$$

where (x_l, y_l, z_l) are the coordinates of the lower left corner of the bounding box. This is an extension of what has been called a *δ-mesh cube* in section 4.3.1 to an anisotropic mesh. Hence the first step will be a conversion of the triangulated surface to a voxel set.

5.1 Generating a voxel representation

Recall that variant 5 of the definition of the box counting dimension in section 4.3.1 states that all voxels that intersect the geometry in question are to be counted. The ER surface is given as a triangulated set $\aleph$ as defined in [Sbalzarini (2001)], meeting

all the requirements stated in section 3.1. The space $\mathbb{R}^3$ in which this triangulation is embedded is covered with a regular cartesian grid with grid spacings δx, δy and δz defining the voxels. All voxels that are intersected by $\aleph$ are then set to one, the others to zero thus yielding the required binary voxel representation. The algorithm proceeds as follows:

Algorithm 5.1 (Voxelize).

Step 1: Generate cartesian grid with spacings δx, δy, δz and determine the bounding box $[x_l, x_h] \times [y_l, y_h] \times [z_l, z_h]$ of the geometry as: $\{x, y, z\}_l = \min_p\{x_p, y_p, z_p\}$,
19-36 $\{x, y, z\}_h = \max_p\{x_p, y_p, z_p\}$ where p loops over all triangle vertices in $\aleph$.

Step 2: Define voxels $V_{i,j,k} = [x_l + (i-1)\delta x,\ x_l + i\delta x) \times [y_l + (j-1)\delta y,\ y_l + j\delta y) \times [z_l + (k-1)\delta z,\ z_l + k\delta z)$ for $i, j, k = 1, 2, 3, \ldots$

33 Step 3: Initialize all voxels to zero: $V_{i,j,k} = 0 \quad \forall\,(i, j, k)$

Step 4: Loop over all triangles $\Delta \in \aleph$

4.1: For each vertex P of Δ determine the voxel it is contained in: $(i, j, k) = \mathrm{int}((P_x - x_l)/\delta x,\ (P_y - y_l)/\delta y,\ (P_z - z_l)/\delta z) + (1, 1, 1)$ and set voxel
59-69 (i, j, k) to one[1]

4.2: Determine the bounding box of voxels for Δ: $(i_u, j_u, k_u) = \max(i, j, k)$
70-75 and $(i_l, j_l, k_l) = \min(i, j, k)$

4.3: For each voxel (i, j, k) in the bounding box $\{i_l \leqslant i \leqslant i_u,\ j_l \leqslant j \leqslant j_u,\ k_l \leqslant k \leqslant k_u\}$ solve:

$$\begin{bmatrix} | & | & | \\ a & b & n \\ | & | & | \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} | \\ c_V - P_1 \\ | \end{bmatrix}$$

78-90

for α, β and λ where $a = P_2 - P_1$ and $b = P_3 - P_1$ are two edge vectors of the triangle Δ, n is its outer normal and P_1 the location vector of vertex 1 of triangle Δ. c_V is the location vector of the centroid of voxel (i, j, k), defined as: $c_V = [x_l + (i - 1/2)\delta x,\ y_l + (j - 1/2)\delta y,\ z_l + (k - 1/2)\delta z]^\top$

4.4: Set voxel (i, j, k) to one if $\alpha \geqslant 0$ and $\beta \geqslant 0$ and $\alpha + \beta \leqslant 1$ and $Y = c_V + \lambda n \in V_{i,j,k}$ which means that the centroid of voxel (i, j, k) is over the triangle Δ and the intersection point of the normal through the centroid
91-93 of the voxel with the triangle is inside the voxel. (see also figure 5.1)

An implementation of this algorithm can be found in the subroutine `Voxelize.f90` in appendix C.5.4.12. A few remarks are in place. The margin numbers refer to the corresponding statements in the Fortran code. In step 4.3, a linear system of equations is to be solved. To speed up the algorithm, this has been done symbolically
48-58 in advance and the general solution is programmed directly.. The condition in step
88-90 4.4 can also be taken to be $\lambda \leqslant \frac{1}{2}\max\{\delta x, \delta y, \delta z\}$. This however turned out to be problematic if the voxels are elongated parallelepipeds rather than cubes. For near cubic voxels and visualization purposes this condition is nevertheless sometimes preferable since it yields nicer images containing less voxels. For the box counting algorithm it is however useless. Figure 5.1 illustrates the condition of step 4.4.

[1] Special index treatment is needed for the rightmost voxel.

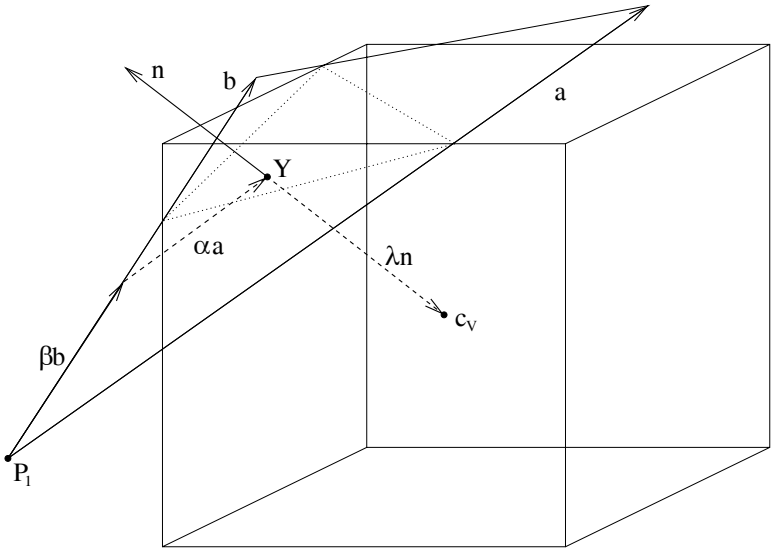


Figure 5.1: Geometric situation of a voxel and a triangle

Validation

The algorithm has been tested for various simple shapes. As an example, figure 5.3 shows a voxel representation of the pyramid depicted in figure 5.2. Figure 5.4 shows a sample picture of a triangulated ER surface and figure 5.5 its voxel representation. The visualizations are made using the freely available OpenDX package (former IBM DataExplorer)[2].

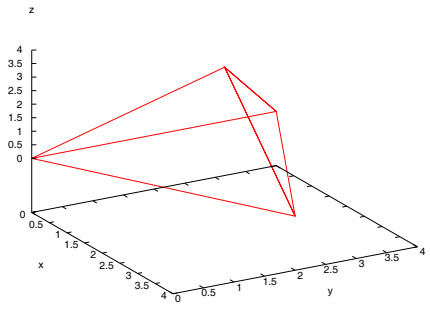


Figure 5.2: Pyramid wireframe model

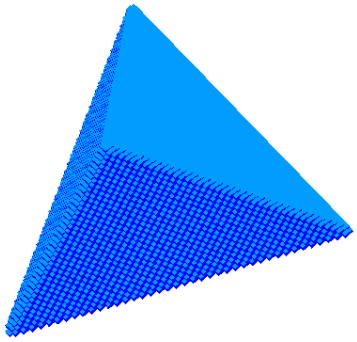


Figure 5.3: Voxel representation

5.2 A 3D box counting algorithm

Now having a voxel set representation $\{(i,j,k) : V_{i,j,k} = 1\}$ for the triangulated set $\aleph$, the determination of the box counting dimension according to section 4.3.1 is straightforward. Algorithms for 2D images exist for quite some time (see e.g. [Liebovitch & Toth (1989)]) however the first application to 3D systems has been

[2] http://www.opendx.org

Figure 5.4: Triangulated surface

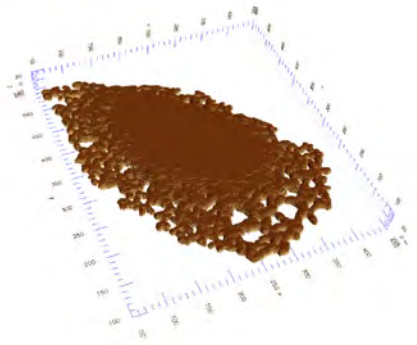


Figure 5.5: Voxel representation

done by [Stoll, Stern & Stucki (1996)], who unfortunately did not publish the algorithm as such. To ensure comparability of results and since we are dealing with triangulated sets rather than point sets, the same procedure as in [Sbalzarini (2001)] is used here but generalized to three dimensions. For details on why this method works or why outlining and low-pass filtering are needed, the reader is referred to that report. The algorithm is as follows:

Algorithm 5.2 (Box counting).

Step 1: Convert the triangulation $\aleph$ to a voxel representation with voxels of size δx, δy, δz using algorithm 5.1.

37

Step 2: Determine the number of resolution reduction steps until the smallest dimension has less than 16 voxels: $n = \text{ceiling}\left(\log\left(\min\left\{\delta x, \delta y, \delta z\right\}/16\right)/\log 2\right)$.

42

Step 3: For each reduction step $i = 1$ to n

3.1: Outline the voxel set: set all voxels to zero that are completely surrounded by voxels of value 1.

51-78

3.2: Count the number of voxels of value one: $\eta_i = \#\{(i,j,k) : V_{i,j,k} = 1\}$ and store the edge length scale of the current voxels: $\sigma_i = \delta x/2^{i-1}$

79-80

3.3: Discrete low-pass filter: replace each voxel by the average of its 26 neighbors and itself.

100-115

3.4: Binarize set: set all voxels with value $<$ threshold to zero, all others to one.

120

3.5: Resolution reduction: delete every second row, column and z-plane from the 3D voxel set.

97-123

3.6: Reallocate voxel array to new size and update all variables.

124-146

147-148 Step 4: Get final values: $\eta_{n+1} = \#\{(i,j,k) : V_{i,j,k} = 1\}$ and $\sigma_{n+1} = \delta x/2^n$

149-152 Step 5: Let $y_i = \log \eta_i$ and $s_i = \log\left(\frac{1}{\sigma_i}\right)\ \forall\ i = 1, 2, \ldots, n+1$

158-169 Step 6: Do a linear least squares fit of y_i vs. s_i to get the slope:

$$\hat{a} = \sum_{i=1}^{n+1}\left((n+1)s_i - \sum_{i=1}^{n+1} s_i\right) y_i$$

Step 7: Deallocate all memory and return the estimated box counting dimension: $D_B = \hat{a}$ 177-181

This algorithm is implemented as a Fortran function in `BCdim.f90` (see appendix C.5.4.13, to which the statement numbers in above algorithm refer) which directly returns the numerical value of the box counting dimension. With considerable effort it could be optimized according to [Hou, Gilmore, Mindlin & Solari (1990)]. However it turned out that its performance is sufficient for the present application so that one can do without further optimization. A few remarks to the algorithm are useful: In step 2, the number of reduction steps is calculated until the smallest occurring number of voxels in a row is less than 16. This is needed since for smaller numbers the accuracy of the voxel representation deteriorates rapidly. Moreover, the definition of the box counting dimension in equation 4.3 contains the limit of $\delta \to 0$ which means that too large voxels are not significant anyway. The outline operation in step 3.1 removes all voxels that are not part of the hull. Since we are considering a surface in space, this is needed to keep the geometry from "thickening" towards a space-filling body which would grossly distort the value of its dimension. The way this outlining is done is a 3D generalization of MATLAB's `bwperim` command. In step 3.4 the set is binarized again. The threshold used here is a user parameter and should be chosen as high as possible such that the resulting dimensions are valid (i.e. smaller than the topological dimension of the space the object is embedded in). Some authors (e.g. [Liebovitch & Toth (1989)]) state that it is advisable to exclude the first few measurement points (i.e. the ones at the finest resolution level) from the least squares fit since they may contain artifacts and measurement noise. In our case however this turned out to be not necessary since the initial voxel representation is generated in the computer (thus free of noise) and already is quite coarse (due to memory limitations). Moreover, the measurement points all lie on a straight line pretty well. This means that there are no inaccurate points that should be excluded and therefore they are kept in order to have more measurement points available.

The least squares fit in step 6 is done according to the following calculation. Let $Y = [y_i]^\top \in \mathbb{R}^{n+1}$ be the vector of the logarithms of the measured voxel numbers and $\Theta = [a\ b]^\top \in \mathbb{R}^2$ the vector of unknown parameters. According to equation 4.3 we now assume a linear relation between the logarithm of the voxel number and the logarithm of the corresponding inverse voxel size: $y_i = as_i + b$. Since this will not be exactly true for all i, we define the residual $\varepsilon_i = as_i + b - y_i$ and require the sum over all i of its squares to be minimum. Defining the matrix

$$\Phi = \begin{bmatrix} s_1 & s_2 & \dots & s_{n+1} \\ 1 & 1 & \dots & 1 \end{bmatrix}^\top$$

the whole problem can be compactly formulated for all i using matrix vector notation:

$$\begin{aligned} Y &= \Phi\Theta \\ \varepsilon &= \Phi\Theta - Y \\ \varepsilon^\top\varepsilon &\stackrel{!}{=} \min \end{aligned}$$

Inserting the definition of ε into the last equation gives:

$$\varepsilon^\top\varepsilon = (\Phi\Theta - Y)^\top(\Phi\Theta - Y) = \Theta^\top\Phi^\top\Phi\Theta - \Theta^\top\Phi^\top Y - Y^\top\Phi\Theta + Y^\top Y$$

The minimization condition requires the derivative with respect to Θ to vanish, thus:

$$\frac{\partial}{\partial \Theta}\left(\varepsilon^\top \varepsilon\right) = 2\Phi^\top \Phi\Theta - 2\Phi^\top Y \stackrel{!}{=} 0$$

Solving this equation for Θ yields the least squares estimate:

$$\widehat{\Theta}_{LS} = \left(\Phi^\top \Phi\right)^{-1} \Phi^\top Y$$

This is however still not very useful for our purpose. Explicitly inserting the definition of Φ gives:

$$\Phi^\top \Phi = \begin{bmatrix} \sum s_i^2 & \sum s_i \\ \sum s_i & n+1 \end{bmatrix} \in \mathbb{R}^{2\times 2}$$

where all the sums are taken over $i = 1, \ldots, n+1$. Its inverse is then given by:

$$\left(\Phi^\top \Phi\right)^{-1} = \frac{1}{\Delta} \begin{bmatrix} n+1 & -\sum s_i \\ -\sum s_i & \sum s_i^2 \end{bmatrix} \in \mathbb{R}^{2\times 2}$$

where the determinant Δ is given by:

$$\Delta = (n+1)\sum s_i^2 - \left(\sum s_i\right)^2$$

Using this allows to explicitly calculate the least squares estimates of the unknown line parameters a and b as:

$$\hat{a} = \sum_{i=1}^{n+1} \left((n+1)s_i - \sum_{i=1}^{n+1} s_i \right) y_i$$
$$\hat{b} = \sum_{i=1}^{n+1} \left(\sum_{i=1}^{n+1} s_i^2 - s_i \sum_{i=1}^{n+1} s_i \right) y_i$$

The slope $\hat{a}$ approximates the box counting dimension D_B according to its definition in equation 4.3.

Validation

The algorithm as presented above is now validated using some simple Euclidean shapes. The first case that is presented here consists of two plane parallel triangles[3] (see figure 5.6) and the second of a cubic box of edge length 4 (see figure 5.7).

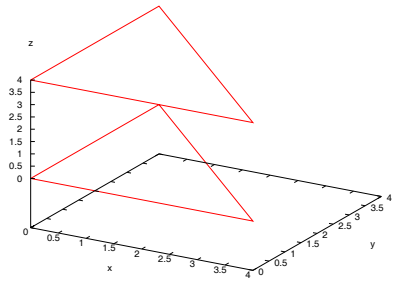


Figure 5.6: Two parallel triangles

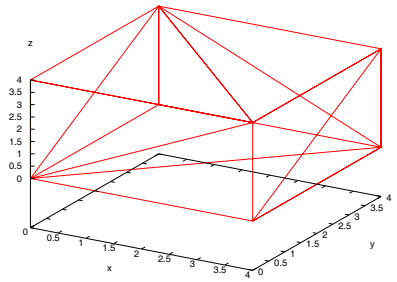


Figure 5.7: Cubic box

[3] because a single triangle would not be a 3D object

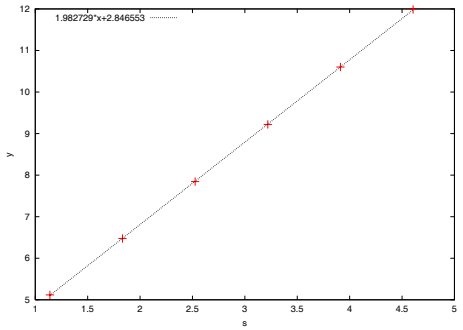


Figure 5.8: $\log \eta$ vs. $\log(1/\sigma)$ (red crosses) for the triangles and its least squares fit (black dotted)

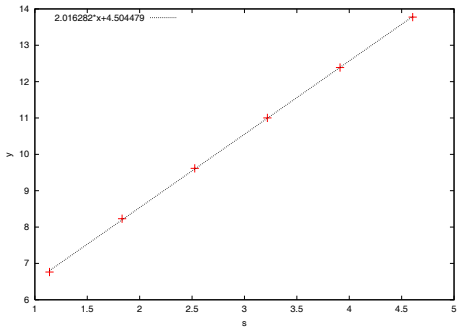


Figure 5.9: $\log \eta$ vs. $\log(1/\sigma)$ (red crosses) for the box and its least squares fit (black dotted)

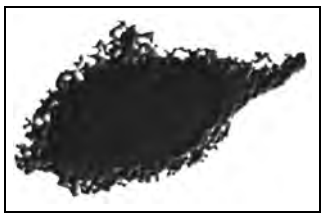
Figure 5.8 shows the result for the triangles. The measurement points of the box counting algorithm are indicated as red crosses and the least squares fit as a dotted black line. As expected for a dimensionally concordant set, the crosses lie on a straight line since the power law assumption is exactly valid (cf. section 4.3). The dimension of the triangles is measured to be 1.9827. For the box we get 2.0163 and the points in figure 5.9 are no longer exactly on the straight line due to the higher complexity of the shape. (but they are still very close to it.)

As it is known that both shapes are of dimension 2.0 for all meaningful definitions of dimension (recall that we just consider their surface, not the volume), it can be stated that the box counting algorithm yields the correct results up to one decimal digit.

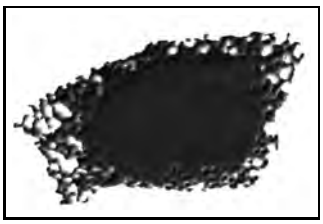
5.3 Results for the ER

The box counting algorithm described so far is now used to estimate the dimension of the surface of the ER. It will then be possible to compare it to the theoretical prediction made in [Sbalzarini (2001)] on page 64 that the dimension should be between 2.8 and 3.0 with a higher probability for the lower bound.

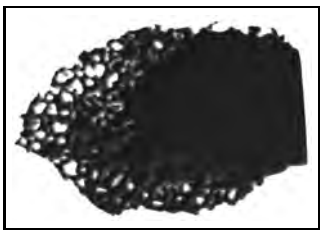
Nine different triangulated 3D reconstructions of ER shapes are considered which allows some statistical averaging. The following figures show views of these nine samples together with their names.



bip2



clx



erp57

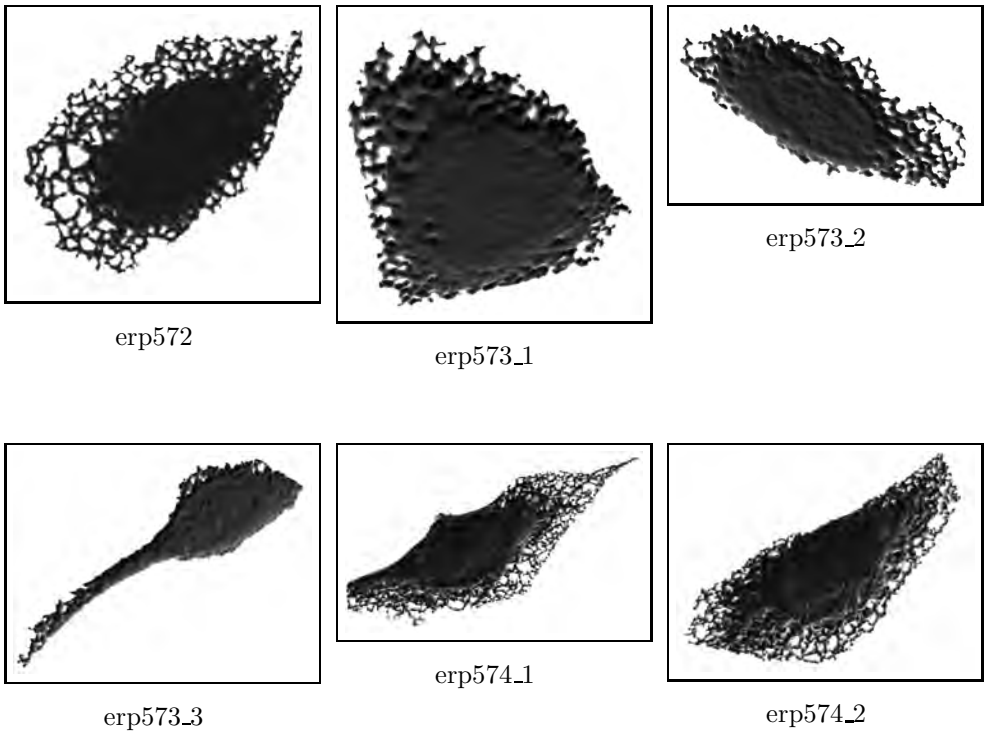


Figure 5.10: Shaded surface views of all reconstructed ER samples.

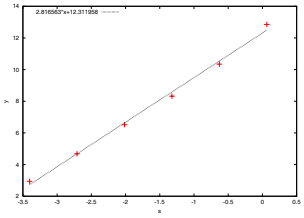
The threshold for the binarization step (step 3.4 in algorithm 5.2) is chosen according to the rules in [Sbalzarini (2001)]. The largest value still yielding meaningful results turns out to be 0.3 for most samples, 0.29 for erp573_2 and erp574_2 and 0.27 for erp573_3. Choosing only slightly larger values causes the measured dimension to jump up to values larger than 3 which is impossible due to the Szpilrajn inequality. The exceptions are caused by variations in the reconstruction of the micrographs. Thus the right threshold seems to be around 0.3. The following values are obtained for the different samples:

	0.3	0.29	0.27
bip2	**2.8166**	2.5655	
clx	**2.7195**	2.4906	
erp57	**2.4839**	2.3379	
erp572	**2.9876**	2.6684	
erp573_1	**2.9554**	2.5391	
erp573_2	3.4788	**2.9704**	
erp573_3	>4	3.2033	**2.9960**
erp574_1	**2.8836**	2.5648	
erp574_2		**2.5628**	

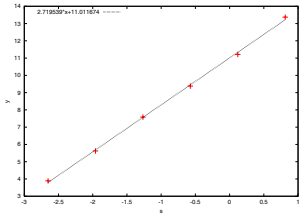
Table 5.1: Measured box counting dimensions

The numbers in bold are the ones for the highest threshold still giving meaningful results. These are the estimated dimensions. Their average value is 2.8195 which is indeed between 2.8 and 3.0 and closer to 2.8, as predicted by the fractal projection theorem in [Sbalzarini (2001)]. The final result to one significant decimal is thus: $\boldsymbol{D_B = 2.8}$.

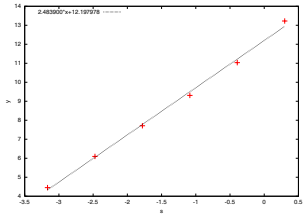
Figure 5.11 finally shows the plots of $\log \eta$ vs. $\log(1/\sigma)$ for all samples. The red crosses mark the measurement points of the reduction steps and the dotted lines the least squares fit. The corresponding numerical values of the box counting dimension are given along with the captions of the sub-figures. It can be seen that the crosses lie on the lines pretty well such that the power law assumption seems to be valid. The box counting dimension thus is a meaningful dimension for the ER and a good estimate of Hausdorff's dimension.



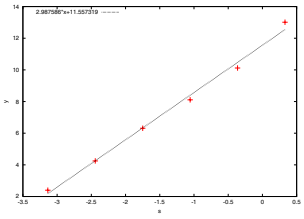
bip2: D_B=2.8166



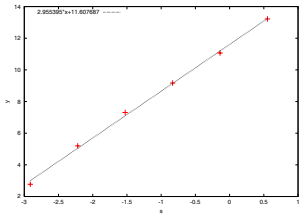
clx: D_B=2.7195



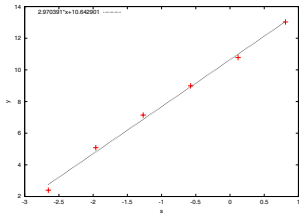
erp57: D_B=2.4839



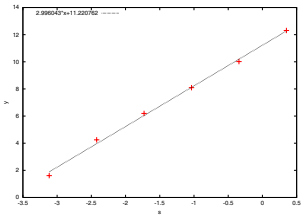
erp572: D_B=2.9876



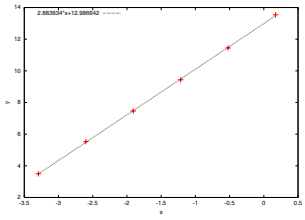
erp573_1: D_B=2.9554



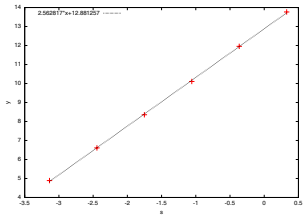
erp573_2: D_B=2.9704



erp573_3: D_B=2.9960



erp574_1: D_B=2.8836



erp574_2: D_B=2.5628

Figure 5.11: $\log \eta$ vs. $\log(1/\sigma)$ (red crosses) for all ER samples and their least squares fits (black dotted). The numerical value of the box counting dimension is given in the caption of each sub-figure.

Chapter 6

Diffusion on fractal sets

6.1 Preliminaries and definitions

In [Sbalzarini (2001)] it was suspected that the fractal dimension of the ER could be a defining geometrical property in the sense that a diffusion process on a model geometry of equal dimension would exhibit "the same" behavior. The quotation marks are there due to the fact that not point-wise identity is required but rather equal time behavior of diffusion, since this is the quantity that is actually measured in a FRAP curve. Model geometry and "real" ER should therefore yield the same FRAP curve when starting from the same initial condition.

In mathematical terms this statement means that two equivalent (in the sense of section 4.2) fractals are suspected to exhibit the same time constant of the solution of the diffusion equation or, in other words, the same eigenvalues of the Laplacian found as exponents in their heat kernel.

To investigate this assertion, some knowledge of the theory of partial differential equations on fractal sets is needed. The PDE to be considered is the heat equation:

$$\frac{\partial u}{\partial t} = D\nabla^2 u$$

which also governs diffusion. Moreover, one must define which dimension the above statement refers to as there is a multitude of different definitions for fractal dimensions. Here, Hausdorff's dimension D_H as defined in section 4.2 is used since a wealth of literature exists for it and its corresponding measure.

All fractal dimensions quantify some sort of "space filling ratio" for the geometrical object in question. For a surface in $\mathbb{R}^3$ it gives an intuitive measure for how much of the volume is filled up by the surface. It is therefore intuitive to assume that this dimension determines diffusion properties of the surface as the diffusion flux increases with increasing surface area per volume. A very porous hot cube with a rugged surface looses its heat faster than a solid cube of same size with smooth surface. Equality of the fractal dimension in general and Hausdorff's dimension in particular is therefore a necessary condition for the equality of time constants. But is it also sufficient?

To address this question, two preliminary definitions are needed. Fractal objects can be subdivided into two main classes. Imagine a random walk on the fractal. If the set of possible destination points for every step of Brownian motion is finite, the fractal is called *finitely ramified*. If the random walker can choose among an infinite number of places to visit within the next step, it is called *infinitely ramified*. These loose definitions can be made mathematically rigorous as follows:

Definition 6.1 (Finitely ramified). A set for which the number of possible different paths between any two points of the set is finite or countable is called *finitely ramified*.

Definition 6.2 (Infinitely ramified). A set for which the number of possible different paths between any two points of the set is infinite is called *infinitely ramified*

6.2 Sufficiency of Hausdorff's dimension

Consider two different fractal sets of equal Hausdorff dimension, e.g. Koch's curve (see figure 6.1) and the cartesian product of two Cantor sets (figure 6.2) that both have $D_H = \log 4/\log 3$. Looking at the two sets, it is hard to believe that diffusion processes on them should have equal time constants. Already the fact that Koch's curve is completely connected whereas the 2D Cantor set consists of individual disconnected areas gives rise to the suspicion that equality of Hausdorff's dimension may not be sufficient.

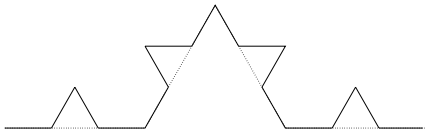


Figure 6.1: Koch curve at generation 2

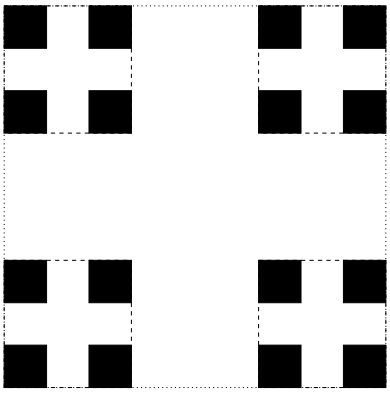


Figure 6.2: 2D Cantor set at generation 2

To disprove that Hausdorff's dimension is sufficient, it is enough to find a single counter example. For reasons of simplicity, the following will be shown for finitely ramified fractals. The results obtained can then be generalized to the more complex class of infinitely ramified fractals that the ER belongs to. The Sierpinski gasket has good enough regularity and connectivity properties to allow reasonable progress. Therefore, it will serve as example.

First, the definition of Brownian motion on the Sierpinski gasket is considered, which then leads to the heat kernel of the diffusion equation, the Laplacian and its eigenvalues. The following arguments closely follow the ones in [Falconer (1997)].

6.2.1 Brownian motion on the Sierpinski gasket

First, some notation needs to be introduced which is best done by recalling the definition of Brownian motion on $\mathbb{R}$. One way to define Brownian motion is as a limit of suitably scaled random walks. Let $X_k(t)$ be the random walk on the real set $\{j2^{-k} : j \in \mathbb{N}_0^+\}$ starting with $X_k(0) = 0$ and taking steps at time intervals of 4^{-k}. Given the position $X_k(m4^{-k})$ of the walker at time $m4^{-k}$, his position $X_k((m+1)4^{-k})$ at time $(m+1)4^{-k}$ is equally likely to be $X_k(m4^{-k}) - 2^{-k}$ or $X_k(m4^{-k}) + 2^{-k}$. It may be shown that as $k \to \infty$ (infinitely small steps in infinitely small time intervals; cf. also [Sbalzarini (2001)], p. 21), the sequence of random

walks $X_k(t)$ converges to a continuous process $X(t)$ on $\mathbb{R}$ named *one-dimensional Brownian motion*. Notice that scaling the base of the time scale as two times the base of the length scale is essential for convergence to a non-degenerate process. Thus if k is sufficiently large, $X_k(t)$ and $X(t)$ look very similar on all but the finest scales and the increments of Brownian motion, $X(t+\delta t) - X(t)$, are normally distributed with mean 0 and variance $2\delta t$ for all t and $\delta t > 0$. Typically, the motion travels a distance of $\sqrt{\delta t}$ in a time interval of duration δt. Moreover, Brownian motion has independent increments, that is, there is no historical memory of the path[1]. More generally, Brownian motion on $\mathbb{R}^n$ may be constructed as the limit of scaled random walks on n-dimensional cubic lattices.

We now attempt to mimic this construction of Brownian motion on the Sierpinski gasket E following the findings of [Barlow & Perkins (1988)]. The Sierpinski gasket is a fractal of Hausdorff dimension $D_H = \log 3/\log 2 \approx 1.585$ embedded in the Euclidean $\mathbb{R}^2$ (see figure 6.3). To avoid the need of regarding the three corner vertices of the bounded Sierpinski gasket as exceptional, consider the *extended Sierpinski gasket*, thus E extends outward to infinity using self-similarity as shown in figure 6.4. There is a natural sequence of Euclidean graphs E_0, E_1, ... that approximate the extended Sierpinski gasket (see figures 6.4 to 6.6). These graphs are called *pre-Sierpinski gaskets* and it holds: $E_0 \subset E_1 \subset E_2 \subset \ldots$ and $E = \bigcup_{k=0}^{\infty} E_k$.

Waclaw Sierpinski, 1882-1969

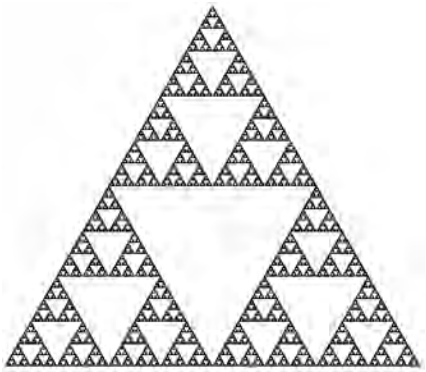


Figure 6.3: Sierpinski gasket E

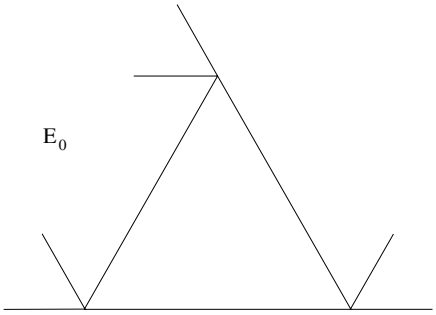


Figure 6.4: Pre-Sierpinski gasket of order 0

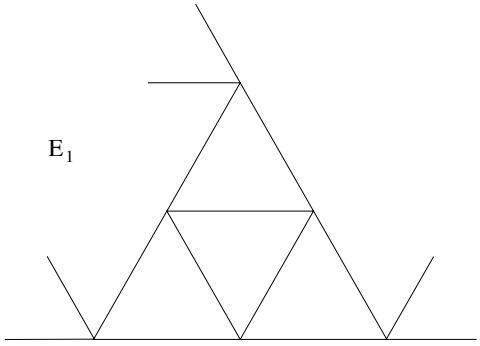


Figure 6.5: Pre-Sierpinski gasket of order 1

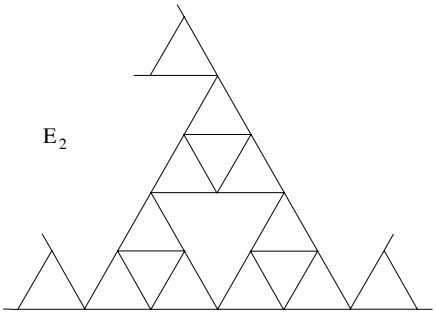


Figure 6.6: Pre-Sierpinski gasket of order 2

[1]Such a random process is called a *Markov chain*.

Let V_k be the set of vertices of E_k. The graph of E_k has edges of length 2^{-k} and each vertex is adjacent to four others. For $k = 0, 1, 2, \ldots$ the vertices of V_{k+1} are obtained by augmenting V_k by additional vertices at the midpoints of the edges of E_k, with appropriate additional edges added to form E_{k+1}.

Now define random walks $X_k(t)$ on V_k by traveling along the edges of E_k, taking steps at time intervals of α_k (where α_k is to be specified) and starting at $X_k(0) = 0$. Thus if $X_k(m\alpha_k)$ is the vertex of V_k occupied by the random walker at time $m\alpha_k$, then $X_k((m+1)\alpha_k)$ is one of the four vertices of V_k adjacent to $X_k(m\alpha_k)$ in E_k chosen with equal probability $\frac{1}{4}$, independent of all previous steps.

For $k \geqslant 1$, this random walk $X_k(t)$ on E_k induces a random walk on E_{k-1}[2]. By the symmetry of E_k there is equal probability of moving to each of the four adjacent vertices of V_{k-1}, so this induced random walk is just $X_{k-1}(t)$ undertaken with steps of varying time interval. For the random walks $X_k(t)$ to have a chance of converging to a reasonable limiting process, we should ideally choose the time intervals α_k so that, for each k, the time for $X_k(t)$ to move from a vertex of V_{k-1} to a neighboring vertex of V_{k-1} is α_{k-1}. This can however only be achieved *on average* by ensuring that the *expected time* of such a step is α_{k-1}.

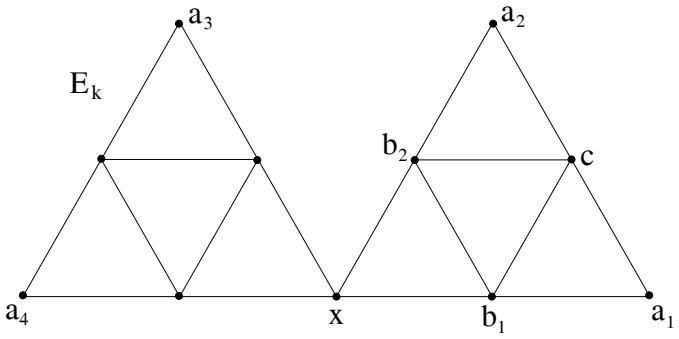


Figure 6.7: Situation of the random walk on E_k from x

Consider a portion of E_k near vertex x. By the symmetry of E_k, this is always equivalent to that shown in figure 6.7, where $A = \{a_1, a_2, a_3, a_4\}$ and b_1, b_2 and c are as indicated. Write $\mathsf{E}(p, A)$ for the expected number of steps in a random walk on E_k to get from a point $p \in V_k$ to any point of A. Given that the random walker starts at x, symmetry allows us to assume without loss of generality that the first step is to b_1 when determining $\mathsf{E}(x, A)$. Examining the probabilities of possible steps from b_1, b_2 and c, one finds:

$$\begin{aligned} \mathsf{E}(x, A) &= 1 + \mathsf{E}(b_1, A) \\ \mathsf{E}(b_2, A) = \mathsf{E}(b_1, A) &= 1 + \frac{1}{4}\mathsf{E}(x, A) + \frac{1}{4}\mathsf{E}(b_2, A) + \frac{1}{4}\mathsf{E}(c, A) + \frac{1}{4} \times 0 \\ \mathsf{E}(c, A) &= 1 + \frac{1}{2} \times 0 + \frac{1}{4}\mathsf{E}(b_1, A) + \frac{1}{4}\mathsf{E}(b_2, A) \end{aligned}$$

Solving these equations for $\mathsf{E}(x, A)$ gives that $\mathsf{E}(x, A) = 5$. Thus if the random walk $X_k(t)$ on E_k is undertaken with time intervals α_k between each step, the induced random walk on E_{k-1} will have a mean time interval of $5\alpha_k$ between each step.

[2]Simply note the sequence of vertices of V_{k-1} visited by $X_k(t)$ ignoring consecutive occurrences of the same vertex and regard these as the sequence of vertices visited by a random walk on E_{k-1}.

Whilst a coefficient of 4 might be expected since each vertex is adjacent to four others, the topology of the Sierpinski gasket makes 5 the right number. To take a non-degenerate limit as $k \to \infty$, the random walk on E_{k-1} induced by $X_k(t)$ must, at large scales, be close to the random walk X_{k-1}. To achieve this we need $\alpha_{k-1} = 5\alpha_k$ for each k.

Hence to ensure scaling compatibility, we set $\alpha_k = 5^{-k}$ for $k = 0, 1, 2, \ldots$. Then it maybe shown, analogously to standard Brownian motion on $\mathbb{R}$ or $\mathbb{R}^n$, that the sequence of random walks $X_k(t)$ converges for $k \to \infty$ to a random process $X(t)$ which we call *Brownian motion on the extended Sierpinski gasket E*.

While the basic properties of Brownian motion on $\mathbb{R}$ are mirrored in Brownian motion on E, the exponents are different. As $X_k(t)$ (and thus in the limit $X(t)$) takes an average time of $5\alpha_k = 5^{-k+1}$ to move between adjacent vertices of V_{k-1} which are distance 2^{-k+1} apart, the Brownian motion $X(t)$ typically moves a distance of $\delta t^{\log 2/\log 5}$ in a time interval δt. This should be compared to $\delta t^{1/2}$ for standard Brownian motion on $\mathbb{R}^2$. This leads to defining the *dimension of the walk* for the extended Sierpinski gasket as $d_w = \log 5/\log 2 \approx 2.322$. It is at least plausible that the mean square of the increments satisfies:

$$\mathsf{E}\left(|X(t+\delta t) - X(t)|^2\right) \asymp \delta t^{2/d_w}$$

for $\delta t > 0$. $\asymp$ indicates that this strictly is only asymptotically true, an inevitable consequence of the fractality of the domain. Again, this should be compare to standard Brownian motion on $\mathbb{R}^2$ with $\mathsf{E}\left(|X(t+\delta t) - X(t)|^2\right) = \delta t$.
Moreover it can be shown (see [Barlow & Perkins (1988)]) that the path of Brownian motion on E is Hölder continuous of order γ, thus:

$$|X(t_1) - X(t_2)| \leqslant c\,|t_1 - t_2|^\gamma$$

for all $\gamma < 1/d_w$ and $0 \leqslant t_1 \leqslant t_2 \leqslant T$ where the constant c depends on γ and T. The Hausdorff dimension of the path of Brownian motion is $\log 3/\log 2$ which is equal to the Hausdorff dimension of E itself, so the path "fills" the set E completely.

6.2.2 Heat kernel and transition density

Having now some knowledge of the basic process of Brownian motion and the definition of the dimension of the walk, the *transition density* $p_t(x, y)$ can be derived. It determines the probability density for position $y \in E$ to be reached after time t by Brownian motion starting at $x \in E$, thus for any measurable set A:

$$P\left(X(t+\delta t) \in A | X(t) = x\right) = \int_A p_{\delta t}(x, y)\, d\mu(y)$$

where μ is the restriction of the D_H-dimensional Hausdorff measure to E (thus μ is the natural locally finite measure on E). A knowledge of $p_t(x, y)$ allows the statistics of the motion to be studied. In particular, the transition density is the stochastic analogue to the heat kernel. Using a procedure similar to the one in [Sbalzarini (2001)], pp. 17-21, [Barlow & Perkins (1988)] derived bounds for the transition density by careful analysis of the underlying random walk:

$$\begin{aligned} c_1 t^{-D_H/d_w} \exp\left\{-c_2\left(|x-y|\, t^{-1/d_w}\right)^{d_w/(d_w-1)}\right\} &\leqslant p_t(x, y) \\ &\leqslant c_3 t^{-D_H/d_w} \exp\left\{-c_4\left(|x-y|\, t^{-1/d_w}\right)^{d_w/(d_w-1)}\right\} \end{aligned} \tag{6.1}$$

for certain constants c_1, c_2, c_3, $c_4 > 0$. For standard Brownian motion on $\mathbb{R}^n$ the dimension of the walk is $d_w = 2$ and $\mu = \mathcal{L}^n$ (the Lebesgue measure, which is equal to the n-dimensional "volume" for any Borel set) and the transition density is given by:

$$p_t(x,y) = (4\pi t)^{-n/2} \exp\left\{-|x-y|^2/(4t)\right\} \tag{6.2}$$

which is the familiar Gaussian kernel of diffusion on an infinite domain. As illustrated in [Sbalzarini (2001)], Brownian motion is intimately connected with solutions of the heat equation. Diffusion of heat on $\mathbb{R}^n$ may be thought of as the aggregate effect of a large number of "heat particles" following independent Brownian paths. Let ν be the heat distribution on $\mathbb{R}^n$ at time $t = 0$. Then, the temperature at point x and time t is given by:

$$u(x,t) = \int_{\mathbb{R}^n} p_t(x,y)\, d\nu(y) \tag{6.3}$$

where $p_t(x,y)$ is the standard transition density or Green's function (i.e. the heat kernel evaluated at time t) given by equation 6.2. It may be checked by differentiation that

$$\frac{\partial p_t}{\partial t} = \nabla_x^2 p_t$$

as required for Green's function. Thus 6.3 satisfies the heat equation

$$\frac{\partial u}{\partial t} = \nabla^2 u$$

on $\mathbb{R}^n$ with $\int_A u(x,t)\,dx \to \nu(A)$ as $t \to 0$. In a similar way, Brownian motion on the extended Sierpinski gasket E may be regarded as modeling diffusion on E. Thus an initial heat distribution ν on E would yield the temperature distribution at time t but with $p_t(x,y)$ now being the transition density given by 6.1.

6.2.3 The Laplacian on the Sierpinski gasket

To obtain a meaningful analogue of the heat equation on E, we must define the Laplacian ∇^2 on E. Again, we start with a discrete approximation on the pre-Sierpinski gaskets E_k, which will in the limit give a continuous operator on E. Recall that the Laplacian on $\mathbb{R}$ is the limit of differences:

$$\begin{aligned}\frac{d^2f}{dx^2} &= \lim_{h\to 0} h^{-2}\left[(f(x+h)-f(x)) + (f(x-h)-f(x))\right] \\ &= \lim_{h\to 0} h^{-2} \sum_{y=x\pm h} (f(y)-f(x))\end{aligned}$$

For a continuous $f : E \mapsto \mathbb{R}$ we use discrete approximations via the geometric graphs E_k (see figures 6.4 to 6.6). Writing $C(E)$ for the set of all continuous functions on E, [Falconer (1997)] defines $\nabla^2 f \in C(E)$ by the requirement that, for every bounded set A:

$$\lim_{k\to\infty} \sup_{x\in A\cap V_k} \left| 5^k \cdot \sum_{w\in V_k(x)} (f(w)-f(x)) - \nabla^2 f(x) \right| = 0 \tag{6.4}$$

where $V_k(x)$ comprises the four vertices of V_k adjacent to x and the number "5" is exactly what is needed for this definition to be meaningful. This is a consequence

of the dimension of the walk of E being $\log 5/\log 2$. We are now able to show that the transition density function on E indeed satisfies the heat equation. For given k, let $x, y \in V_k$. On the assumption that Brownian motion on E starting at x at time 0 reaches one of the four adjacent vertices of V_k at time $\delta t = 5^{-k}$ (we now know that this is true *on average*), consideration of transition densities to y gives:

$$p_{t+\delta t}(x,y) \simeq \sum_{w \in V_k(x)} \frac{1}{4} \cdot p_t(w,y)$$

where $x_1, \ldots, x_4 \in V_k(x)$ are adjacent to x. This is true on average, due to the symmetry of E. Thus by simple arithmetic:

$$\left(p_{t+\delta t}(x,y) - p_t(x,y)\right)/\delta t \simeq \frac{1}{4} \cdot 5^k \cdot \sum_{w \in V_k(x)} \left(p_t(w,y) - p_t(x,y)\right)$$

Letting $\delta t \to 0$ and inserting this into equation 6.4 gives:

$$\frac{\partial p_t}{\partial t}(x,y) = \frac{1}{4}\nabla^2 p_t(x,y)$$

Moreover, $p_t(x,\cdot)$ is concentrated around x when t is small since $X(t)$ will not have moved far. Therefore:

$$\int_A u(x,t)\, d\mu(x) \to \nu(A) \qquad \text{as} \quad t \to 0$$

The last two equations mean that the transition density p_t meets all the requirements for a Green's function (see e.g. [Sbalzarini (2001)]) and therefore u, as given by equation 6.3 with p_t now being the transition density for Brownian motion on E, satisfies the diffusion equation on the Sierpinski gasket E:

$$\frac{\partial u}{\partial t} = \frac{1}{4}\nabla^2 u$$

Also notice the changed apparent diffusivity as compared to the Euclidean case. With considerable effort, these arguments can be made more rigorous (see e.g. [Barlow & Perkins (1988)]).

6.2.4 Eigenvalues of the Laplacian

We move on to the related problem of finding the eigenvalues of the Laplacian on a fractal domain. Here we need the domain to be bounded, so from now on we take E to be the non-extended usual Sierpinski gasket and adapt the previous notation in an obvious way to the bounded setting. Thus E_k is the graph with a finite vertex set V_k and edges of length 2^{-k} that approximates E. The definition of the Laplacian given by equation 6.4 is slightly modified to require $\nabla^2 f \in C(E)$ to satisfy:

$$\lim_{k\to\infty} \sup_{x \in V_k \setminus V_0} \left| 5^k \cdot \sum_{w \in V_k(x)} \left(f(w) - f(x)\right) - \nabla^2 f(x) \right| = 0$$

where $V_k(x)$ is the set of vertices in V_k adjacent to x, other than the vertices of V_0. We are interested in eigenfunctions of the Neumann problem in this context, that is, for functions with vanishing normal derivative on V_0, the three corners of E. The eigenvalues of Helmholtz's equation

$$\nabla^2 u + \lambda u = 0 \quad \text{with} \quad u \in C(E) \quad \text{and} \quad \nabla u = 0 \quad \text{for} \quad x \in V_0 \tag{6.5}$$

may be shown to be real and non-negative. We now seek estimates for the *eigenvalue distribution function*:

$$N(\lambda) := \#\{k : \lambda_k \leqslant \lambda\} \tag{6.6}$$

where λ_k are the eigenvalues of problem 6.5. Define the *spectral dimension* as:

$$d_s = \frac{2D_H}{d_w}$$

where D_H is Hausdorff's dimension and d_w is the dimension of the walk as defined in subsection 6.2.1. Whereas d_w indicates the scaling behavior of the rate of diffusion through E, d_s is defined in terms of the "density of states". d_s therefore can be interpreted as the asymptotic frequency of the large eigenvalues of the Laplacian on a bounded region E, which is related to is harmonic composition (hence the name *spectral* dimension). [Barlow & Perkins (1988)] have shown that the eigenvalue distribution function satisfies:

$$N(\lambda) \asymp \lambda^{d_s/2}$$

This should be compared to Weyl's theorem[3] for bounded open domains in $\mathbb{R}^n$ where d_s is replaced by n. Again, one notices the fact that the basic laws remain unchanged when going from Euclidean to fractal sets but the exponents change.

Much more precise information may be obtained on the asymptotics of $N(\lambda)$ by applying the renewal theorem (see e.g. [Falconer (1997)], section 7.2). This theorem gives a periodic positive function $p(\cdot)$ with period $\log 5$ such that:

$$N(\lambda) \sim p(\log \lambda)\lambda^{d_s/2}$$

6.2.5 Extension to infinitely ramified fractals

Up to this point, we have shown that Hausdorff's dimension is not a sufficient parameter to describe the time constants of diffusion on finitely ramified fractals. The Sierpinski gasket has been used as a counter example to disprove the proposition made in [Sbalzarini (2001)]. The ER however belongs to the class of infinitely ramified fractals.

The Sierpinski carpet as depicted in figure 6.8 is a well suited example of this class and will be taken as a model here. Its Hausdorff dimension is $D_H = \log 8/\log 3 \approx 1.893$. As shown in [Barlow & Bass (1992)], Brownian motion on the Sierpinski carpet is a strong Markov process[4] with continuous paths and state space. The infinitesimal generator of the limiting process is again called a "Laplacian". Using similar techniques as in the previous subsections, the fundamental solution of the diffusion equation on the Sierpinski carpet can be derived as the transition density of the underlying Brownian motion. [Barlow & Bass (1992)] found:

$$c_1 t^{-D_H/d_w} \exp\left\{-c_2\left(|x-y|\, t^{-1/d_w}\right)^{d_w/(d_w-1)}\right\} \leqslant p_t(x,y)$$
$$\leqslant c_3 t^{-D_H/d_w} \exp\left\{-c_4\left(|x-y|\, t^{-1/d_w}\right)^{d_w/(d_w-1)}\right\}$$

for certain constants $c_1, \ldots, c_4$. This is exactly the same as equation 6.1 for the Sierpinski gasket. The only difference is that in the present case, the value of d_w

[3] Weyl's theorem states that if $\partial\Omega$ is sufficiently smooth, the eigenvalue distribution function on Ω is $N(\lambda) \asymp c_n \mathcal{L}^n(\Omega)\lambda^{n/2}$ where $\mathcal{L}^n$ means the n-dimensional "volume" and $c_n = (2\pi)^{-n}\mathcal{L}^n(B)$ with B being the unit ball in $\mathbb{R}^n$.

[4] meaning that all steps are independent and there is no historical memory on the paths

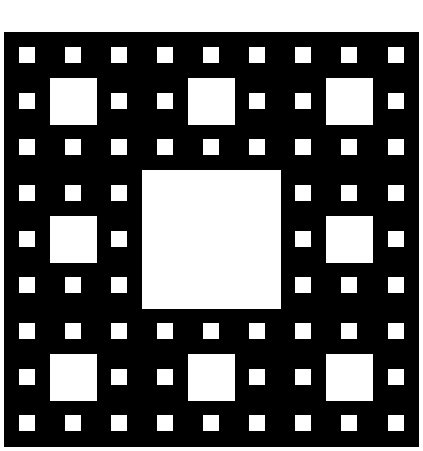


Figure 6.8: Pre-Sierpinski carpet of order 3

is unknown. For infinitely ramified fractals, it is not possible to get numerical values for the dimension of the walk since Brownian motion is now a continuous process. All that can be done is to derive limiting resistances for diffusion (see e.g. [Barlow & Bass (1992)]).

Nevertheless, the identity of the two transition densities shows that the solution of the diffusion equation is topologically equivalent to the one on finitely ramified fractals but with possibly different numerical values for some constants. The basic finding for finitely ramified fractals that Hausdorff's dimension is generally not sufficient to capture the time scales of the solution is therefore true for infinitely ramified fractals as well. This can be intuitively seen if one imagines an infinitely ramified fractal as the limiting case of a partially space-filling curve. Since the curve itself is finitely ramified, it is natural that its infinitely ramified limiting case will inherit the topological properties of the diffusion solution.

6.2.6 Extension to anisotropic diffusion

As real diffusion in the ER might be anisotropic, this case deserves some attention as well. As shown by [Barlow, Hattori, Hattori & Watanabe (1997)], the most interesting aspects of the behavior of diffusion (e.g. the spectral dimensions) are embodied in the asymptotic behaviors of *effective resistances*. These resistances are defined as the H_1 norm of a potential which in turn is the solution to a Laplace equation with appropriate Neumann or Dirichlet boundary conditions. It is thus at least plausible that resistance and diffusion are closely related.

Let $R_n^x(r)$ and $R_n^y(r)$ be the effective resistances of the pre-Sierpinski carpet at the n^{th} stage of its construction in x and y direction, respectively. The anisotropy of the "material" it is made of is parametrized by the ratio of resistances for a unit square: $r = R_0^y/R_0^x$. [Barlow, Hattori, Hattori & Watanabe (1997)] have proven that for sufficiently large n, the ratio $R_n^y(r)/R_n^x(r)$ is bounded by a positive constant independent of r. Furthermore, the ratio decays exponentially fast if $r \gg 1$. This means that for the true Sierpinski carpet with $n \to \infty$ and large ratios of anisotropy, isotropic conditions are weakly restored. Furthermore it is proven that the effective resistances asymptotically grow exponentially with an exponent equal to that found for the isotropic case. Therefore, the topological behavior of the diffusion solution is the same and all deviations are bounded by a positive constant. The role of the spectral dimension is obvious as it is embodied in the effective resistances. What has

been said in the previous sections remains therefore valid (at least asymptotically for large r and n) for the anisotropic case as well.

6.2.7 Diffusion on domains with fractal boundary

The last missing piece in this survey of diffusion on fractal sets is the connection to Euclidean domains. Recall that the ER is not a true fractal in reality but there is a lower bound to the geometrical scales, given by the molecular structure of the ER membrane. At larger scales though, the ER can be *modeled* using *concepts* from fractal theory. The limiting case between the two scales corresponds to a Euclidean domain with fractal boundary.

Then the question is how and whether diffusion *inside* the domain is influenced by the fractality of its boundary. Again we will consider the eigenvalues of the Laplacian as characteristic for the time scales of the solution. Let $D \subset \mathbb{R}^n$ $(n \geqslant 1)$ be a bounded open region with boundary ∂D. Here we do not need D to be connected (although the ER is). The eigenvalue problem is again given by Helmholtz's equation:

$$\nabla^2 u + \lambda u = 0 \qquad \text{in} \quad D$$

with Neumann boundary condition $\nabla u(x) \cdot n = 0$ for $x \in \partial D$. The eigenvalues are those λ_k for which there is a non-trivial solution. Again we are interested in the eigenvalue distribution function as defined by equation 6.6. In particular how its behavior reflects the nature of the domain boundary ∂D. A classical result of Weyl states that is ∂D is sufficiently smooth then:

$$N(\lambda) \sim c_n \mathcal{L}^n(D) \lambda^{n/2}$$

as $n \to \infty$, where $c_n = (2\pi)^{-n} \mathcal{L}^n(B)$, B is the unit ball in $\mathbb{R}^n$ and $\mathcal{L}^n$ is the n-dimensional Lebesgue measure (n-dimensional "volume"). This asymptotic expansion can be continued as:

$$N(\lambda) = c_n \mathcal{L}^n(D) \lambda^{n/2} + b_n \mathcal{L}^{n-1}(\partial D) \lambda^{(n-1)/2} + o\left(\lambda^{(n-1)/2}\right)$$

for a constant b_n depending only on n. Thus the "surface area" of the boundary ∂D determines the second term in the expansion of $N(\lambda)$. Notice that the exponent $(n-1)/2$ is half the dimension of the boundary. For fractal boundaries, it has been shown by [Falconer (1997)], that there is a connection between the fractal dimension of the boundary and the second term in the expansion of $N(\lambda)$. One can thus "hear" the dimension of an oscillating fractal membrane. It can be shown that $\lambda^{s/2}$ is an upper bound for the second term where s is the *interior Minkowski dimension*, a one-sided variant of the box counting dimension as defined in section 4.3.2.

6.3 Conclusions

Summarizing the previous sections it can be said that a model geometry for the ER that has the same box counting or Hausdorff dimension will not necessarily exhibit the same FRAP behavior. Equality in Hausdorff's dimension is **necessary but not sufficient** for the model to be valid. A necessary and sufficient set of equalities would comprise Hausdorff's dimension and the *spectral dimension* (see section 6.2.4), since only then, the two objects will have the same dimension of the walk as defined in section 6.2.1. It is this *dimension of the walk* that captures the properties of Brownian motion and thus diffusion on a given geometry. For Euclidean objects it is always equal to 2, for fractals it will be different. The Sierpinski gasket for example has been shown to have $d_w = 2.322\ldots$. The greater value for the fractal indicates that diffusion on it is slower. The expected mean square increment is $\delta t^{2/d_w}$. This makes sense since the topological structure of the fractal set constrains the Brownian motion. On the other hand diffusion *through a fractal boundary* is faster than through a Euclidean one since its effective surface is larger (e.g. porous media). Moreover, the exponents in most physical laws, the coefficients in the definition of the Laplacian and its eigenvalue distribution function as well as the apparent diffusivity will be different for fractal and Euclidean domains. Only if all of this is taken into account, models that capture the basic integral time behavior can be derived. Unfortunately, the spectral dimension of any given object cannot be easily measured. Indeed it is still unknown how to determine it (or the dimension of the walk) for infinitely ramified fractals. This leads to the conclusion that the idea of a fractal *model geometry* for the ER as proposed in [Sbalzarini (2001)] is impracticable. Instead, the fact that the dimension of the walk completely captures the influence of geometry (or the restriction of Brownian motion to it) on the time behavior of diffusion will be used to directly infer FRAP data models that appropriately account for the complexity of the ER geometry. The results of this chapter show that this is possible using d_w as a single numerical parameter.

Chapter 7

Simulation techniques

7.1 Random walk

For all random walk simulations of diffusion, the 3D variant of the code developed and described in [Sbalzarini (2001)] is used. However, the following improvements have been made in the meantime:

1. The geometry description input files now contain a file header telling the program whether it is an old-style `*.surf` file (cf. [Sbalzarini (2001)]) or an OpenInventor 3D file according to [SGI (1992a)] and [SGI (1992b)]. Therefore the input file read routines have been changed to implement this.

2. The global parameter file `bdiff.dat` has been changed to a more flexible format now allowing comments and directives of the form `KEYWORD = value` in arbitrary order (cf. appendix C.5.2). The new subroutine `ReadParams` (given in appendix C.5.4.5) implements this. For all parameters not being explicitly specified in the input file, default values are now set in `Defaults.f90` (similar to the one in appendix C.5.4.3).

3. The numerical value of π is computed to machine precision as $\pi = 4 \arctan(1)$ rather than being hard-coded to a certain (fixed) precision.

4. Position and size of the bleached box are now read from the parameter file `bdiff.dat` as `bleachbox = <Xmin>, <Ymin>, <Xmax>, <Ymax>` rather than being hard-coded in the program.

5. A global tolerance parameter TOL has been added to make real value comparisons more robust against round-off errors. Two real numbers are now considered equal if their difference is less than TOL.

6. The subroutine `point_in_domain` no longer needs the triangle's normals but explicitly counts the number of triangles intersected along a ray in x-direction starting at the point in question. See algorithm 7.8 for further details on this.

7. Finally, the directions of the random walk steps are no longer chosen on the unit sphere around the current position but on the unit semi-sphere. Consequently, the step length is no longer of fixed sign but is now also allowed to be negative to make the whole space reachable again. Thus, for space particles, the algorithm now proceeds as follows:

Algorithm 7.1 (Space random walk).

Step 1: Choose a random point on the unit semi-sphere centered at the current position $\{(r, \varphi, \vartheta) : r = 1,\ 0 \leqslant \varphi < \pi,\ -\pi/2 \leqslant \vartheta \leqslant \pi/2\}$ according to:

$$\varphi = \pi \cdot \mathcal{U}(0,1) \qquad \vartheta = \arcsin\left(2 \cdot \mathcal{U}(0,1) - 1\right) + \frac{\pi}{2}$$

where $\mathcal{U}(0,1)$ is a uniformly distributed random number between 0 and 1.

Step 2: Choose the step variance according to $\sigma^2 = 2nD\delta t$ where D is the diffusion constant (interpolated as described in [Sbalzarini (2001)]), δt the simulation time step size and n the number of spatial dimensions (thus $n = 3$).

Step 3: Now the random walk step is given by:

$$s = \mathcal{N}(0, \sigma^2) \cdot \begin{bmatrix} \sin\vartheta\cos\varphi \\ \sin\vartheta\sin\varphi \\ \cos\vartheta \end{bmatrix}$$

where $\mathcal{N}(0, \sigma^2)$ means a Gaussian (normal) deviate with zero mean and variance σ^2.

Step 4: Advance to the new position: $x_{n+1} = x_n + s$

For surface particles the corresponding algorithm becomes:

Algorithm 7.2 (Surface random walk).

Step 1: Choose a random point on the unit semi-circle around the current position $\{(r, \vartheta) : r = 1,\ 0 \leqslant \vartheta < \pi\}$ according to: $\vartheta = \pi \cdot \mathcal{U}(0,1)$

Step 2: The step variance is again given by: $\sigma^2 = 2nD\delta t$ with n now being 2 and D the surface diffusion coefficient on the tube in question.

Step 3: The polar angle (φ) and axial coordinate (ζ) of the new position are now given by:

$$\begin{aligned} \varphi_{n+1} &= \varphi_n + \frac{1}{R}\mathcal{N}(0, \sigma^2)\cos\vartheta \qquad \mod 2\pi \\ \zeta_{n+1} &= \zeta_n + \mathcal{N}(0, \sigma^2)\sin\vartheta \end{aligned}$$

where R is the radius of the current tube.

Besides these changes, the program code as well as its input and output facilities are the same as in [Sbalzarini (2001)]. A complete reference manual as well as a printed version of the source code is included in Appendix C of [Sbalzarini (2001)] and will not be reproduced in this report.

7.2 Particle Strength Exchange

In addition to the random walk code, the method of particle strength exchange (PSE) is used. It is a Lagrangian particle method suited to solve diffusion or the diffusive (viscous) part of other equations such as the Navier-Stokes equations in complex geometries. For this work, the PSE method is preferred to other alternatives such as embedded boundary methods ([McCorquodale, Colella & Johansen (2001)])

or the diffusion velocity method ([Beaudoin, Huberson & Rivoalen (2001)]). Advantages over the random walk method are that it converges much faster (as will be shown in section 8.4) and yields smooth solutions. Being a particle method, it is suited for simulations in arbitrarily shaped domains, just as the random walk is. With prospect to future work, the fully anisotropic implementation of [Zimmermann, Koumoutsakos & Kinzelbach (2001)] is used in this project.

7.2.1 The principles of particle methods

The method of particle strength exchange (PSE) is part of a larger family of particle methods widely used in various areas of computational physics such as fluid mechanics, electrostatics, plasma physics, etc. A complete introduction and survey of particle methods can for example be found in [Hockney & Eastwood (1988)], for applications in fluid mechanics [Cottet & Koumoutsakos (2000)] is recommended.

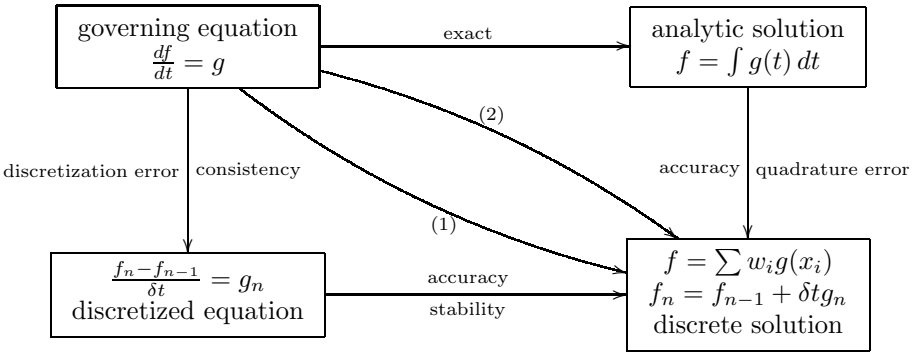


Figure 7.1: Ways to solve a differential equation numerically.

The basic idea of all of these methods consists of finding an integral approximation of the differential operator under consideration, which is then solved by numerical quadrature using the irregularly distributed particle locations as quadrature points. Figure 7.1 depicts the general philosophy. Starting from the governing differential equation (ODE or PDE) of the problem, two ways of solution exist. One is to discretize the equation using a grid-based method such as finite differences, finite elements or finite volumes. This discrete equation can then be solved numerically. The issues to be concerned about include consistency and accuracy for the discretization step as well as stability and accuracy for the solution step. If either consistency or stability criteria are violated, the discretized equation will not represent the original physical problem any more. This is possible since it is – in principle – a completely new, artificial formula for which we are not a priori guaranteed that is has anything to do with the governing equation we started from. The second way consists of writing the analytical solution of the equation (in integral form using Green's function or some kernel) even though we may not be able to explicitly solve it. This step is exact and no errors and constraints need to be worried about. One then proceeds by numerical integration (sometimes called quadrature) of this integral solution. The only issue of this step is accuracy. In fact, this way of solving the equation is *always stable*, provided that the actual solution of the governing equation is stable itself, and consistency is guaranteed by construction of the method. The drawbacks however are that (*i*) quadrature converges slower than numerical solutions of the discretized equation and (*ii*) the resulting system of equations is an N-body problem making it potentially scaling of $\mathcal{O}(N^2)$. It was

this high computational cost that long prevented the use of particle methods in computational science. Fortunately, fast N-body solvers such as cell-list algorithms or multipole expansions which make the methods $\mathcal{O}(N \log N)$ or even $\mathcal{O}(N)$ are available nowadays. See for example [Hockney & Eastwood (1988)] and appendix B of [Cottet & Koumoutsakos (2000)] for detailed descriptions of those fast solvers.

Generally speaking, particle methods are a way of translating the governing PDE of a physical problem to a set of N ODEs where N is the total number of particles involved. These ODEs are called *equations of motion* since they describe the motion of the particles as:

$$\frac{dx_k}{dt} = F(x, t, \omega) \qquad k = 1, \ldots, N$$

As we are interested in representing some smooth function (e.g. charge, concentration, probability, etc) by a particle distribution, the particles also carry a physical quantity generally referred to as their *strength* ω. The strength is also allowed to change in time as characterized by a second set of ODEs:

$$\frac{d\omega_k}{dt} = G(x, t, \omega) \qquad k = 1, \ldots, N$$

Thus, we define a particle as follows:

Definition 7.1 (Particle). A *particle* P is a mathematical object described by its location x_p and the strength ω_p it is carrying: $P = (x_p, \omega_p)$.

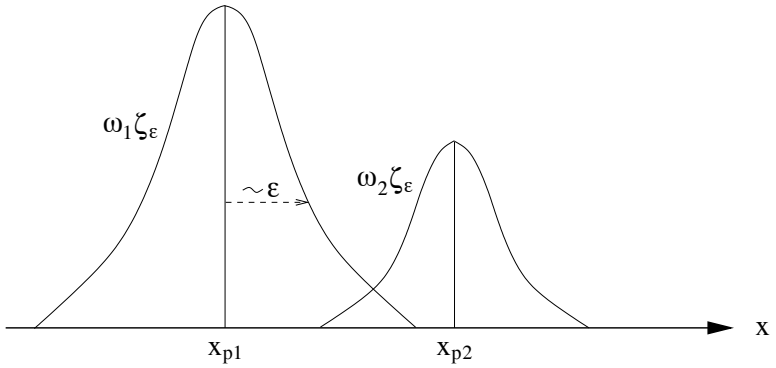


Figure 7.2: Two particles of strength ω_1 and ω_2 carrying mollifier functions ζ_ϵ

In order to be able to reconstruct a smooth function from the particles, they are assigned *mollifier functions* ζ_ϵ. A mollifier is a local (but not necessarily compact) function centered at the particle's position (see figure 7.2). It can be thought of as a cloud of charge, mass, etc. that is carried around by the particle and has to meet certain requirements (see [Cottet & Koumoutsakos (2000)] for details). The *core size* ϵ of this mollifier determines the size of the particles. Introducing such finitely sized particles is equivalent to fixing the particle interaction potential such that is goes to zero if the distance between two particles vanishes[1]. The smaller the particles are, the more accurate the representation of the smooth function gets.

[1]This unphysical fix is needed to prevent the particle interactions from becoming of infinite magnitude. It will however not introduce a large error as the particles will not come that close in real simulations anyway.

For the approximation to converge, it is however required that the distance h between any two particles is always less than their size[2], thus:

$$\frac{h}{\epsilon} < 1$$

This means that more accurate function approximations can only be obtained by increasing the number of particles when decreasing their size.

7.2.2 The isotropic PSE method

As for diffusion, the governing equation is given by

$$\frac{\partial c}{\partial t} = D\nabla^2 c(x,t) \tag{7.1}$$

the PSE method as introduced by [Degond & Mas-Gallic (1989a)] starts by finding an integral formulation of the Laplacian that allows consistent evaluation on the particle locations. To start with, consider the solution at a location y and expand it into a Taylor series around x:

$$c(y) = c(x) + \sum_{i=1}^{r+1} \frac{1}{i!} (y-x)^i \nabla^i c(x) + O\left(|y-x|^{r+2}\|c\|_\infty\right)$$

Subtracting $c(x)$ on both sides, multiplying the whole equation by a *regularized kernel function* η_ϵ and integrating over y yields:

$$\begin{aligned}\int (c(y)-c(x))\,\eta_\epsilon(y-x)\,dy = \sum_{i=1}^{r+1} \frac{1}{i!} \int (y-x)^i \nabla^i c(x) \eta_\epsilon(y-x)\,dy \\ + \|c\|_\infty O\left(\int |y-x|^{r+2}\eta_\epsilon(y-x)\,dy\right)\end{aligned}$$

To get an approximation of the Laplacian, we have to ask the following general requirement for the kernel function η (according to [Degond & Mas-Gallic (1989a)]):

$$\int_{\mathbb{R}^n} \prod_{i=1}^{n} x_i^{\alpha_i} \eta(x)\,dx = \begin{cases} 0, & \forall\, \alpha \in \mathbb{N}^n,\ \alpha \neq 2e_i,\ 1 \leqslant \sum_{i=1}^{n} \alpha_i \leqslant r+1 \\ 2, & \text{if } \alpha = 2e_i,\ i \in [1,\ldots,n] \end{cases}$$

where n is the dimension of the space, r is the *order* of the regularized kernel function and $x = (x_1,\ldots,x_n)$ is a position in $\mathbb{R}^n$. $\alpha = (\alpha_1,\ldots,\alpha_n)$ is an n-dimensional index and $(e_1,\ldots,e_n)$ is the canonical base of $\mathbb{R}^n$. In the 3D case, this requirement may be expressed as:

$$\int x_i x_j \eta(x)\,dx = 2\delta_{ij} \qquad \text{for } i,j = 1,2,3 \tag{7.2}$$

$$\int x_1^{i_1} x_2^{i_2} x_3^{i_3} \eta(x)\,dx = 0 \quad \text{if } i_1+i_2+i_3 = 1 \text{ or } 3 \leqslant i_1+i_2+i_3 \leqslant r+1 \tag{7.3}$$

$$\int |x|^{r+2} |\eta(x)|\,dx < \infty \tag{7.4}$$

for $i_1, i_2, i_3 \in \mathbb{N}_0^+$. The first condition is to normalize the kernel function. The second one requires all moments up to order r+1 to vanish and the third one is

[2]This is known as "particles must overlap".

required to have the truncation error bounded. The regularized kernel is derived from this kernel function as follows:

$$\eta_\epsilon(x) = \epsilon^{-n}\eta\left(\frac{x}{\epsilon}\right) \qquad , \epsilon > 0$$

where n is the topological dimension of the space we wish to solve the diffusion equation in. Using requirements 7.2 and 7.3, one notices that all terms of the form $\int (y-x)^i \eta_\epsilon(y-x)\, dy$ in above series expansion vanish. The only term remaining is:

$$\int (c(y) - c(x))\, \eta_\epsilon(y-x)\, dy = \nabla^2 c(x) \int (y-x)^2 \eta_\epsilon(y-x)\, dy + \|c\|_\infty O\left(\int |y-x|^{r+2} \eta_\epsilon(y-x)\, dy\right)$$

The term $\nabla^2 c(x)$ can be taken out of the integral as it does not depend on y. After change of variables $z = (y-x)/\epsilon$ in the integrals, one obtains:

$$\epsilon^{-n} \int (c(y) - c(x))\, \eta_\epsilon(y-x)\, dy = \nabla^2 c(x) + O\left(\epsilon^r\right)$$

due to the requirement 7.2 for the kernel function η. Thus the integral operator that approximates the Laplacian is found to be:

$$\nabla^2_\epsilon c(x) = \epsilon^{-n} \int (c(y) - c(x))\, \eta_\epsilon(y-x)\, dy \tag{7.5}$$

and the approximation error is $O(\epsilon^r)$ with r being the largest integer for which conditions 7.3 and 7.4 are fulfilled (see [Cottet & Koumoutsakos (2000)] for a rigorous error treatment). Using the particle locations as quadrature points leads to the discrete version of the operator:

$$\nabla^2_{\epsilon,h} c^h(x_p^h) = \epsilon^{-n} \sum_{q \neq p} (v_q c_q^h - v_p c_p^h) \eta_\epsilon(x_q^h - x_p^h) \tag{7.6}$$

where v_q and v_p are the particle's volumes such that $v_q c_q^h = c(x_q^h)\, dy$ is the strength (mass is this context). It is noteworthy that this operator is not the only possibility of discretizing the Laplacian onto particles. Compared to another method (called Fishelov's scheme) it has however the big advantage of being conservative. As above operator implicitly incorporates the conservation of mass, it is considered the better choice for solving the diffusion equation.

The approximation c^h to the continuous concentration c at any location and time can be reconstructed from the strengths $v_p c_p^h$ of the particles using:

$$c^h(x,t) = \sum_p v_p c_p^h(t) \zeta_\epsilon(x - x_p^h)$$

where $\zeta_\epsilon(x) = \epsilon^{-n}\zeta(x/\epsilon)$ is the mollifier function that the particles "carry around" (not to be confused with the PSE kernel η_ϵ). In the simplest case of point particles it is identical to the Dirac delta distribution: $\zeta(x) = \delta(x)$. The final PSE scheme is now easily obtained by inserting equation 7.6 into equation 7.1:

$$\frac{\partial c_p^h}{\partial t} = D\epsilon^{-n} \sum_{q \neq p} (v_q c_q^h - v_p c_p^h) \eta_\epsilon(x_q^h - x_p^h) \qquad \forall\, p \in [1, \ldots, N]$$

This is an N-body problem as for each particle it involves a sum over all other particles. However, since the kernel η_ϵ is chosen to be local, only the nearest neighbors

of each particle significantly contribute to its sum. The simulation code therefore implements a cell-list algorithm for nearest neighbor search and interactions are only calculated between particles that are closer than a cut-off of 10ϵ. It can also be seen from this equation, that in order to simulate diffusion, the strengths of all the particles change (i.e. they exchange mass) while their locations remain the same, i.e. they do not move. This means that all the geometry and boundary condition handling only needs to be done once when initializing the particles (cf. section 7.2.5). Moreover, it allows regular spacing of the particles inside the computational domain in which case the particle volumes simply become: $v_p = h_1 h_2 h_3$ where $h_{1,2,3}$ are the (constant) inter-particle spacings in all three spatial directions.

The anisotropic advection-dispersion equation

It is however possible to also take convection into account. In this case the particles move as governed by the convection and they exchange strength according to diffusion. It is this appealing physical problem separation character that makes part of the beauty of particle methods. The simulation code used in this work directly implements a solver for the advection-dispersion equation:

$$\frac{\partial c(x,t)}{\partial t} + \nabla(u(x,t) \cdot c(x,t)) - \sigma(x,t) = \nabla(D(x,t) \cdot \nabla c(x,t)) \tag{7.7}$$

where u is a given mean velocity field, c stands for the local concentration and σ is a term to account for sources and sinks. Due to the anisotropy, D now is a symmetric tensor of second rank that contains diffusion and dispersion coefficients.

Using the principles of particle methods, this equation is now transformed into the following set of equations:

$$\frac{dx_p}{dt} = u(x_p, t) \tag{7.8}$$

$$\frac{\partial c(x,t)}{\partial t} = \nabla(D(x,t) \cdot \nabla c(x,t)) + \sigma(x,t) \tag{7.9}$$

with $u(x,t)$ being a given external mean velocity field which moves the particles along their characteristics. The first equation describes the movement of the particles due to convection, the second equation describes the changes in their strength due to diffusion and is solved using the anisotropic PSE method as described below.

7.2.3 The anisotropic extension of the PSE method

In order to have a PSE simulation that is functionally equivalent to the random walk code, an extended anisotropic version of it is used. In this case, the diffusion coefficient is no longer a scalar but a symmetric tensor of second rank that is allowed to vary in space (in 3D):

$$D(x) = \begin{bmatrix} D_{xx}(x) & D_{xy}(x) & D_{xz}(x) \\ D_{xy}(x) & D_{yy}(x) & D_{yz}(x) \\ D_{xz}(x) & D_{yz}(x) & D_{zz}(x) \end{bmatrix} \tag{7.10}$$

Similar to the derivations for the isotropic case in section 7.2.2, it has been shown by [Degond & Mas-Gallic (1989b)] by means of Taylor series expansions that the following integral operator Q_ϵ is an approximation of the differential diffusion operator:

$$\nabla(D\ \nabla c(x,t)) \approx Q_\epsilon(t)\ c(x,t) = \int_{\mathbb{R}^n} \sigma_\epsilon(x,y,t)\ [c(y) - c(x)]\, dy \tag{7.11}$$

The regularized kernel $\sigma_\epsilon = \epsilon^{-n}\sigma(x/\epsilon)$ again satisfies certain moment conditions. The discretized particle approximation $\overline{Q}_\epsilon^h$ is obtained by applying a quadrature rule to the integral operator $Q_\epsilon(t)$ using the particles as quadrature points:

$$\overline{Q}_\epsilon^h(t)\; c_k^h(t) = \sum_l \sigma_\epsilon(x_k(t), x_l(t), t)\; [c_l(t) - c_k(t)]\; \cdot h^n \tag{7.12}$$

where h is the inter-particle spacing and n is the space dimension. The regularized kernel is defined as:

$$\sigma_\epsilon(x_k, x_l, t) = \epsilon^{-n} \sum_{i,j=1}^{n} M_{ij}(x_k, x_l, t)\psi_{ij}^\epsilon(x_l - x_k) \tag{7.13}$$

where ϵ is again the core size of the particles and $M_{ij}(x, y, t)$ is a function of the diffusion tensor D_{ij}. With the spherically symmetric matrix cut-off (smoothing) function:

$$\psi_{ij}^\epsilon(x) = \epsilon^{-n}\psi_{ij}\left(\frac{x}{\epsilon}\right)$$

According to [Degond & Mas-Gallic (1989b)], the components of ψ are chosen to be:

$$\psi_{ij} = \epsilon^{-(n+2)}\overline{\Theta}\left(\frac{|x_k - x_l|}{\epsilon}\right) \cdot \sum_{i,j=1}^{n} (x-y)_i (x-y)_j$$

with a scalar cut-off function $\overline{\Theta}(x)$. Substituting everything into equation 7.13 yields the regularized anisotropic PSE kernel function:

$$\sigma_\epsilon(x_k, x_l, t) = \frac{1}{\epsilon^{n+4}}\,\overline{\Theta}\left(\frac{|x_k - x_l|}{\epsilon}\right) \sum_{i,j=1}^{n} M_{ij}(x_k, x_l, t)(x_k - x_l)_i (x_k - x_l)_j$$

Introducing the following constraint according to [Degond & Mas-Gallic (1989b)]:

$$\frac{4\pi}{15}\int_0^\infty r^6\;\overline{\Theta}(r)\,dr \stackrel{!}{=} 1$$

with $r = (x_i - x_j)$, $\overline{\Theta}(r) = a_0 \cdot e^{-\beta r^2}$ and $\beta = 1$, we obtain the following normalized 2^{nd} order cut-off function:

$$\overline{\Theta}\left(\frac{|x_k - x_l|}{\epsilon}\right) = \frac{4}{\pi\sqrt{\pi}}\; e^{-\frac{(x_k - x_l)^2}{\epsilon^2}} \tag{7.14}$$

[Degond & Mas-Gallic (1989b)] suggest $M(x_k, x_l, t)$ to be of the form

$$M(x_k, x_l) = \frac{1}{2}\left(m(x_k) + m(x_l)\right)$$

where

$$m = D \;-\; \frac{1}{n+2}\,Tr(D)\cdot\mathbb{I}$$

with $Tr(\cdot)$ standing for the trace of a matrix, and $\mathbb{I}$ for the identity matrix.
In 3D, the diffusion tensor is given by equation 7.10 and thus we get:

$$m = \begin{bmatrix} \frac{1}{5}(4D_{xx} - D_{yy} - D_{zz}) & D_{xy} & D_{xz} \\ D_{xy} & \frac{1}{5}(4D_{yy} - D_{xx} - D_{zz}) & D_{yz} \\ D_{xz} & D_{yz} & \frac{1}{5}(4D_{zz} - D_{yy} - D_{xx}) \end{bmatrix}$$

where all coefficients can be functions of the location x. Finally, using a simple Euler time discretization with time step size δt, the discretized form of equation 7.9 in 3D may be expressed as:

$$c_k^{n+1} = c_k^n + \frac{\delta t \cdot h_1 h_2 h_3}{\epsilon^7} \sum_{l=1}^{N} (c_l^n - c_k^n) \cdot \frac{4}{\pi\sqrt{\pi}}\, e^{-\frac{(x_k - x_l)^2}{\epsilon^2}} \Big(M_{11}(x_k - x_l)_1^2 + 2 \cdot M_{12}(x_k - x_l)_1 (x_k - x_l)_2 + 2 \cdot M_{13}(x_k - x_l)_1 (x_k - x_l)_3 + M_{22}(x_k - x_l)_2^2 + 2 \cdot M_{23}(x_k - x_l)_2 (x_k - x_l)_3 + M_{33}(x_k - x_l)_3^2 \Big)$$

As in the isotropic case, the sum is only to be taken over the nearest neighbors within a distance of 10ϵ due to the local character of the interaction kernel (negative exponential). This is efficiently done using a cell-list algorithm. The algorithm has been implemented in a parallel simulation code using MPI by Dr. Jens Walther and Stephanie Zimmermann (see [Zimmermann, Koumoutsakos & Kinzelbach (2001)] for a reference application). The changes and additions needed to handle finite domains with arbitrary triangulated boundaries and to make the code restartable are however products of the present project.

7.2.4 Boundary condition handling

Since the PSE algorithm as described above strictly only applies to infinite domains, some kind of boundary condition handling needs to be included. The easiest way to do so consists of placing mirror particles in a r_c-neighborhood outside of the simulation domain Ω. The same technique is also used for analytically solving partial differential equations using Green's function. It is exact for domains whith boundaries that can be described as a set of infinite straight planes (e.g. cubes, boxes, etc.). This clearly is not the case for the ER, giving rise to some boundary errors. Since such errors violate the no flux boundary condition, they will change the total mass inside the domain Ω. Checking the conservation of mass therefore is a way to monitor those errors.

The PSE code is extended to handle three different kinds of particles: real particles, mirror particles for the boundary condition and ghost particles for inter-processor communication, each kind is identified by a particle attribute which is positive for real particles, negative for mirror particles and zero for ghost particles. To enforce the boundary condition, the strength of all mirror particles is set to the one of their corresponding real particle every time step directly after the PSE solver.

7.2.5 Geometry processing and initialization

Since no convection is present (we are simulating pure diffusion), the PSE algorithm does not move the particles but only changes their strengths. This is an important difference to the random walk technique and eliminates the need for geometry processing (checking whether a particle is inside or outside the domain) at every time step. Instead, the geometry handling needs to be done only once when initializing the particles. Therefore, a preprocessor code is written that writes an input file for the PSE solver containing all initial particle positions, strengths and attributes. This preprocessor is the only algorithm that needs to know the triangulated surface description of the domain Ω. Once the particles are initialized, the information about the shape of the domain are contained in their positions since no particles with positive attribute will be placed outside of Ω.

All geometry processing and particle initialization is concentrated in the program `init_part`. The particles are initialized on a regular cartesian lattice placing a

particle at the center of each grid cell[3]. The complete source code is given in appendix C.5.4. The top-level algorithm proceeds as follows, marginal numbers refer to statement labels in the main program (appendix C.5.4.2):

Algorithm 7.3 (Preprocessing).

36 Step 1: Set default values for all variables.

110 Step 2: Read triangulation set and problem parameters.

Step 3: Determine the bounding box of the triangulation $[x_l, x_h] \times [y_l, y_h] \times [z_l, z_h]$ as: $\{x, y, z\}_l = \min_p\{x_p, y_p, z_p\}$, $\{x, y, z\}_h = \max_p\{x_p, y_p, z_p\}$ where p loops over
148-163 all triangle vertices in $\aleph$.

231-240 Step 4: Calculate centroids and normals of all triangles.

241 Step 5: Set up triangle lists and sort triangles.

Step 6: Check topological and syntactical validity of the triangulation according to
256-264 the algorithms presented in chapter 3.

Step 7: Initialize particles on regular lattice by placing a particle at the center of each
294-316 grid cell that is inside the domain Ω.

Step 8: Initialize strengths of all particles by setting the ones inside the bleached box to zero, all others to c_0. Initialize all attributes to any value > 0 (typically a
402-414 running index).

390-394 Step 9: Add mirror particles for boundary condition handling.

Step 10: Calculate and output some diagnostics.

417-424 Step 11: Write particle file to be read in by the PSE solver.

448-459 Step 12: Deallocate memory and terminate.

Some explanations to selected steps are in order. Step 5 is the core part of two cell-list algorithms included for performance reasons. Without any sorting, step 7 would have to check every triangle for each grid cell to determine whether the cell's center is inside the domain or not, making it $\mathcal{O}(NM)$. This is computationally not feasible for any surfaces containing more than just a few dozens of triangles. Therefore, step 5 sets up two different lists of triangles: the first list sorts the triangles into (y, z)-bins B of size $[x_l, x_h] \times b_y \times b_z$:

$$B_{i,j} = \{(x, y, z) : x_l \leqslant x \leqslant x_h,\ (i-1)b_y \leqslant y - y_l < ib_y,\ (j-1)b_z \leqslant z - z_l < jb_z\}$$

It proceeds as follows (see appendix C.5.4.9 for the source code, marginal numbers refer to the corresponding statement labels):

Algorithm 7.4 (Make bin lists).

Step 1: Subdivide the bounding box into $N_y \times N_z$ cylindrical bins of size $b_x = x_h - x_l$,
21-27 $b_y = (y_h - y_l)/N_y$, $b_z = (z_h - z_l)/N_z$.

Step 2: Associate a triangle list $L_{i,j}(k)$ with each bin (i, j)

Step 3: Loop over all bins and assign all triangles that intersect the parallelepiped $\{(x, y, z) : x_l \leqslant x \leqslant x_h,\ y_l + (i-1)b_y \leqslant y < y_l + ib_y,\ z_l + (j-1)b_z \leqslant z <$
28-143 $z_l + jb_z\}$ to the triangle list $L_{i,j}$ of bin (i, j).

[3]This staggered arrangement is needed to avoid particle loss due to round-off errors if particles sit exactly on the domain boundaries

Since a triangle typically belongs to several bins, regular lists have to be used instead of linked lists[4] (see section 7.2.6 for details on the list structure). The code efficiently determines all bins that are intersected by a certain triangle Δ_k using the following algorithm:

Algorithm 7.5 (Sort triangles to bin lists).

Step 1: For each vertex $P_{1,2,3}$ of Δ_k, determine the index of the bin it is in: $(i,j) = \text{ceiling}\left((P_y - y_l)/b_y,\ (P_z - z_l)/b_z\right)$, where ceiling$(f)$ is the closest integer larger or equal to f. If Δ_k is not yet contained in the list $L_{i,j}$, add it: $L_{i,j}(\text{length}(L_{i,j})+1) = k$. `41-63`

Step 2: Determine the minimum and maximum bin indices for Δ_k: $\{i,j\}_l = \min_{i,j}\{(i,j) : \Delta_k \in L_{i,j}\}$, $\{i,j\}_h = \max_{i,j}\{(i,j) : \Delta_k \in L_{i,j}\}$ `64-67`

Step 3: Loop over all bins $\{(i,j) : i \in [i_l, i_h],\ j \in [j_l, j_h]\}$

3.1: If any corner point of the bin's projection onto the (y,z)-plane is inside the triangle's projection, add Δ_k to $L_{i,j}$ if it is not already in this bin's triangle list. Go to step 3. `90-106`

3.2: If any edge of the triangle's projection onto the (y,z)-plane intersects any edge of the bin's projection, add Δ_k to $L_{i,j}$ if it is not already in this bin's triangle list. `107-138`

Steps 3.1 and 3.2 use 2×2 linear systems of equations to determine the intersection points, analogous to the ones in chapter 3. Figure 7.3 depicts the geometrical situation, taken from a real run with the erp572 ER sample. The point in question is $P = (401.2188\ldots,\ 387.6003\ldots,\ 5.2666\ldots)$. It is marked with a large light-blue asterisk. The corresponding bin this point belongs to has indices $(70, 38)$ and is shown as a red box. All 49 triangles that intersect this bin (thus are elements of its triangle list) are drawn in green. The 4 triangles that actually are intersected by the ray starting at P and proceeding parallel to the x-axis are shown in pink, the ray itself and all ray-triangle intersection points are colored blue. Due to the fact that there are 4 intersected triangles, the point P is outside of the domain Ω.

The second list sorts the triangles into (x,y,z)-cells C of size $b_x \times b_y \times b_z$:

$$C_{i,j,k} = \{(x,y,z) : (i-1)b_x \leqslant x - x_l < ib_x,\ (j-1)b_y \leqslant y - y_l < jb_y,\ (k-1)b_z \leqslant z - z_l < kb_z\}$$

It is only used for the creation of the boundary condition mirror particles as it simplifies the search for points closer than r_c to the surface and – for such points – the search for the triangle closest to it (to mirror at). The creation of the lists is done in `AllocateLL` (see appendix C.5.4.10) as:

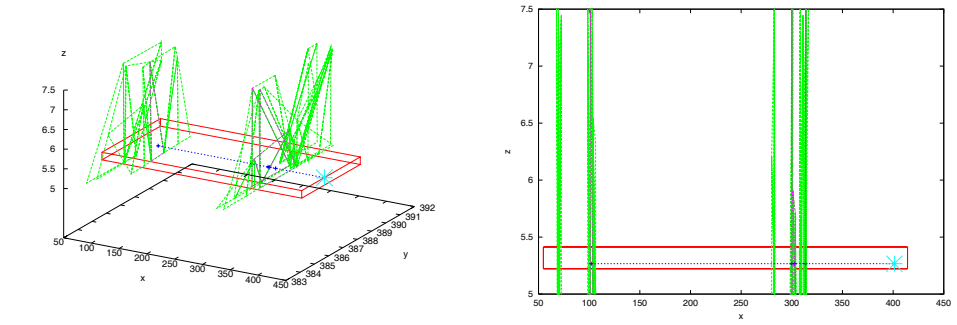
Algorithm 7.6 (Make cell lists).

Step 1: Determine the number of cells needed: $N_x = \text{int}((x_h - x_l)/r_c)$, $N_y = \text{int}((y_h - y_l)/r_c)$, $N_z = \text{int}((z_h - z_l)/r_c)$ where int(f) means the closest integer $\leqslant f$. `35-37`

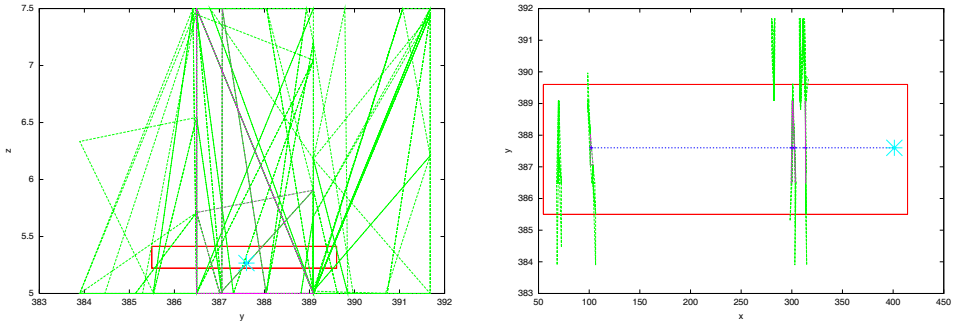
Step 1: Subdivide the bounding box into cubic cells of edge length $b_x = (x_h - x_l)/N_x$, $b_y = (y_h - y_l)/N_y$, $b_z = (z_h - z_l)/N_z$. `41-43`

Step 2: Associate a triangle list $L_{i,j,k}(p)$ with each cell (i,j,k) `45-66`

Step 3: Loop over all cells and assign triangles to cell lists according to algorithm 7.7



Perspective view Side view



Front view Top view

Figure 7.3: Geometric situation for the bin list algorithm. See text for explanations

It is necessary to set up completely separate lists for bins and cells since neither the size of the cells nor the criterion when a triangle is element of a certain cell's list are identical. Since the bins discussed above serve to determine whether a point is inside or outside of the domain, all triangles that intersect the bin must be in its lists. The cells discussed now however serve to determine whether a point is closer than r_c to a triangle. Therefore their size is not arbitrary but must be set to r_c. Furthermore, not all triangles that intersect a certain cell must be in its triangle list but only those being closer than r_c to the cell's centroid (thus in the in-sphere of the cell). To keep the lists as short as possible, the following algorithm is implemented in `SortT` (appendix C.5.4.9) to sort the triangles Δ_n to the cell lists:

Algorithm 7.7 (Sort triangles to cell lists).

Step 1: For each vertex $P_{1,2,3}$ of Δ_n, determine the index of the cell it is in: $(i,j,k) = \text{ceiling}\left((P_x - x_l)/b_x,\ (P_y - y_l)/b_y,\ (P_z - z_l)/b_z\right)$, where ceiling$(f)$ is the closest integer larger or equal to f. If Δ_n is not yet contained in the list $L_{i,j,k}$,
172-197 add it: $L_{i,j,k}(\text{length}(L_{i,j,k}) + 1) = n$.

Step 2: Determine the minimum and maximum cell indices for triangle Δ_n: $\{i,j,k\}_l = \min_{i,j,k}\{(i,j,k) : \Delta_n \in L_{i,j,k}\}$, $\{i,j,k\}_h = \max_{i,j,k}\{(i,j,k) : \Delta_n \in L_{i,j,k}\}$
198-203

Step 3: Loop over all cells $\{(i,j,k) : i \in [i_l, i_h],\ j \in [j_l, j_h],\ k \in [k_l, k_h]\}$

3.1: If triangle Δ_n is not already in $L_{i,j,k}$, calculate the intersection point of the triangle's normal through the cell's centroid with the triangle (see

[4] In a linked list of objects, each object can only occur once

figure 5.1 for the geometrical situation). If this intersection point is inside the triangle and inside the cell, add Δ_n to $L_{i,j,k}$. 222-241

Notice that the second algorithm (the cell lists) is only needed for boundary mirror particles. It is therefore not executed when boundary condition handling is switched off.

Using the described bin and cell list algorithms has the effect of reducing the execution time from several dozens of days to about one minute for a typical ER sample consisting of about $8 \cdot 10^4$ triangles and $2 \cdot 10^6$ grid points.

Now having completed step 5 of the top-level algorithm 7.3, some comments on steps 7 and 9 follow. These steps are concerned with the actual initialization of the particles in the regular lattice cells. The core algorithm needed is the one to determine whether a given point is inside or outside of the domain Ω. According to the solution of the topological container problem given in [Sbalzarini (2001)], this is done by counting the number of intersections of an arbitrary ray with the domain boundary $\partial\Omega$ starting at the point in question. If this count is odd, the point is inside Ω, else outside. Using the bin lists described above, this is implemented for any given point Q in $\mathcal{O}(N \log M)$. The code is given in appendix C.5.4.11 to which the marginal statement labels refer.

Algorithm 7.8 (Point in domain).

Step 1: Determine indices of bin Q is in: $(i, j) = \text{ceiling}\left((Q_y - y_l)/b_y,\ (Q_z - z_l)/b_z\right)$ and set the intersection counter to zero: $N_i = 0$. 18-26

Step 2: Loop over all triangles in this bin's triangle list, thus $\{\Delta_k : \Delta_k \in L_{i,j}\}$ 61

2.1: Intersect the ray $Q + \lambda[1, 0, 0]$ with triangle Δ_k by solving the following linear system of equations:

$$\begin{bmatrix} | & | & 1 \\ a & b & 0 \\ | & | & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ -\lambda \end{bmatrix} = \begin{bmatrix} | \\ Q - P_1 \\ | \end{bmatrix}$$

73-86

where $a = P_2 - P_1$ and $b = P_3 - P_1$ are edge vectors of Δ_k and $P_{1,2,3}$ are its vertices. Since the direction can be chosen arbitrarily, it is chosen to yield a simple system of equations that can be solved at minimum cost. For this reason, the general triangle-ray intersection routine described in section 3.2 is not used but above system's solution is hard-coded.

2.2: If $\alpha \geqslant 0$ and $\beta > 0$ and $\alpha + \beta \leqslant 1$ and $\lambda > 0$ (up to a tolerance TOL), increment the intersection counter by 1: $N_i = N_i + 1$. 87,105

Step 3: If N_i is odd, the point Q is inside the domain Ω, else outside. 109-113

Finally, step 9 of algorithm 7.3 adds the mirror particles for the boundary condition handling according to section 7.2.4. It is only invoked if needed and proceeds for any given point $Q \in \Omega$ as follows:

Algorithm 7.9 (Add mirror particles).

Step 1: Determine the indices of the cell Q belongs to: $(i, j, k) = \text{ceiling}\left((Q_x - x_l)/b_x,\ (Q_y - y_l)/b_y,\ (Q_z - z_l)/b_z\right)$. 119-127

Step 2: Loop over all triangles Δ_n in this cell and its 26 nearest neighbors $(i \pm 1, j \pm 1, k \pm 1)$ (if they exist). 128-131

2.1: Intersect the normal n_n of Δ_n through Q with Δ_n using the general intersection routine described in section 3.2: $Q + \lambda_n n_n = P_1 + \alpha a_n + \beta b_n \Rightarrow \lambda_n$. 134

Step 3: Find the closest triangle by selecting the one with the minimum λ: $\lambda_{min} = \min_n \lambda_n$ and $n_{min} = \text{argmin}_n \lambda_n$. 135-140

Step 4: If $\lambda_{min} \leqslant r_c$, mirror point Q according to: $Q_{mirror} = Q + 2\lambda_{min} n_{n_{min}}$ where $n_{n_{min}}$ means the normal of the the closest triangle. Set the strength (mass) of Q_{mirror} to strength of Q and the attribute of Q_{mirror} to the negative attribute of Q. 142 main 391

This algorithm makes use of the cell list as described in algorithms 7.6 and 7.7 to reduce the computational cost from $\mathcal{O}(MN)$ to $\mathcal{O}(N \log M)$.

Now the preprocessing is completed and the particle locations, strengths and attributes are written to a binary file which can be read by the PSE solver as an initial condition.

7.2.6 Implementation notes

Some details on the technical implementation of the algorithms described in the preceding section seem noteworthy. They mainly concern the data structures involved and the methods of parallelization using the MPI message passing interface. Readers not interested in such details can easily skip this subsection without loss of essential information. Nevertheless, the following is included in this report for the sake of completeness since the ideas presented hereafter are neither obvious nor straightforward.

Dynamic list data structures

The algorithms presented in section 7.2.5 often involve lists of triangles associated to cells or bins. The general structure is that the bounding box of the computational domain is subdivided into a number of disjoint box-shaped sub-spaces. These subspaces are called *cells* if the subdivision takes place in all three spatial dimensions and *bins* if the space is only subdivided along two dimensions. Obviously, bins are just cells with one index removed. Therefore they can be treated using the same data structures. It was however necessary to distinguish between them in the previous section since the criteria for a triangle to be a member of a bin or cell are different. The following will only deal with cells, the application to bins (2-index cells) is straightforward.

Assume the bounding box of the domain is subdivided into $N_x \times N_y \times N_z$ cells that are addressed using integer indices $\{(i, j, k) : 1 \leqslant i \leqslant N_x,\ 1 \leqslant j \leqslant N_y,\ 1 \leqslant k \leqslant N_z\}$. To each cell, we associate a list $L_{i,j,k}$ that contains the indices p of all the triangles that "belong" to the cell (i.e. meet all the criteria of the cell): $L_{i,j,k}(p)$. The obvious way to implement such an object in Fortran 90 would be an array of rank 4. Its size in the fourth dimension would have to be equal to the number of triangles in the longest list of all cells. This is however not favorable for the following two reasons:

1. One triangle will typically belong to several lists (i.e. it intersects several cells), making the length of the longest list large.

2. The triangles are very inhomogeneously distributed in the bounding box (as they are concentrated on the surface of the domain).

The latter point means that in fact most cells will not contain any triangles. Just the ones intersecting the domain boundary will have lists of non-zero length. It would thus be a huge waste of memory to allocate a cubic four dimensional array.

Instead, a new data structure is defined which can be seen as an array of pointers to lists. A list is implemented as a 1D array of variable length. This allows the lengths of all the lists to be individually set and it brought down the main memory usage of the algorithm from about 500 MB to 1 MB. The following Fortran statements define the new data type as implemented in `globals.f90` (see appendix C.5.4.1, statements 5 to 7 and 27, 29):

```
TYPE ptr_to_list
   REAL(MK), DIMENSION(:), POINTER               :: list
END TYPE
TYPE(ptr_to_list), DIMENSION(:,:,:), ALLOCATABLE :: cell
```

The following statements allocate an array of $N_x \times N_y \times N_z$ empty triangle lists:

```
ALLOCATE(cell(Nx,Ny,Nz))
nullify(cell)
```

The current lengths of all the lists are stored in an additional array of integers: `ncell(Nx,Ny,Nz)`. Every time we wish to add an element to a list, we first check whether the list is long enough to hold the new element. If this is not the case, it is dynamically enlarged. The process of adding an element m to the end of a list therefore is described by:

```
p = ncell(i,j,k)+1
if(ASSOCIATED(cell(i,j,k)%list)) then
   if(p > size(cell(i,j,k)%list)) then
      cell(i,j,k)%list=>reallocate(cell(i,j,k)%list,p)
   end if
else
   ALLOCATE(cell(i,j,k)%list(p))
end if
cell(i,j,k)%list(p) = m
```

The function `reallocate` encapsulates the work of dynamically resizing an object while preserving its contents. It takes a pointer to an array and returns a pointer to the new, resized array. According to [Numerical Recipes in Fortran 90 (1996)] it is multiply overloaded for different array dimensions and data types and implemented as a Fortran 90 module (see appendix C.5.4.15):

```
INTERFACE reallocate
   MODULE PROCEDURE reallocate_rv,reallocate_rm,reallocate_iv,  &
        reallocate_im,reallocate_hv
END INTERFACE
```

Since all triangle lists are 1D arrays of integers, only the version for integer vectors is used. It reads as follows (statements 22 to 35):

```
function reallocate_iv(p,n)
  INTEGER(I4B), DIMENSION(:), POINTER         :: p, reallocate_iv
  INTEGER(I4B), INTENT(IN)                    :: n
  INTEGER(I4B)                                :: nold, ierr
```

```
  ALLOCATE(reallocate_iv(n), STAT=ierr)
  if(ierr .NE. 0) then
     WRITE(*,'(A)') 'Error allocating memory in reallocate_iv'
     return
  end if
  if(.NOT. ASSOCIATED(p)) return
  nold = size(p)
  reallocate_iv(1:min(nold,n))=p(1:min(nold,n))
  DEALLOCATE(p)
END function reallocate_iv
```

To simplify the usage of these list types, the subroutine `AllocateLL` has been implemented (see appendix C.5.4.10). It takes 5 integer arguments: a flag, the three indices of the cell the operation should affect and an error status. Table 7.1 summarizes possible ways of using it. After the subroutine returns, the value of the fifth variable is zero if no error occurred, else it is non-zero.

Call	Purpose
`AllocateLL(0,0,0,0,istat)`	Initialize all lists and allocate memory
`AllocateLL(1,0,i,j,istat)`	Enlarge bin list (i, j)
`AllocateLL(1,i,j,k,istat)`	Enlarge cell list (i, j, k)
`AllocateLL(2,0,0,0,istat)`	Clear all lists and free all memory

Table 7.1: Calling modi of `AllocateLL`

Paradigms of parallelism

For reasons of speed and to be prepared for larger problems, the particle preprocessing algorithm 7.3 is parallelized using MPI. Examination of algorithm 7.3 reveals that the most time consuming part is step 7 as it consists of invoking algorithm 7.8 for every grid point. Two different ways of parallelizing this step can be imagined:

1. Distribute grid points: all processors know all triangles but only check a fraction of the grid points.
2. Distribute triangles: one processor loops over all the grid points but the triangles to be intersected with are distributed among the other processors.

For the following reasons, the first possibility is better than the second:

- The number of points is typically more than an order of magnitude larger than the number of triangles.
- Communication only happens once at the very beginning of step 7 and once at its end (for possibility 2 one would have to collect the numbers of intersection after every point).

The amount of data that has to be communicated is $9M + G$ for the first case and $G(\log M + n)$ for the second with G being the number of grid points, M the number of triangles and n the number of processors. Even though the amount of data that has to be transmitted is larger in the first case, only two communication channels have to be opened as opposed to $G \approx 10^6$ is the second case. Since for MPI, the latency of opening a channel is crucial rather than the bulk amount of data transmitted once the channel is open, the first alternative will be faster.

The parallel algorithm therefore proceeds as follows (statement numbers refer to the listing in appendix C.5.4.2):

Algorithm 7.10 (MPI parallelization).

Step 1: Start MPI and probe all channels and processors. 37–105

Step 2: Master processor reads all triangles from file. 110

Step 3: Master broadcasts all triangles to slaves. 127–147

Step 4: Master does cartesian domain decomposition and assigns a set of grid points to check to each processor. 176–191

Step 5: Each processor loops over all grid points it got assigned and checks whether they are inside or outside the domain (i.e. invokes algorithm 7.8). The results are stored in a local `flag` array. 294–316

Step 6: Slaves communicate their results (`flag`) to master. 319–330

Step 7: Master concatenates flag arrays. 327

The array `flag` temporarily holds the local result of algorithm 7.8 for all grid points on a processor. This is recommended to avoid the need of communicating them back to the master after each grid point.

7.3 Post-processing and visualization

The PSE simulation code following the algorithms described in sections 7.2.2 and following can write output files at certain time steps containing the positions and strengths of all the particles. The files are in ASCII format and each line contains the information of one particle as:

```
(4E16.8) xp(1,i), xp(2,i), xp(3,i), c(i)
```

where `xp(1..3,i)` are the x, y and z coordinates of the position of particle i and `c(i)` is its strength (i.e. the local concentration).

In order to be able to use these output files to visualize concentration fields in OpenDX[5], they have to be post-processed. Just as other visualization tools, OpenDX can only handle data when its *connection structure* is known. This means that the data points either have to be on a grid or a triangulation of the scattered data locations has to be performed, which is $\mathcal{O}(N^4)$ however. Since the preprocessor described in section 7.2.5 initializes the particles on a regular lattice and they do not move during the PSE simulation (there is no convection), it is best to convert the output files to a grid representation again. The grid will extend throughout the bounding box of the particles. However, not all grid points will be occupied by a particle since they have only been placed inside the ER geometry. The conversion therefore consists of determining which grid points are associated to which particle and which are empty.

The program `res2dx` as given in appendix C.3.2.1 performs this task for all the input files passed to it as command line arguments. Before being able to translate the PSE result files to OpenDX grid data, it has to recover the grid geometry from the available data. Since the grid is known to be cartesian and regular, this only includes the numbers of grid points N_x, N_y, N_z in all three spatial directions, the bounding box of the data and the corresponding grid spacings δx, δy, δz. Using this information, the OpenDX header file (called a *general* file) can then be written. This file contains all the information about the data's structure and organization.

[5]OpenDX is a freely available data visualization tool formerly known as IBM Visualization Data Explorer. See http://www.opendx.org for further information.

For regular grid data, its syntax is:

```
file = <name of grid data file>
grid =  <Nx> x <Ny> x <Nz>
format = ascii
interleaving = record
majority = row
field = c
structure = scalar
type = float
dependency = positions
positions =  regular, regular, regular, <xl>, <dx>, <yl>, <dy>, <zl>, <dz>
end
```

where `xl,yl,zl` are the lower boundaries for x, y and z (i.e. the coordinates of the lower left corner of the bounding box). Then, the strengths of all the particles are written to a separate grid data file that simply contains one floating point number per line:

```
c(i)
```

indicating the strength of the particle at grid point i. Hereby, the grid points are listed in row-majority order, as declared in the header file. This means that for each grid point, the conversion program has to find the particle that belongs to it in order to write its strength to the output file. Since particles are only placed where there is ER geometry, not every grid point will have a particle, so this check is actually needed. Without further tricks, this would be $\mathcal{O}(NN_xN_yN_z)$ and thus unfeasible. A chaining mesh of cells is therefore established in the bounding box of the data which is thus subdivided into $Nc_x \times Nc_y \times Nc_z$ mesh cubes C of size $b_x \times b_y \times b_z$:

$$C_{i,j,k} = \{(x,y,z) : (i-1)b_x \leqslant x - x_l < ib_x,\ (j-1)b_y \leqslant y - y_l < jb_y,\\ (k-1)b_z \leqslant z - z_l < kb_z\}$$

Unlike for all the triangle lists discussed so far, a particle belongs to exactly one such cell $C_{i,j,k}$. It is therefore possible to use a linked list of particles instead of the dynamic list data structures presented in section 7.2.6. To do so, a *head of chain* particle $H_{i,j,k}$ is assigned to each cell $C_{i,j,k}$ and a linked list of particles $\ell(p)$ is set up for $p = 1, \ldots, N$ such that each entry points to the next particle in the same cell. The end of the list is marked by a zero entry. This cell-list algorithm brings the computational cost for the conversion down from $\mathcal{O}(NN_xN_yN_z)$ to $\mathcal{O}(N_xN_yN_z)$. Figure 7.5 illustrates this structure.

The following algorithm describes the conversion process in detail. The marginal numbers refer to the statement labels in the source code as given in appendix C.3.2.1.

Algorithm 7.11 (Conversion of particles to grid data).

53 Step 1: Determine number of particles in input file: N

Step 2: Determine bounding box of data $[x_l, x_h] \times [y_l, y_h] \times [z_l, z_h]$ as: $\{x,y,z\}_l = \min_p\{x_p, y_p, z_p\}$, $\{x,y,z\}_l = \min_p\{x_p, y_p, z_p\}$ where p loops over all input
55-60 particles: $p = 1, \ldots, N$

Step 3: Determine grid spacings as the minimum distances of all possible pairwise distinct particles in all three spatial directions: $\delta x = \min_{(p,q),p\neq q} |x_p - x_q|$,
61-63 $\delta y = \min_{(p,q),p\neq q} |y_p - y_q|$, $\delta z = \min_{(p,q),p\neq q} |z_p - z_q|$

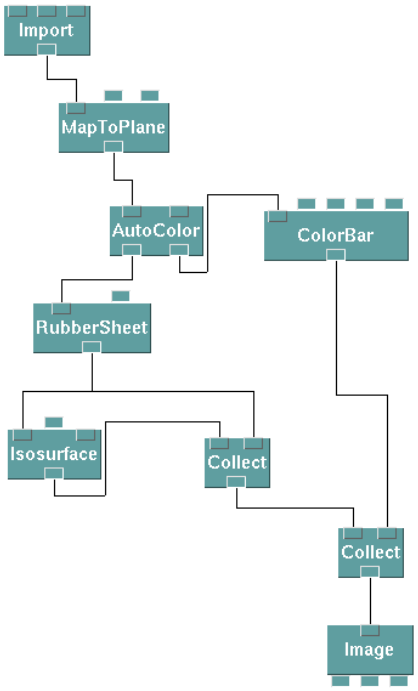


Figure 7.4: OpenDX visual program

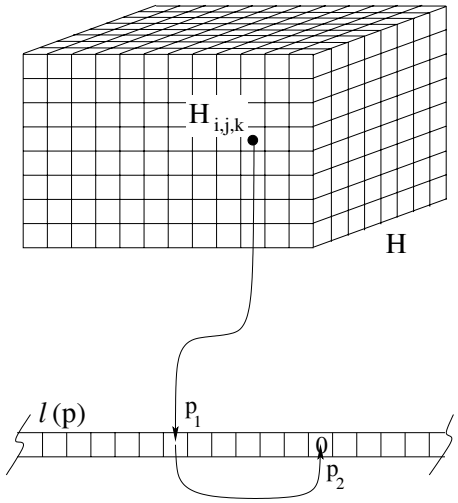


Figure 7.5: A linked list data structure

Step 4: Calculate the number of grid points as: $(N_x, N_y, N_z) = \text{floor}((x_h - x_l)/\delta x,\ (y_h - y_l)/\delta y,\ (z_h - z_l)/\delta z)$ 67–69

Step 5: Write data header file according to above syntax 74–87

Step 6: Nullify the linked list: $H_{i,j,k} = 0\ \forall\ (1 \leqslant i \leqslant Nc_x,\ 1 \leqslant j \leqslant Nc_y,\ 1 \leqslant k \leqslant Nc_z)$, $\ell(p) = 0\ \forall\ p \in [1, N]$ 88–99

Step 7: Determine the size of the cells: $b_x = (x_h - x_l)/Nc_x,\ b_y = (y_h - y_l)/Nc_y,\ b_z = (z_h - z_l)/Nc_z$ 107–109

Step 8: For each particle x_p, $p = 1, \ldots, N$:

- 8.1: Determine the indices of the cell the particle is in: $(i, j, k) = \text{ceiling}((x_p - x_l)/b_x,\ (y_p - y_l)/b_y,\ (z_p - z_l)/b_z)$ 111–119
- 8.2: Add particle p to the linked list of cell $C_{i,j,k}$: $\ell(p) = H_{i,j,k},\ H_{i,j,k} = p$ 120–121

Step 9: For each grid point (x, y, z):

- 9.1: Determine the indices of the cell it is in: $(i, j, k) = \text{ceiling}((x - x_l)/b_x,\ (y - y_l)/b_y,\ (z - z_l)/b_z)$ 133–141
- 9.2: Loop over all particles in the same cell by following the linked list. If a particle is found that is closer to the grid point (x, y, z) than $r = \frac{1}{2}\min\{\delta x, \delta y, \delta z\}$, write its strength to the output grid data file. If no particle is found at all, output 0. 142–152

Step 9 loops over all grid points in *row majority*. This means that the innermost loop is over z and the outermost is over x. Traversing the linked list in step 9.2 for a cell $C_{i,j,k}$ is done as follows:

```
f = 0
p = H_{i,j,k}
do while p ≠ 0 and f = 0
   if ((x_p, y_p, z_p) − (x, y, z))^2 < r^2 then
      write c(p)
      f = 1
   end if
   p = ℓ(p)
end do
if f = 0 then
   write 0
end if
```

Using the header and grid files produced by this algorithm, the visual program shown in figure 7.4 can be used in OpenDX to produce 3D visualizations of the concentration field. Figure 7.6 shows an example taken from a simulation run of section 9.1.

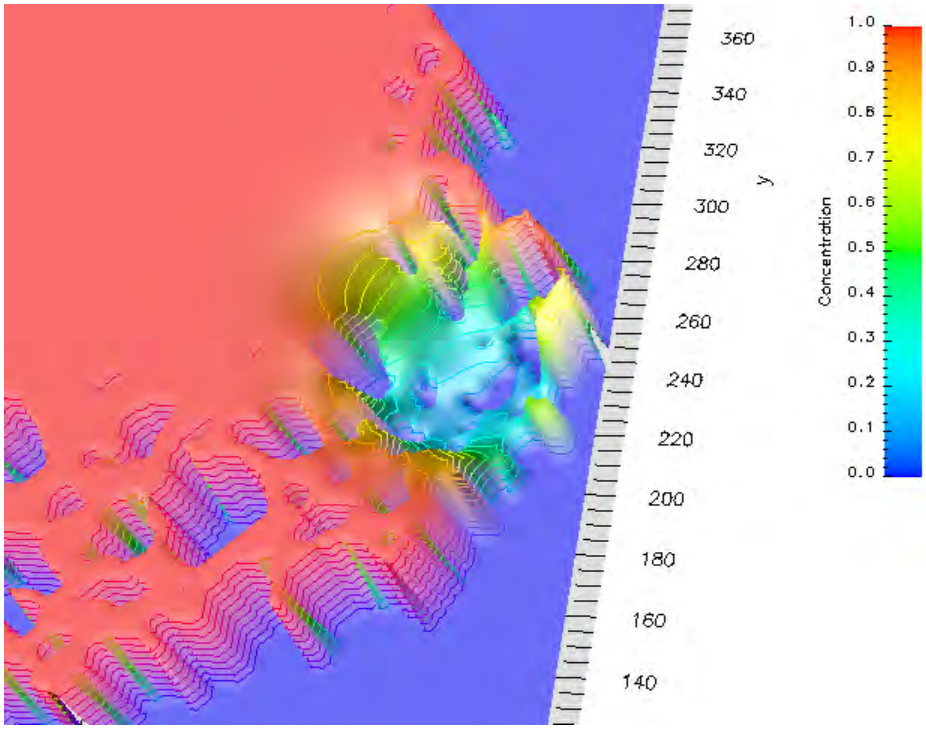


Figure 7.6: Example visualization of a simulated concentration field

Chapter 8

Tests and validation

8.1 Test case description

In order to be able to validate the random walk and PSE simulation codes, a simple test case for which an analytic solution exists is sought. To maintain the link to FRAP analysis, the cubic box used throughout [Sbalzarini (2001)] is chosen, thus the problem domain consists of the cube:

$$\Omega = [0, L] \times [0, L] \times [0, L]$$

with boundary $\partial\Omega$ and interior $int(\Omega) = \{\Omega \setminus \partial\Omega\}$. Since ER lumen proteins do not cross the ER membrane, the biologically correct boundary condition is zero flux at all walls, thus:

$$\frac{\partial(\cdot)}{\partial n} = \nabla(\cdot) \cdot n = 0 \qquad \text{on } \partial\Omega$$

where n is the outer normal on $\partial\Omega$.

This cubic domain is intersected by the square cylinder:

$$B = [a, b] \times [c, d] \times [0, L]$$

which can be thought of as the area where photobleaching takes place at time $t = 0^-$ (i.e. just before time zero). Therefore, the cylinder B will be called "bleached box" hereafter. See figure 8.1 for an overview of this geometrical setting. The initial condition that corresponds to a FRAP setting is:

$$u_0(x) = \begin{cases} 0 & \text{if } x \in B \\ c_0 & \text{if } x \in int(\Omega) \setminus B \end{cases} \tag{8.1}$$

The governing equation to be solved on this geometry with above initial and boundary conditions is the diffusion equation:

$$\begin{cases} \frac{\partial u}{\partial t} = D\nabla^2 u + q(x,t) & \text{for } x \in int(\Omega),\ t > 0 \\ \alpha\frac{\partial u}{\partial n} + \beta u = r(x,t) & \text{for } x \text{ on } \partial\Omega \\ u(x, t=0) = u_0(x) & \text{for } x \in int(\Omega) \end{cases} \tag{8.2}$$

where ∇^2 is the Laplacian operator and $u(x,t)$ is the unknown function to be solved for. Since we have zero flux boundary conditions and there are no sources to be considered, the simplifications $q \equiv 0$, $r \equiv 0$, $\alpha = 1$ and $\beta = 0$ are readily made.

8.2 The analytic solution

Eigenfunctions and Eigenvalues

The analytic solution of the test case as defined in the previous section can be derived using the general approach of eigenfunction series expansions. The basic domain Ω is the cartesian product of three identical intervals. Therefore, the normalized eigenfunctions $\varphi(x)$ and eigenvalues λ of Helmholtz's equation on the 1D domain $[0, L]$ are needed. The eigenproblem to be solved is:

$$\begin{cases} \nabla^2\varphi + \lambda\varphi = 0 & \text{if } x \in (0, L) \\ \frac{d\varphi}{dx} = 0 & \text{for } x = 0 \text{ and } x = L \end{cases} \tag{8.3}$$

Writing φ' for $\frac{d\varphi}{dx}$, this becomes:

$$\varphi''(x) + \lambda\varphi(x) = 0$$

which has the general solution

$$\varphi(x) = A\cos\left(\sqrt{\lambda}x\right) + B\sin\left(\sqrt{\lambda}x\right)$$

using the boundary condition at $x = 0$ yields:

$$\varphi'(0) = B\sqrt{\lambda} \stackrel{!}{=} 0 \qquad \Rightarrow \quad B = 0$$

since λ must not vanish in all cases. The second boundary condition at $x = L$ leads to:

$$\begin{aligned} \varphi'(L) = -A\sqrt{\lambda}\sin\left(\sqrt{\lambda}L\right) \stackrel{!}{=} 0 \qquad & \Rightarrow \quad \sin\left(\sqrt{\lambda}L\right) = 0 \\ & \Rightarrow \quad \sqrt{\lambda}L = k\pi \qquad \forall\, k \in \mathbb{N}_0^+ \end{aligned}$$

since $\lambda \equiv 0$ is, again, not a valid solution. These eigenfunctions are now normalized to unity by requiring:

$$\int_0^L \varphi^2(x)\, dx \stackrel{!}{=} 1$$

which is equal to setting their scalar products to one[1]. For $k > 0$, this corresponds to:

$$\begin{aligned} & A^2 \int_0^L \cos^2\left(\sqrt{\lambda}x\right)\, dx \stackrel{!}{=} 1 \\ \Rightarrow & A^2 \left[\frac{1}{2\sqrt{\lambda}}\left(x + \sin\left(\sqrt{\lambda}x\right)\cos\left(\sqrt{\lambda}x\right)\right)\right]_0^L = \frac{1}{2}A^2L \stackrel{!}{=} 1 \\ \Rightarrow & A = \sqrt{\frac{2}{L}} \end{aligned}$$

For $k = 0$, λ is zero as well and above condition becomes:

$$A^2 \int_0^L dx = A^2L \stackrel{!}{=} 1 \qquad \Rightarrow \quad A = \sqrt{\frac{1}{L}}$$

[1] We require the eigenfunctions to be an orthonormal basis of the solution space of problem 8.2

Therefore, the required normalized eigenfunctions and eigenvalues for this case are given by:

$$2\varphi_k(x) = \begin{cases} \sqrt{\frac{2}{L}} \cos\left(\frac{k\pi}{L}x\right) & \text{if } k > 0 \\ \sqrt{\frac{1}{L}} & \text{if } k = 0 \end{cases} \tag{8.4}$$

$$\lambda_k = \left(\frac{k\pi}{L}\right)^2 \qquad k \in \mathbb{N}_0^+ \tag{8.5}$$

It is easy to verify that the $\varphi(x)$ fulfill

$$\int_0^L \varphi_i(x)\varphi_j(x)\,dx = \delta_{ij}$$

which is necessary and sufficient for an orthonormal function base.

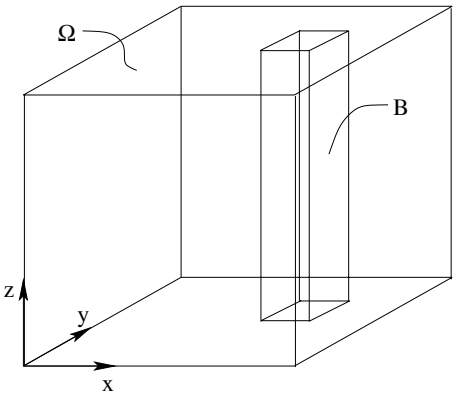


Figure 8.1: Geometry of test case

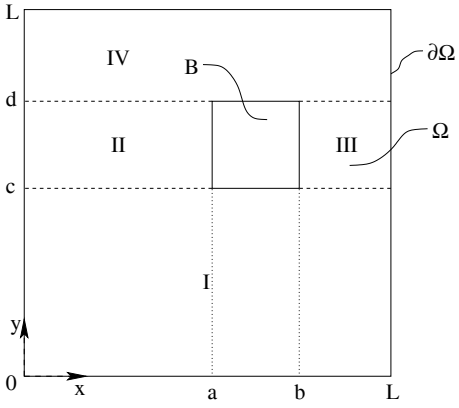


Figure 8.2: Slice through domain with integral areas

Solution by Eigenfunction series expansion

The general solution of problem 8.2 for $\alpha = 1$ and $\beta = 0$ is given by:

$$u(x,t) = \int_\Omega K(x,\xi,t)u_0(\xi)\,d\xi + \int_0^t \int_\Omega K(x,\xi,t-\tau)q(\xi,\tau)\,d\xi\,d\tau$$

$$+D\int_0^t \int_{\partial\Omega} K(x,\xi,t-\tau)r(\xi,\tau)\,ds_\xi\,d\tau$$

where ds_ξ means an infinitesimal segment of $\partial\Omega$ and $K(x,\xi,t)$ is the heat kernel (see also [Sbalzarini (2001)] pp. 20-21). According to the product space theorem, the heat kernel of a product space is equal to the product of the heat kernels of its cartesian factors. Since Ω is the cartesian product of three intervals in x, y and z, one writes:

$$K(x,y,z,\xi,\eta,\zeta,t) = K_x(x,\xi,t)\cdot K_y(y,\eta,t)\cdot K_z(z,\zeta,t) \tag{8.6}$$

where the individual factor kernels $K_{x,y,z}$ are given by:

$$K(x,\xi,t) = \sum_{k=0}^{\infty} \varphi_k(x)\varphi_k(\xi)e^{-D\lambda_k t} \tag{8.7}$$

$$K(y,\eta,t) = \sum_{k=0}^{\infty} \varphi_k(y)\varphi_k(\eta)e^{-D\lambda_k t} \tag{8.8}$$

$$K(z,\zeta,t) = \sum_{k=0}^{\infty} \varphi_k(z)\varphi_k(\zeta)e^{-D\lambda_k t} \tag{8.9}$$

with φ_k and λ_k being above eigenfunctions and eigenvalues of Helmholtz's equation 8.3. Using $q \equiv 0$ and $r \equiv 0$, the solution can formally be written as:

$$u(x,y,z,t) = \int_\Omega K(x,y,z,\xi,\eta,\zeta,t)u_0(\xi,\eta,\zeta)\,d\xi\,d\eta\,d\zeta$$

Taking into account the special topology of the initial condition, this is equal to:

$$u(x,y,z,t) = c_0 \int_{\Omega\setminus B} K(x,y,z,\xi,\eta,\zeta,t)\,d\xi\,d\eta\,d\zeta$$

and basically consists of integrating the heat kernel over $\Omega \setminus B$. This is done by splitting the integral into four parts according to figure 8.2:

$$\begin{aligned} u(x,y,z,t) =& c_0 \underbrace{\int_0^L\int_0^c\int_0^L K\,d\xi\,d\eta\,d\zeta}_{I} + c_0 \underbrace{\int_0^a\int_c^d\int_0^L K\,d\xi\,d\eta\,d\zeta}_{II} \\ &+ c_0 \underbrace{\int_b^L\int_c^d\int_0^L K\,d\xi\,d\eta\,d\zeta}_{III} + c_0 \underbrace{\int_0^L\int_d^L\int_0^L K\,d\xi\,d\eta\,d\zeta}_{IV} \end{aligned} \tag{8.10}$$

The only differences between these integrals are their limits. It is therefore sufficient to know the general definite integral of the k-th Eigenfunction:

$$\int_a^b \varphi_k(x)\,dx = \begin{cases} \sqrt{\frac{2}{L}}\frac{L}{k\pi}\left[\sin\left(\frac{k\pi}{L}b\right) - \sin\left(\frac{k\pi}{L}a\right)\right] & \text{if } k > 0 \\ \sqrt{\frac{1}{L}}\,(b-a) & \text{if } k = 0 \end{cases} \tag{8.11}$$

The four integral terms are now treated in turn.

Integral I

Substituting the heat kernel from equations 8.6 to 8.9 into integral I gives:

$$\int_0^L\int_0^c\int_0^L \sum_{k=0}^{\infty}\sum_{l=0}^{\infty}\sum_{m=0}^{\infty} \varphi_k(x)\varphi_l(y)\varphi_m(z)\varphi_k(\xi)\varphi_l(\eta)\varphi_m(\zeta)e^{-D(\lambda_k+\lambda_l+\lambda_m)t}\,d\xi\,d\eta\,d\zeta$$

Switching the order of integration and summation and rearranging some terms this becomes:

$$\sum_{k=0}^{\infty}\sum_{l=0}^{\infty}\sum_{m=0}^{\infty} \varphi_k(x)\varphi_l(y)\varphi_m(z)\int_0^L \varphi_k(\xi)\,d\xi\int_0^c \varphi_l(\eta)\,d\eta\int_0^L \varphi_m(\zeta)\,d\zeta\; e^{-D(\lambda_k+\lambda_l+\lambda_m)t}$$

This can now be integrated using 8.11 three times with appropriate substitutions. Then, since $\sin(k\pi) \equiv 0 \ \forall k$, only terms with $k = m = 0$ contribute to the sums. Taking this into account, substituting explicit expressions for the eigenfunctions using equation 8.4 and taking all terms for $l = 0$ out of the sum to get rid of the case distinctions leads to the final result for integral I:

$$I = \frac{c}{L} + \sum_{l=1}^{\infty} \frac{2}{l\pi} \sin\left(\frac{l\pi}{L}c\right) \cos\left(\frac{l\pi}{L}y\right) e^{-D\lambda_l t} \tag{8.12}$$

Integral II

Substituting the heat kernel from equations 8.6 to 8.9 into integral II, switching the order of integration and summation and doing some rearrangements of terms gives:

$$\sum_{k=0}^{\infty}\sum_{l=0}^{\infty}\sum_{m=0}^{\infty} \varphi_k(x)\varphi_l(y)\varphi_m(z) \int_0^a \varphi_k(\xi)\,d\xi \int_c^d \varphi_l(\eta)\,d\eta \int_0^L \varphi_m(\zeta)\,d\zeta\, e^{-D(\lambda_k+\lambda_l+\lambda_m)t}$$

Using 8.11 for each integral in this expression and noticing that only terms with $m = 0$ are non-zero leads to the following final expression for integral II:

$$II = \sum_{k=0}^{\infty}\sum_{l=0}^{\infty} \varphi_k(x)\varphi_l(y) \underbrace{\begin{cases} \frac{\sqrt{2L}}{k\pi}\sin\left(\frac{k\pi}{L}a\right) \\ \sqrt{\frac{1}{L}}a \end{cases}}_{t_1} \underbrace{\begin{cases} \frac{\sqrt{2L}}{l\pi}\sin\left(\frac{l\pi}{L}d\right) \\ \sqrt{\frac{1}{L}}(d-c) \end{cases}}_{t_3} e^{-D(\lambda_k+\lambda_l)t} \tag{8.13}$$

where the stacked notation in curly braces means that the upper expression is to be used if the corresponding index variable (k or l) is > 0, the lower one if it is zero.

Integral III

Substituting the heat kernel from equations 8.6 to 8.9 into integral III, again switching the order of integration and summation and doing some rearrangements, one finds:

$$\sum_{k=0}^{\infty}\sum_{l=0}^{\infty}\sum_{m=0}^{\infty} \varphi_k(x)\varphi_l(y)\varphi_m(z) \int_b^L \varphi_k(\xi)\,d\xi \int_c^d \varphi_l(\eta)\,d\eta \int_0^L \varphi_m(\zeta)\,d\zeta\, e^{-D(\lambda_k+\lambda_l+\lambda_m)t}$$

Using 8.11 for each integral in this expression and noticing that only terms with $m = 0$ contribute to the sum leads to the following final expression for integral III:

$$III = \sum_{k=0}^{\infty}\sum_{l=0}^{\infty} \varphi_k(x)\varphi_l(y) \underbrace{\begin{cases} -\frac{\sqrt{2L}}{k\pi}\sin\left(\frac{k\pi}{L}b\right) \\ \sqrt{\frac{1}{L}}(L-b) \end{cases}}_{t_2} \underbrace{\begin{cases} \frac{\sqrt{2L}}{l\pi}\sin\left(\frac{l\pi}{L}d\right) \\ \sqrt{\frac{1}{L}}(d-c) \end{cases}}_{t_3} e^{-D(\lambda_k+\lambda_l)t} \tag{8.14}$$

Integral IV

Substituting the heat kernel from equations 8.6 to 8.9 into integral IV and doing the usual rearrangements gives:

$$\sum_{k=0}^{\infty}\sum_{l=0}^{\infty}\sum_{m=0}^{\infty}\varphi_k(x)\varphi_l(y)\varphi_m(z)\int_0^L\varphi_k(\xi)\,d\xi\int_d^L\varphi_l(\eta)\,d\eta\int_0^L\varphi_m(\zeta)\,d\zeta\,e^{-D(\lambda_k+\lambda_l+\lambda_m)t}$$

Inserting explicit expressions for the eigenfunctions from 8.4 and again using equation 8.11 in its appropriate form and the fact that only for $k = m = 0$ all of the three factors are non-zero, the final result for integral IV becomes:

$$IV = \frac{L-d}{L} - \sum_{l=1}^{\infty}\frac{2}{l\pi}\sin\left(\frac{l\pi}{L}d\right)\cos\left(\frac{l\pi}{L}y\right)e^{-D\lambda_l t} \tag{8.15}$$

Notice that the sum is to be taken starting from $l = 1$ rather than $l = 0$ since the $l = 0$ terms have been explicitly taken out of the sum to get rid of case distinctions (cf. also integral I).

Final solution and discussion

Now the final solution for $u(x,t)$ is known from equation 8.10 by substituting all integral expressions 8.12, 8.13, 8.14 and 8.15:

$$u(x,y,t) = c_0\left(I(y) + II(x,y) + III(x,y) + IV(y)\right) \tag{8.16}$$

If is noteworthy that the exact solution to the test problem turns out to be independent of z. This is due to the zero flux Neumann boundary condition as it allows free slip in directions tangential to the domain boundary $\partial\Omega$. Dirichlet (no slip) boundary conditions would in fact induce a z-dependence since the solution would have to skew towards the two ends of the bleached box in order to maintain $u = 0$ on $\partial\Omega$. Using Neumann boundary conditions on the other hand, the concentration front diffuses in orthogonal to $\partial\Omega$ and the solution will look the same on all z-planes. This is no contradiction to the findings in [Sbalzarini (2001)] as the apparent D values will be different in 2D and 3D since all moves parallel to the z axis are missed by the observer. Nevertheless, the shape of the solution in 3D looks the same on all z-planes. It will however not look the same as the 2D solution at the same time. At different times though, the 2D and 3D solutions will in fact be equal and it is exactly due to this time shift that the apparent diffusion constant is different[2]

Numerical results and visualizations

The exact solution as given by equation 8.16 consists of an infinite series expansion in a function space whose base functions are the eigenfunctions of Helmholtz's equation on the given domain Ω. In order to get useful results, a numerical evaluation of it for certain locations in space and certain times is made. First implementations in MATLAB turned out to be impracticable as the solution consists of double sums which makes it $\mathcal{O}(M^2)$ to be evaluated using M terms in the expansion. Therefore, a Fortran program has been written to do the evaluation fast. The corresponding source code is given in appendix C.1.2.1. The code is written to minimize its execution time by pre-calculating as many terms as possible and using the fact that the same terms appear in different places of the solution, e.g. t_1, t_2 and t_3 as labeled in equations 8.13 and 8.14. Moreover, it contains the compiler directives needed to parallelize it on a vector processor.

[2] recall that the diffusion constant basically reflects the solution's time scales for fixed lengths.

The solution is approximated on a regular cartesian grid with spacing δ in each direction up to a certain number of terms in the series expansion and for discrete time steps, thus:

$$u(x, y, t) \approx \widetilde{u}(x_i, y_j, t_n)$$

where

$$x_i = (i-1)\delta \qquad\qquad y_i = (j-1)\delta$$

37

for $i, j = 1, 2, 3, \ldots$. It turns out that taking 300 terms of the expansion is sufficient to obtain a converged solution to machine precision. The program has been tested to recover the initial condition for $t \downarrow 0$. To be mathematically correct and to avoid Gibbs oscillations (sharp corners in the initial condition !), the final code however directly outputs the initial condition for $t = 0$ and only employs the series expansion for $t > 0$. This is the mathematically correct way since to solution of a PDE is only defined for $t > 0$ anyway.

To be able to check the boundary condition, the conservation of mass in the domain Ω is considered. The total mass is hereby calculated as the sum of the masses of all grid cells. A grid cell is defined as the parallelepiped between 4 grid points, thus:

$$C_{i,j} = \{(i, j) : (i-1)\delta \leqslant x < i\delta,\ (j-1)\delta \leqslant y < j\delta\}$$

and its mass $m(C_{i,j})$ is given by the concentration at its center times its volume where the center concentration c_c is taken to be the mean of the concentrations at the surrounding grid vertices:

$$m(C_{i,j}, t_n) = c_c(t_n)\delta^2$$

$$c_c(t_n) = \tfrac{1}{4}\left(\widetilde{u}(x_i, y_j, t_n) + \widetilde{u}(x_i + 1, y_j, t_n) + \widetilde{u}(x_i, y_j + 1, t_n) + \widetilde{u}(x_i + 1, y_j + 1, t_n)\right)$$

110-116

This is the consistent way of evaluating the mass. Just taking the sum of $\widetilde{u}$ over all grid points would not be sufficient (in fact, the results would be wrong).

In order to be able to get a reference FRAP curve, a corresponding output facility is implemented in the code. The FRAP value at time t_n is taken to be the mean concentration in the bleached box B, thus:

$$F(t_n) = \frac{1}{\mathcal{L}^3(B)} \sum_{(i,j)\in B} m(C_{i,j}, t_n)$$

117

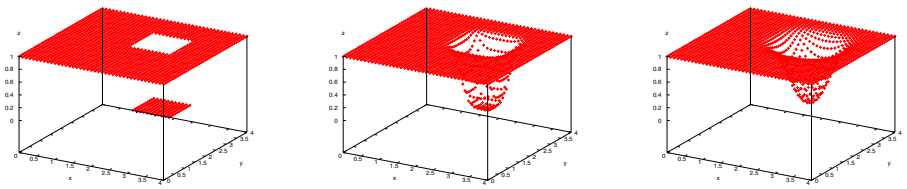
where $\mathcal{L}^3(B)$ means the volume of B. This is consistent with above way of evaluating the mass and physically meaningful since the light intensity of the fluorescence is proportional to the concentration of the fluorophore in the area in question. Plotting F vs. t now approximates the exact FRAP curve of the test case to machine precision.

For the concrete test case the numerical parameter values from [Sbalzarini (2001)] are taken. Hence Ω is the cube of edge length 4 and the bleached box is $B = [2.0, 3.0] \times [2.0, 3.0] \times [0.0, 4.0]$. The initial concentration in $\Omega \setminus B$ is chosen to be unity, the diffusion coefficient is 0.03. For the numerical evaluation, a cartesian 41×41 grid is chosen and the solution is calculated on all grid points every 0.05 s. Table 8.1 summarizes these problem parameters.

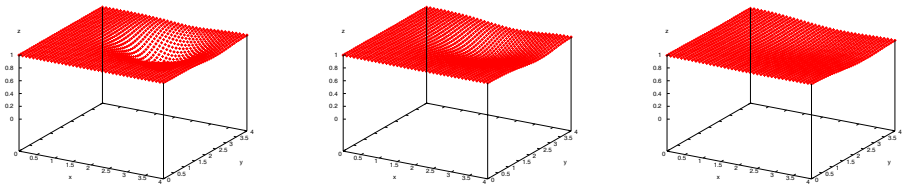
Parameter	Value
a	2.0
b	3.0
c	2.0
d	3.0
L	4.0
c_0	1.0
D	0.03
Grid	41×41
Time step	0.05
Number of terms	300

Table 8.1: Test case parameters

Figure 8.3 shows snapshots of the exact solution at times $t = 0$, $t = 0.3$, $t = 1$, $t = 5$, $t = 10$ and $t = 20$. Figure 8.4 shows the reference FRAP curve $F(t_n)$. The dashed blue line indicates the steady state value for $t \to \infty$. Due to the no flux boundary condition this is lower than 1 as Ω is initially not completely filled. Figure 8.5 shows the total mass inside Ω. As expected from the boundary condition, it is conserved with an absolute error (due to the finite number of terms that have been taken for the series expansion) of 10^{-14} which is comparable to the machine epsilon. Moreover, a movie animation of the exact solution is available on volume 1 of the companion CD of this report (see appendix A).



$t = 0.0$ $t = 0.3$ $t = 1.0$



$t = 5.0$ $t = 10.0$ $t = 20.0$

Figure 8.3: Snapshots of the exact solution at different times

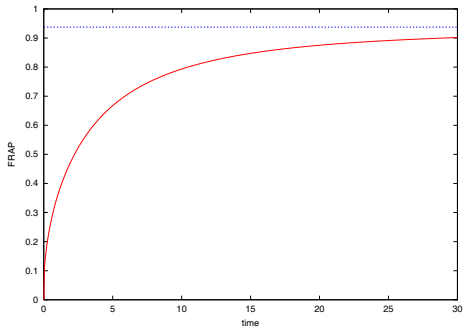


Figure 8.4: Analytic FRAP curve (red) with asymptote 0.9375 (blue)

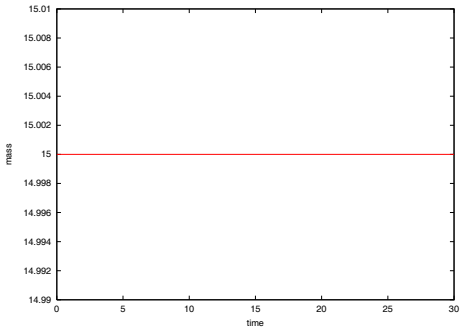


Figure 8.5: Total mass for analytic solution

8.3 A finite difference code

To double-check the exact solution derived in section 8.2 and to have an additional benchmark for the PSE simulation code, a simple high resolution finite difference code is implemented for the test case presented in section 8.1.

To keep the possibility of programming errors as low as possible, the standard CRAYFISHPAK library[3] version 2.0 is used. This library contains fast solvers for the non-homogeneous linear Helmholtz equation:

$$\nabla^2 u + \lambda u = f$$

Doing an implicit Euler time discretization of the diffusion equation leads to Helmholtz's equation for u_{n+1} as follows:

$$\begin{aligned} \frac{\partial u}{\partial t} &= D\nabla^2 u \\ \text{Euler:} \qquad u_{n+1} &= u_n + D\delta t \nabla^2 u_{n+1} \\ \Rightarrow \nabla^2 u_{n+1} - \frac{1}{D\delta t} u_{n+1} &= -\frac{1}{D\delta t} u_n \end{aligned}$$

Solving Helmholtz's equation with $\lambda = -(D\delta t)^{-1}$ and $f = \lambda u_n$ for the new solution u_{n+1} in every time step thus corresponds to solving the diffusion equation with implicit time stepping.

Neumann zero flux boundary conditions are taken on all 6 walls of the cubic computational domain Ω, which is reflected by boundary condition type 3 and boundary value 0 in CRAYFISHPAK. To set the initial condition given in equation 8.1, an initialization routine (`H3GCIS` in this case) is called once. Inside the time loop, the fast solver `H3GCSS` for regular cartesian 3D grids is called repeatedly. The complete Fortran source is given in appendix C.2.2.1.

The test case is run with the geometry parameters given in table 8.1 on a regular cartesian $81 \times 81 \times 81$ mesh with time step $\delta t = 0.005$ and $D = 0.03$. Figure 8.6 shows the resulting FRAP curve in comparison to the one of the exact solution. It can be seen that the finite difference code stays below the exact solution due to numerical dissipation. The total mass in the domain is again calculated as the sum of the masses of all grid cells where the mass of a grid cell is defined to be the concentration at its center times its volume. The concentration at the center is

[3] Commercial library by Green Mountain software, Madeira Beach, FL.

taken to be the mean value of the concentrations of the 8 surrounding grid vertices (cf. section 8.2). As a check, figure 8.7 shows the total mass vs. time. It is conserved up to an absolute error of 10^{-9} which reflects the no flux boundary condition very well.

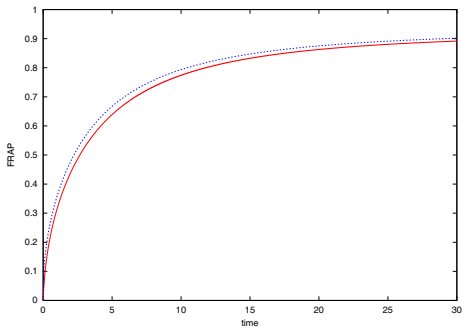


Figure 8.6: FRAP curve of FD code (red) compared to exact solution (blue)

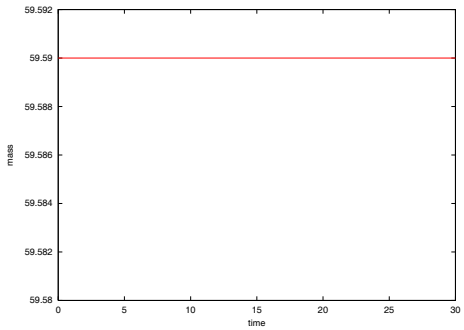


Figure 8.7: Total mass for the FD code

The discretisation used is first order in time and second order in space and fails to recover the error introduced by the very steep gradients of the initial condition. This can be seen when running it with starting time $t_0 = 0.1$ rather than from the beginning. The initial condition given by equation 8.1 is now replaced with the exact solution at time t_0. Since this is smooth, the finite difference code now performs very well indicating that the error observed in figure 8.6 indeed stems from the initial condition. Figure 8.8 shows the corresponding FRAP curve in comparison to the exact solution and the FRAP curve obtained when starting at $t_0 = 0$. Figure 8.9 shows the total mass vs. time, which is conserved within an absolute error of 10^{-9} again.

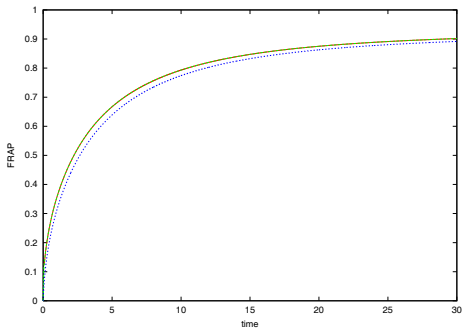


Figure 8.8: FRAP curve of FD starting at $t_0 = 0.1$ (red) compared to exact solution (green) and FD starting at $t_0 = 0$ (blue)

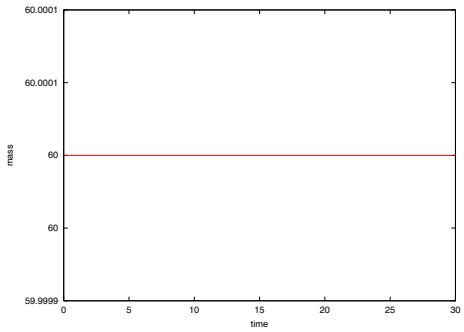


Figure 8.9: Total mass for FD code starting at $t_0 = 0.1$

As the time discretization is implicit, it is possible to use very large time steps, exceeding the CFL limit[4] of the problem and still getting a bounded solution. To check this, the previous run is repeated with $\delta t = 1.0$ which is about two orders of magnitude larger than the CFL limit $\delta t < \frac{\delta x^2}{2D} = 0.041667\ldots$. It can be seen from figure 8.10 that the solution indeed is bounded and does not deviate grossly from the previous one. The mass is still conserved up to an absolute error of 10^{-9}.

[4] stating that $2D\delta t/\delta x^2 \leqslant 1$ for $x \in \mathbb{R}^n$

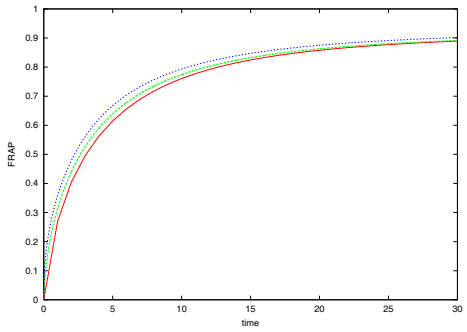


Figure 8.10: FRAP curve for $\delta t = 1.0$ (red) compared to FRAP curve for $\delta t = 0.005$ (green) and exact solution (blue)

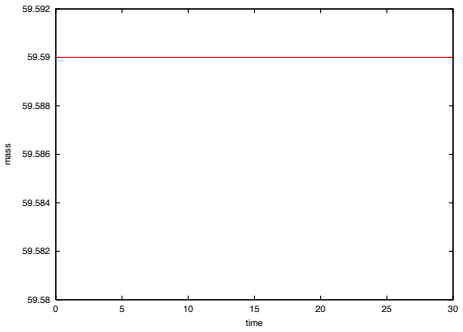


Figure 8.11: Total mass vs. time for the large time step $\delta t = 1.0$

8.4 Validation of random walk and PSE

In order to validate the PSE simulation method as described in section 7.2 as well as the random walk method outlined in section 7.1, they are applied to the test case introduced in section 8.1, again using the numerical parameter values given in table 8.1. Furthermore, comparative timings for the PSE and the random walk method are made to estimate their performance.

Recall that the FRAP value is the normalized mean concentration in the bleached box. Since the particles in a PSE simulation carry mass as their strength, it is simply the sum of the strengths of all the particles in the bleached box divided by the volume of the bleached box, thus:

$$F(t_n) = \frac{1}{\mathcal{L}^3(B)} \sum_{p \in B} v_p^h c_p^h(t_n) = \frac{1}{\#\{p : p \in B\}} \sum_{p \in B} c_p^h(t_n)$$

since all the particles have the same volume and they completely fill the bleached box. Figure 8.12 shows a comparison of the FRAP curves from the analytic solution (green) and the one obtained with the PSE algorithm (red). For the PSE, 64000 particles have been used, placed on a regular cartesian grid with grid spacing 0.1. The core size of the particles is determined from the following convergence requirement:

$$\frac{h}{\epsilon} < \text{overlap}$$

where h is the inter-particle spacing (thus 0.1) and the overlap parameter is set to 0.9, which means that the particles are required to always overlap by at least 11% of their core size. Evaluating this expression gives $\epsilon > 1/9$, so it is set to $\epsilon = 0.1118$ giving $\epsilon^2 = 0.0125$. The simulation is run for $T = 600$ time steps with a step size of $\delta t = 0.05$ which is smaller than the time integration stability requirement

$$\delta t < \frac{h^2}{2D} \approx 0.16667\ldots$$

with the diffusion constant $D = 0.03$ and the inter-particle spacing $h = 0.1$.

A very good coincidence of the FRAP curves of the PSE algorithm and the exact solution can be observed even for small times. For times larger than 4 they hardly differ, resulting in a relative error of $-5.64 \cdot 10^{-4}$ at time 30. This is about 20 times

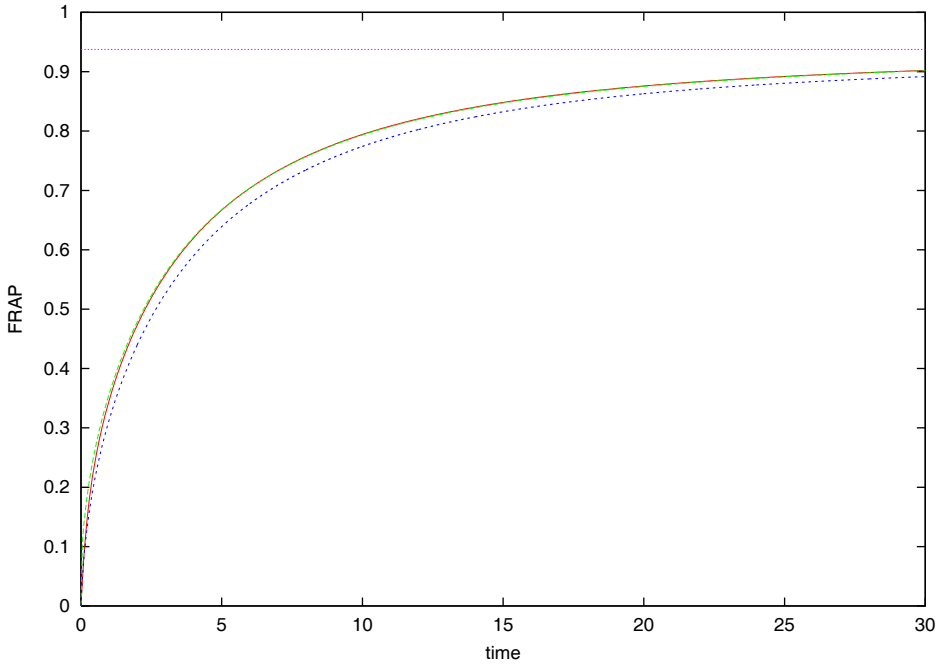


Figure 8.12: PSE FRAP curve (red) compared to analytic curve (green) and finite difference result (blue). The asymptotic value 0.9375 is shown in pink

better than the finite difference result with a relative error of $-1.10 \cdot 10^{-2}$ (blue curve). Both curves stay below the analytic one due to numerical dissipation.

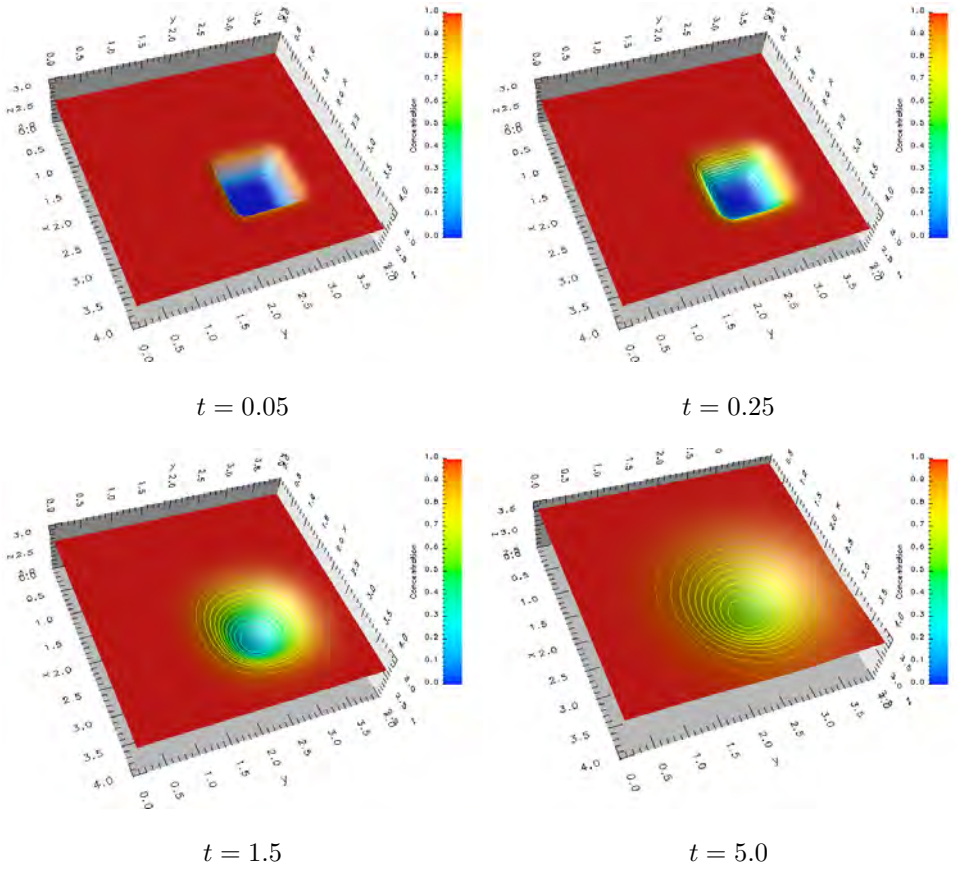
The RMS error is defined as:

$$E_{\text{RMS}} = \sqrt{\frac{1}{T}\sum_{n=1}^{T}\left(F(t_n) - F^*(t_n)\right)}$$

where $F(t_n)$ is the simulated FRAP value at time $t_n = (n-1)\delta t$, $F^*(t_n)$ is the exact value from the analytic solution at the same time and T is the number of time steps. For the PSE, an RMS error value of $E_{\text{RMS}} = 5.705 \cdot 10^{-3}$ results and for the finite difference code one of $E_{\text{RMS}} = 2.131 \cdot 10^{-2}$, again about an order of magnitude worse. Figure 8.13 shows snapshots of the PSE simulation at different times. The visualizations have been done using the techniques described in section 7.3.

Figure 8.14 shows the comparison between the FRAP curves of the random walk code and the analytic solution. The random walk has been simulated using 10^6 particles initially placed at random locations. It is run for 4000 time steps with a step size of $\delta t = 0.005$. However, since the random walk code simply outputs the relative number of particles in the bleached box whereas the analytic curve reflects the mean concentration, some post processing is needed to make the outputs comparable. Let N be the total number of particles in a random walk simulation and p the unknown mass (or strength) per particle. Due to the fact that the volume of the bleached box is 1/16 of the total volume of the cubic domain, the initial concentration outside the bleached box is given by:

$$c_0 = \frac{16 \cdot Np}{15 \cdot V}$$



$t = 0.05$ $t = 0.25$

$t = 1.5$ $t = 5.0$

Figure 8.13: Snapshots of concentration distribution for a PSE simulation of the box test case

with V being the total volume of the domain. As $c_0 = 1.0$ by construction of the problem, this can be solved for p. The instantaneous mean concentration (thus the FRAP value) in the bleached box is then given by:

$$c(t) = \frac{16 \cdot N_B(t)p}{V} = 15 \cdot c_0 \frac{N_B(t)}{N}$$

where $N_B(t)$ denotes the number of particles in the bleached box. Therefore, the output of the random walk code N_B/N has to be multiplied by 15 before comparing it to any other FRAP curve. It can be seen from figure 8.14 that the random walk curve follows the one of the analytic solution pretty well resulting in a relative error of $4.66 \cdot 10^{-3}$ at time 20 which is about 3 times less than the finite difference code with a relative error of $-1.41 \cdot 10^{-2}$. The RMS error for the random walk is $3.976 \cdot 10^{-3}$, thus of comparable accuracy as the PSE. The random walk results could be further improved by performing several runs and subsequent ensemble averaging to obtain smoother FRAP curves.

Another disadvantage of the random walk method besides the missing smoothness of the results is its slow convergence for increasing numbers of particles. According to [Cottet & Koumoutsakos (2000)], the error of a random walk simulation decreases like $\sqrt{N}$ if the number of particles N is increasing. Thus an increase in the number of particles by a factor of 10 will only reduce the error by a factor of about 3 as can be seen from the RMS error values in table 8.2. The PSE on the other hand converges with N^2 meaning that an increase in the number of particles by a factor

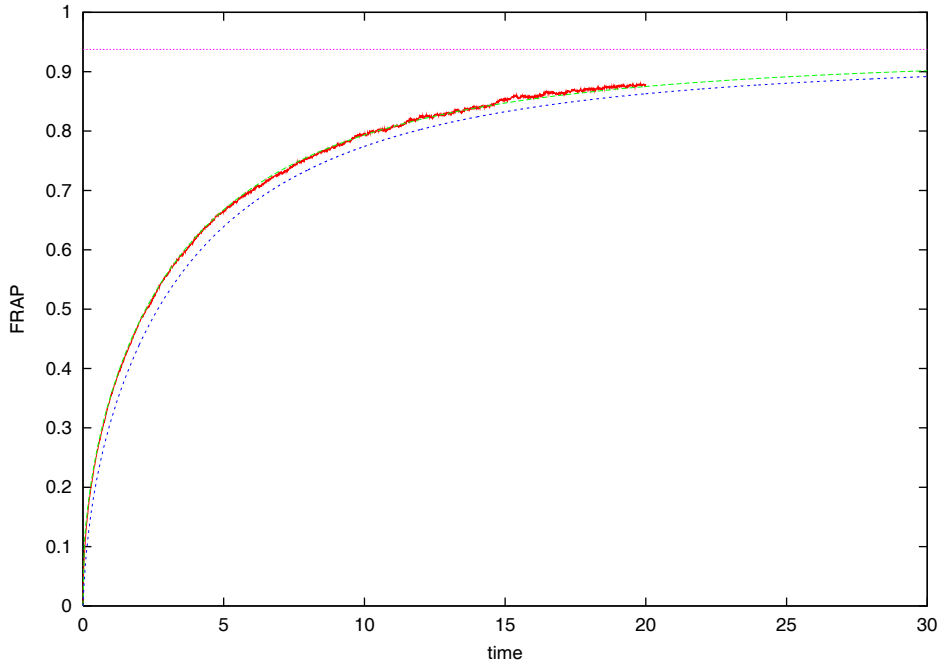


Figure 8.14: Random walk FRAP curve (red) compared to analytic curve (green) and finite difference result (blue). The asymptotic value 0.9375 is shown in pink

of 10 will reduce the error by a factor of 100. Figure 8.15 shows the FRAP result for a random walk run with 10^4 particles and figure 8.16 for one with 10^5 particles. Figure 8.14 has been created using 10^6 particles. Despite their variance, the means of all curves follow the exact solution quite well, indicating that the random walk code works correctly.

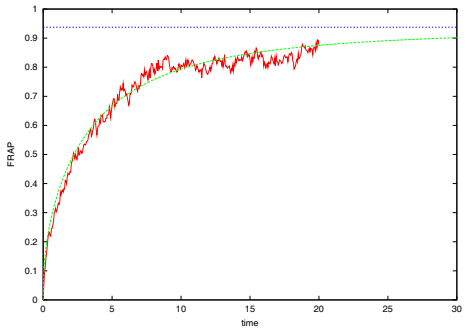


Figure 8.15: Random walk FRAP curve with 10000 particles (red) compared to analytical solution (green)

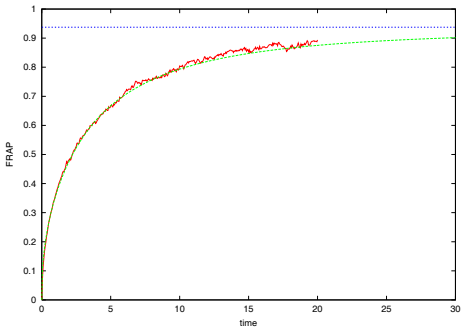


Figure 8.16: Random walk FRAP curve with 100000 particles (red) compared to analytical solution (green)

To further compare the PSE and random walk methods, timings are made. Both codes are compiled (using maximum optimization `-O3 -fast` and the Portland Group Fortran 90 compiler) and run on the same machine, a 1.4 GHz AMD Athlon

Run	RMS error
Random walk, $N = 10^4$	0.02896
Random walk, $N = 10^5$	0.01076
Random walk, $N = 10^6$	0.003976
Finite difference	0.02131
PSE, $N = 64000$	0.005708

Table 8.2: RMS error comparison

yielding 2792 MIPS[5]. Table 8.3 summarizes the results. The first two PSE runs with 64000 particles have been performed with an inter-particle spacing of 0.1 and a particle core size of 0.1118, just as before. The difference is that for the first run the code was compiled in its serial version whereas for the second run, the parallel version including MPI has been used, although run on single processor. It can be seen that the communication overhead added by MPI amounts to about 2.25 s per time step. Therefore, running in parallel is only useful for problems at least as large as this test case.

Run	CPU time per time step
PSE without MPI, $N = 64000$	3.888 s
PSE with MPI, $N = 64000$	6.141 s
PSE without MPI, $N = 10648$	0.783 s
Random walk, $N = 10^5$	0.494 s
Random walk, $N = 10^6$	5.149 s
Random walk, $N = 10^7$	57.024 s

Table 8.3: Comparative timings

The first random walk run with 10^5 particles is faster than the PSE but cannot be used without averaging, which makes more than one run necessary. The second run with 10^6 particles is comparable to the PSE method in accuracy but already slower than its serial version. The last run with 10^7 particles finally is about 15 times slower than the PSE.

In addition to this, it can be seen that the time needed for the random walk scales almost linearly with the number of particles used. The PSE even scales sub-linear due to the small number of particles compared to the computational cost of all auxiliary and bookkeeping routines.

Taking all the results of this chapter into account leads to the decision that for the simulations to come, the PSE is the algorithm of choice. The random walk will not be used any further due to the following disadvantages:

- It is slower than the PSE at comparable accuracy
- Due to its $\sqrt{N}$ convergence, the number of particles needed to simulate diffusion in a full ER exceeds 10^{10} thus breaking all computer memory limits. (fewer particles can be used if several runs are performed and averaged but this will again cost time.)
- The random walk code is, at the current stage of development, not parallelized, excluding the (necessary) use of distributed memory machines.

[5]Million instructions per second. Determined using the Linux kernel time calibration loop

- The geometry handling is the random walk code is not optimized. Thus the code is $\mathcal{O}(TNM)$ where M is the number of triangles in the surface, N the number of particles and T the number of time steps. This should be compared to the PSE for which an optimized geometry preprocessor has been written making it $\mathcal{O}(TN \log N + N \log M)$.

Random walk nevertheless has been applied by [Ölveczky & Verkman (1998)] and [Sbalzarini (2001)] to the problem studied in this work and if future work makes the use of sources, sinks, tubes or separator planes necessary, one would have to revert to random walk simulations, optimizing the algorithm wherever possible.

Chapter 9

FRAP simulations

9.1 PSE simulations in all ER samples

In this chapter, the PSE simulation method as described in section 7.2 will be applied to the simulation of diffusion in reconstructed ER geometries (cf. chapter 2). The random walk code will not be used hereafter for the reasons mentioned at the end of section 8.4. The only difference to the PSE simulations conducted for the text case in section 8.4 emerges from the fact that the bleached area is no longer a box but rather the set intersection of a box with the space Ω enclosed by the ER membrane. Thus we define the bleached area as:

$$A = B \cap \Omega$$

and the FRAP value is calculated as:

$$F(t_n) = \frac{1}{\mathcal{L}^3(A)} \sum_{p \in A} v_p^h c_p^h(t_n) = \frac{1}{\#\{p : p \in A\}} \sum_{p \in A} c_p^h(t_n)$$

This definition includes the box test case in the limit $A = B$. To initialize the particles, all ER samples are preprocessed as described in section 7.2.5. As we wish to simulate a FRAP experiment, the proper initial condition is given by:

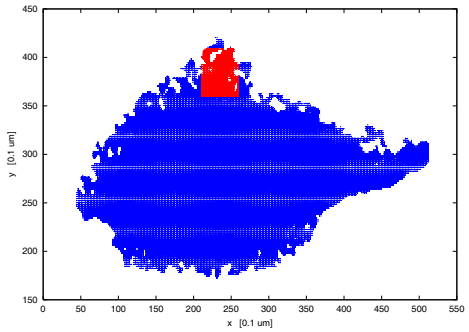
$$c_0^h(x) = \begin{cases} c_0 & \text{if } x \in int(\Omega) \setminus A \\ 0 & \text{if } x \in A \end{cases} \tag{9.1}$$

Without loss of generality, the constant initial concentration outside the bleached are is chosen to be $c_0 = 1$ as this simply corresponds to normalizing the FRAP curves. Figure 9.1 shows the initial particle distributions for all the ER samples considered. The blue points symbolize particles of initial concentration 1, the red points those of initial concentration 0 (i.e. those inside A). The assumption of a homogeneous initial concentration distribution outside the bleached area A seems feasible due to the following facts:

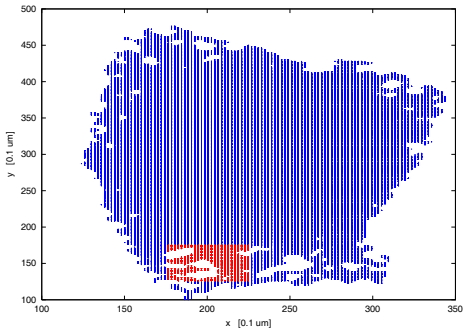
1. After transfection, the cells are incubated for 12 hours. During this time they express the green fluorescent protein which freely diffuses in the ER lumen and fills it completely. The time scale of diffusion is about 5 s. Therefore a homogeneous distribution inside the ER can be assumed after 12 hours.

2. The experimenter chooses "sane" cells, i.e. cells which exhibit homogeneous fluorescence and do neither under- nor over-express the protein.

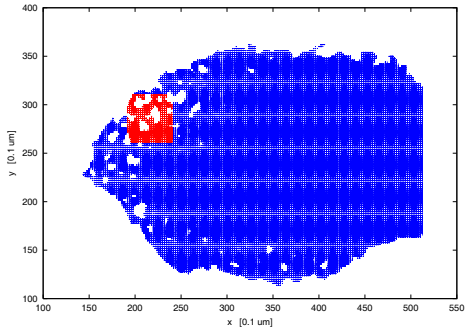
3. The pixel intensities of the pictures taken from the microscope are normalized to the level of homogeneous concentration before any FRAP data is calculated.



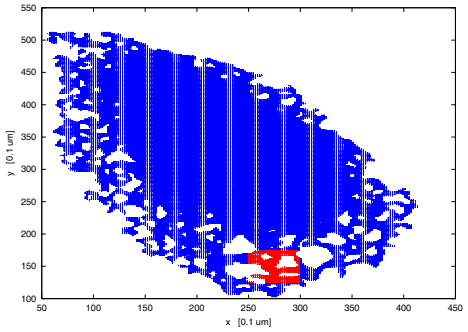
bip2



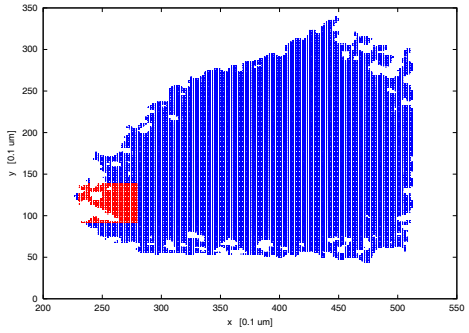
clx



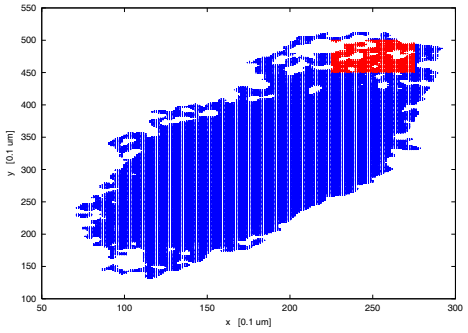
erp57



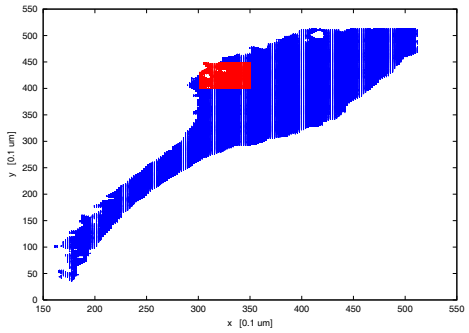
erp572



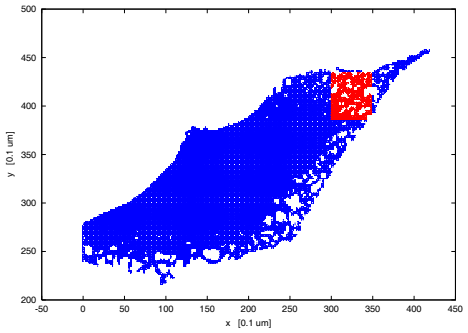
erp573_1



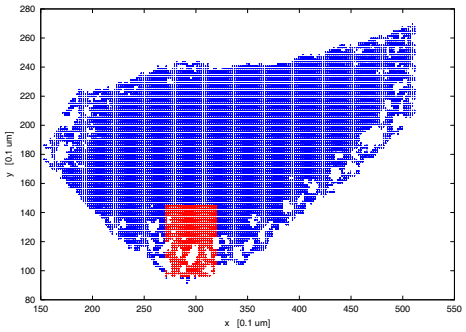
erp573_2



erp573_3



erp574_1



erp574_2

Figure 9.1: Initial particle distributions for all ER samples. Blue points correspond to particles of concentration 1, red points of those of concentration 0.

Please refer to figure 5.10 in section 5.3 for shaded surface views of the geometries. The bounding boxes of all the samples are given in table 9.1 and table 9.2 states the geometries of the bleached boxes used. All boxes are of the form $[a, b] \times [c, d] \times [z_l, z_h]$ (recall that the actual bleached area is the set intersection of the bleached box and the effective ER geometry).

Sample	x_l	y_l	z_l	x_h	y_h	z_h
bip2	43.6768	171.4180	−1.7248	512.7590	421.4890	17.3869
clx	122.9590	99.3180	−2.0038	344.7140	477.0080	17.3165
erp57	142.7870	112.7570	−2.0900	513.4100	362.9470	15.5391
erp572	54.8070	102.2590	−1.8534	414.2720	512.7500	17.2687
erp573_1	225.6450	42.7788	−1.8912	513.1250	339.9230	18.3436
erp573_2	70.3277	131.2920	−1.9464	292.5250	512.5840	17.8942
erp573_3	160.5760	34.8480	−1.9580	512.8440	513.3540	17.9499
erp574_1	−1.5552	215.3650	−2.1574	419.5930	458.9890	30.0631
erp574_2	152.2640	90.8761	−2.0212	512.4500	270.2680	30.2559

Table 9.1: Bounding boxes of all ER sample domains

Sample	a	c	b	d
bip2	210.0	360.0	260.0	410.0
clx	175.0	125.0	225.0	175.0
erp57	190.0	260.0	240.0	310.0
erp572	250.0	125.0	300.0	175.0
erp573_1	230.0	90.0	280.0	140.0
erp573_2	225.0	450.0	275.0	500.0
erp573_3	300.0	400.0	350.0	450.0
erp574_1	300.0	385.0	350.0	435.0
erp574_2	270.0	95.0	320.0	145.0

Table 9.2: Positions and sizes of the bleached boxes B

All PSE simulations are run for the same value of the diffusion constant $D = 7.5$ in order to be able to study the influences of geometry. Running at a different diffusion constant would simply correspond to stretching the time axis accordingly.

This can be seen when recalling that all possible solutions to the diffusion equation are linear combinations of its elementary solution (i.e. its Green's function). This elementary solution is given by:

$$K(x,\xi,t) = \frac{1}{(4\pi Dt)^{n/2}} \exp\left[-\frac{(x-\xi)^2}{4Dt}\right]$$

and only contains Dt as a product pair. Enlarging D by a certain factor and at the same time reducing t by the same factor therefore leaves its function value unchanged. Since diffusion is a linear problem, this applies not only to the single elementary solutions but also to any linear combination of them, thus to all solutions to the diffusion equation. Performing a diffusion run at $2 \cdot D$ is therefore the same as running at D and replacing t by $1/2 \cdot t$.

The initial inter-particle spacings in x, y and z direction (termed h_1, h_2 and h_3, respectively) are chosen according to the geometrical resolution desired. Since the domains are quite flat compared to their x and y extensions (cf. table 9.1), the particles are placed denser in the z direction to have enough resolution. x and y spacings are chosen so as to be smaller than the smallest occurring triangle edge. This is recommended to resolve the whole geometry as described by the triangulated surface and to increase the accuracy of the boundary condition handling[1]. Table 9.3 summarizes the particle grid spacings and the resulting total numbers of particles for the different simulation runs.

Sample	h_1	h_2	h_3	N
bip2	2.0	2.0	0.4	493850
clx	2.0	2.0	0.4	488054
erp57	2.0	2.0	0.4	509578
erp572	2.0	2.0	0.4	545859
erp573_1	2.0	2.0	0.4	473466
erp573_2	1.3	2.0	0.4	495373
erp573_3	2.0	2.0	0.4	335480
erp574_1	2.0	1.7	0.4	481922
erp574_2	2.0	1.6	0.4	504770

Table 9.3: Inter-particle spacings and total numbers of particles

The time step for all the simulations is chosen according to the time integration requirement requirement (see [Zimmermann, Koumoutsakos & Kinzelbach (2001)]):

$$\delta t < \frac{\min_{i=1,2,3} h_i^2}{2D}$$

where h_i stands for the inter-particle spacing in the i^{th} direction. As the smallest occurring inter-particle spacing is 0.4 for all simulations and D is constant as well, a time step of $\delta t = 0.01 < 0.01067\ldots$ is chosen for all runs. All simulations are run for $T = 2000$ time steps to a final time of $t_{final} = 20.0$. The PSE kernel function's core size is determined using the requirement:

$$\epsilon > \frac{\max_{i=1,2,3} h_i}{\text{overlap}}$$

[1] If the particles are too far apart, the core size ϵ becomes large and it will eventually be the case that boundary image particles of one part of the domain overlap with real particles of another part or – put differently – that particles in one ER filament directly interact with those in another one.

with overlap $\leqslant 1$, this is a generalized version of the requirement stated in section 7.2.1. For safety reasons, we choose overlap = 0.901 for all simulations. As the maximum occurring inter-particle distance is 2.0 for all samples, the core size is chosen to be $\epsilon = 2.222$ for all runs. Table 9.4 summarizes the parameters that are common to all simulations of this chapter.

Parameter	ϵ	δt	D	t_{final}
Value	2.222	0.01	7.50	20.0

Table 9.4: Common parameters for all PSE runs

In order to be able to compare the FRAP curves of the simulation runs amongst each other, they are normalized to their respective steady-state value. Since the bleached areas of the different ER samples contain different numbers of particles and the total number of particles also varies, the different FRAP curves will have different asymptotic levels as $t \to \infty$. This is the case due to the no-flux boundary condition[2] and the fact that the total mass in the domain is conserved. Initially, the total mass in the system is given by:

$$m_t = \sum_{p=1}^{N} v_p c_p^h = v_p(N - N_B)$$

where $N_B = \#\{p : p \in B\} = \#\{p : p \in A\}$ is the number of particles inside the bleached area. The last equality makes use of the initial condition as given by equation 9.1. The asymptotic value of the concentration is now given by homogeneously distributing this mass among all the particles, thus:

$$c_\infty = \frac{m_t}{N v_p} = \frac{N - N_B}{N}$$

Table 9.5 lists the values for the ER samples under consideration. The gap between the asymptotic FRAP value and the pre-bleach level (1 in this case) is called *immobile fraction* in biology. It is observed due to the no-flux boundary condition, the conservation of mass and the finiteness of the domain. In fact, the simulations conducted in this chapter have an average immobile fraction of 2.48%, which is in nice coincidence with the experimentally observed 2 to 3% by Anna Mezzacasa.

Another interesting property to consider is the volume-filling coefficient of the ER geometry in the bleached box. This number is the fraction of the bleached box's volume that is actually taken up by the ER lumen (i.e. is filled with particles). It is formally defined as:

$$\phi_0 = \frac{\mathcal{L}^3(\Omega \cap B)}{\mathcal{L}^3(B)}$$

For the box test case considered in section 8.4, this value is $\phi_0 = 1$ by definition as the domain Ω and its bounding box are identical. For the ER geometries however, this is no longer the case and the volume-filling coefficient can be approximated as:

$$\phi_0 = \frac{N_B}{N_g} = \frac{h_1 \cdot h_2 \cdot h_3 \cdot N_B}{(b-a)\,(d-c)\,(z_h - z_l)}$$

where N_g stands for the number of grid points inside the bleached box B. The values for all the ER samples are also listed in table 9.5. It can be seen that the ER

[2]This is the biologically correct boundary condition since ER lumen proteins do not spontaneously cross the ER membrane.

typically fills about 1/3 of the space in the area where the bleaching takes place. The effects of geometry on diffusion are therefore expected to be significant since diffusion is not free in space but restricted to about 1/3 of it.

Sample	N	N_B	N_g	c_∞	ϕ_0
bip2	493850	11892	29375	0.9759	0.4048
clx	488054	12195	30000	0.9750	0.4065
erp57	509578	6396	27500	0.9874	0.2326
erp572	545859	7767	29375	0.9858	0.2644
erp573_1	473466	9950	31250	0.9790	0.3184
erp573_2	495373	12904	46550	0.9740	0.2772
erp573_3	335480	15942	30625	0.9525	0.5206
erp574_1	481922	12177	58000	0.9747	0.2099
erp574_2	504770	23284	62000	0.9539	0.3755

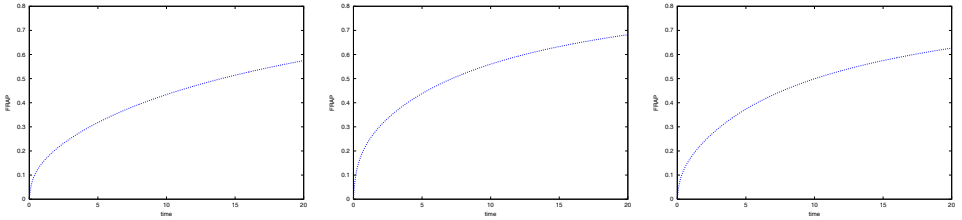
Table 9.5: Asymptotic FRAP values and volume-filling coefficients

Having now all the preliminary information, the simulation results can be discussed. Figure 9.2 shows the resulting FRAP curves for all ER samples using the bleached box geometries given in table 9.2. All curves are normalized using their respective c_∞ out of table 9.5. Therefore, they all asymptotically recover to 1.0 what makes them comparable. Figure 9.3 shows all 9 normalized FRAP curves in a single plot. As expected, they differ due to geometry influences (recall that all simulations have been done using the same diffusion constant).

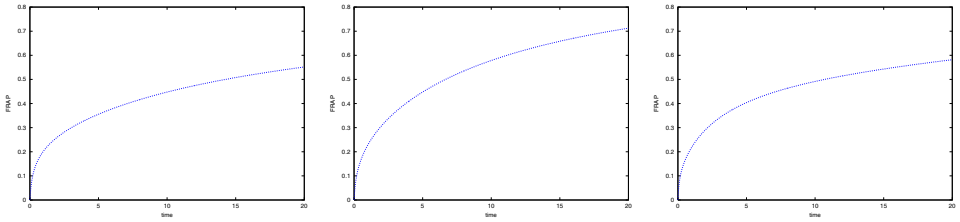
As already mentioned, a way to check if the boundary condition is treated well is to consider the total mass in the system versus time. It turned out that for all runs presented in this chapter, the total mass is conserved to machine precision (plots are therefore omitted). This is possible due to the fact that (*i*) the particle discretization of the Laplacian chosen in section 7.2.2 is conservative and (*ii*) the geometrical resolution and curvature have been taken into account when choosing the inter-particle spacings and the PSE kernel core size (i.e. the cut-off).

Figure 9.5 finally shows a comparison between the FRAP curves of the box test case (drawn in red) and of the ER sample erp574_2 (in blue). Fluorescence recovery is clearly faster for the box since diffusion is free and not limited to certain geometrical structures. The half-recovery time for the ER sample is about 3 times the one for the box test case. This is consistent with above observation that the typical volume-filling coefficient is 1/3. Both curves have again been normalized by their asymptotic value, so they ultimately recover to 1.0.

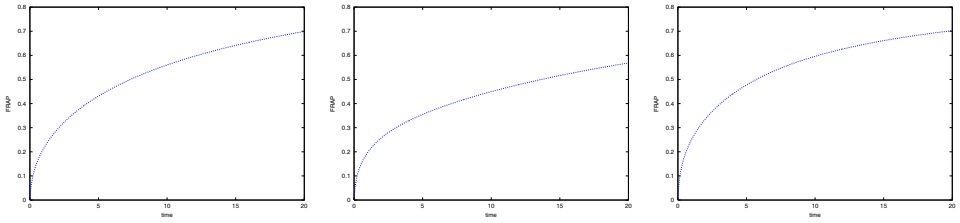
Figure 9.4 shows snapshots of the concentration distribution from the simulation in the erp572 sample at times $t = 0.25,\ 1.5,\ 3.0$ and 5.0. They have been created according to the techniques described in section 7.3. The concentration in a slice through the center of the ER parallel to the xy-plane is plotted in the third direction and color coded. The area of interest around the bleached box is enlarged.



bip2 clx erp57



erp572 erp573_1 erp573_2



erp573_3 erp574_1 erp574_2

Figure 9.2: Simulated normalized FRAP curves for all ER samples.

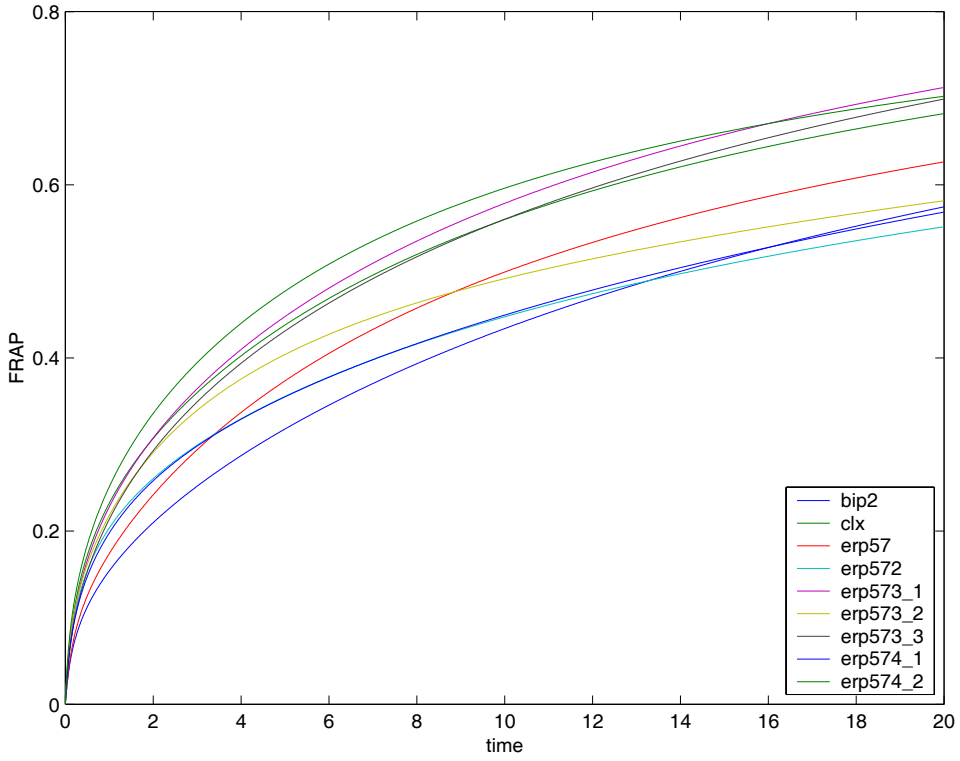
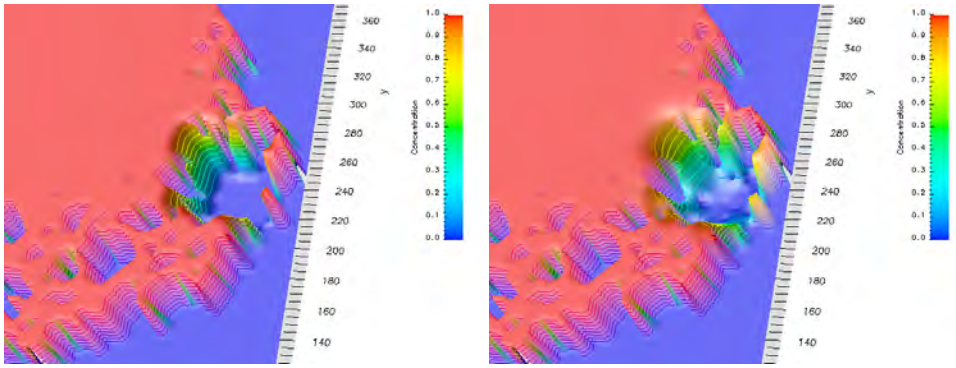
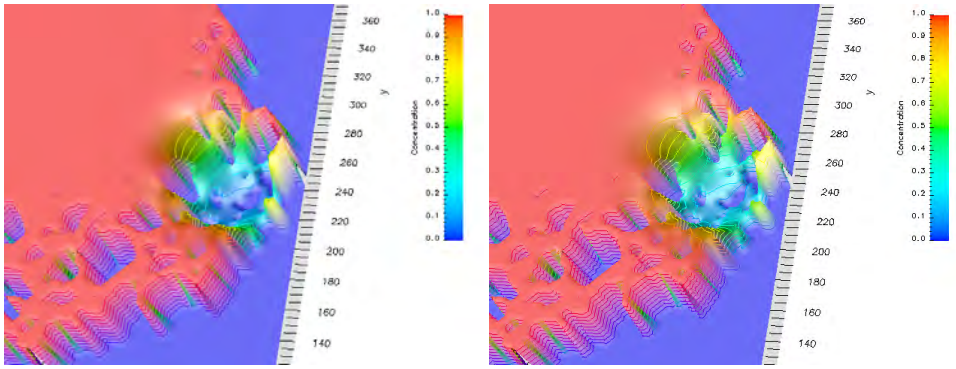


Figure 9.3: Comparison of all normalized FRAP curves



$t = 0.25$ $t = 1.5$



$t = 3.0$ $t = 5.0$

Figure 9.4: Snapshots of concentration distribution for erp572 PSE simulation

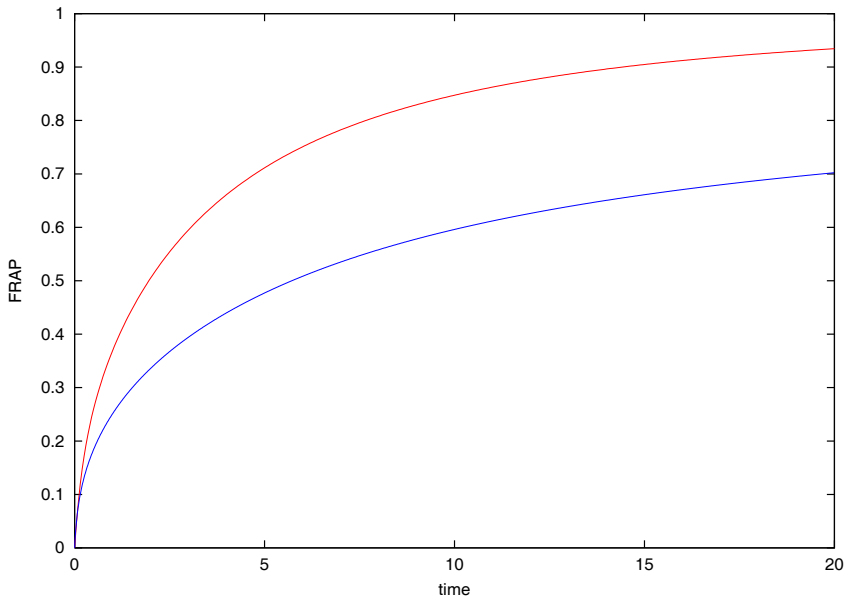
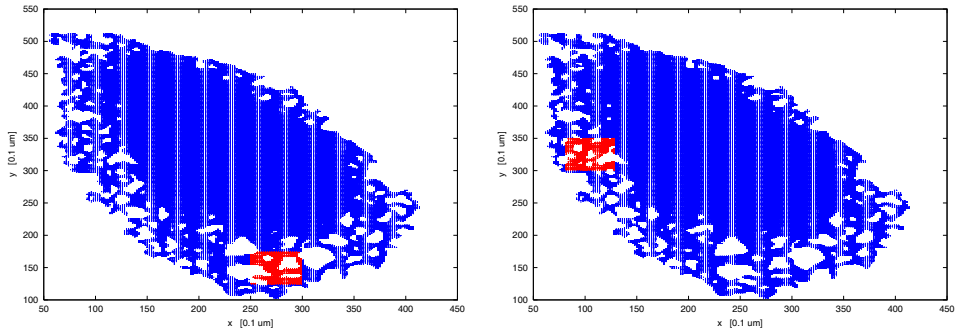


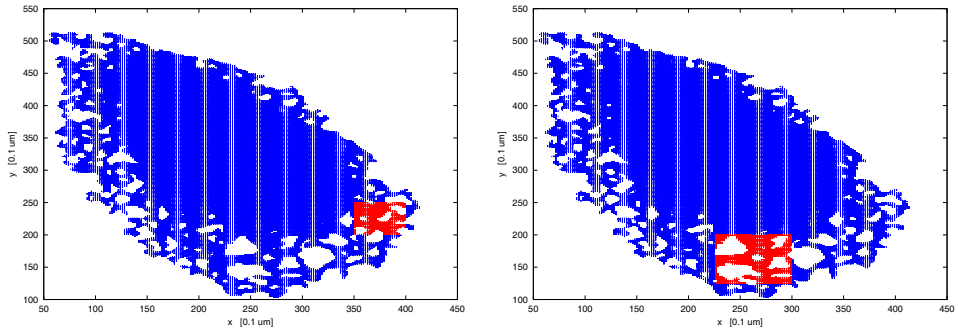
Figure 9.5: Comparison of box test case (red) and erp574_2 (blue)

9.2 Influence of bleached box geometry

It is expected that not only the geometry of the domain (i.e. the ER) influences the resulting FRAP curve but also the position and size of the bleached box. Fluorescent proteins that diffuse into the bleached area have to pass certain apertures defined by the intersection of the bleached box's surface and the ER membrane. These apertures change in size and (possibly) number if the intersecting planes are shifted. To investigate this, additional simulations are performed. All of them are to use the same ER sample geometry, namely erp572. All simulation parameters except the bleached box geometry are kept the same as for the original run in this sample (cf. section 9.1). The new runs will be termed erp5722, erp5723 and erp5724. Table 9.6 contains the corresponding bleached boxes, figure 9.6 shows the initial particle distributions. All relevant information of the original erp572 run is restated to allow convenient comparisons. Compared to the original run (see figure 9.6a), the first two have their bleached box placed at a different location. For the third one both the location and the size of the bleached box are changed as its edge length is 75 instead of 50.



a) erp572 b) erp5722



c) erp5723 d) erp5724

Figure 9.6: Initial particle distributions for different bleached box geometries. Blue points correspond to particles of concentration 1, red points to those of concentration 0.

Run	a	c	b	d
erp572	250.0	125.0	300.0	175.0
erp5722	80.0	300.0	130.0	350.0
erp5723	350.0	200.0	400.0	250.0
erp5274	225.0	125.0	300.0	200.0

Table 9.6: Comparison of bleached box geometries

Table 9.7 shows the steady-state FRAP values for all runs. The FRAP curves will again be normalized by c_∞ to be able to compare the influences of geometry. Figure 9.7 shows the resulting normalized FRAP curves from all four runs. In accordance with the theoretical predictions of [Axelrod et al. (1976)], it can be seen that position and size of the bleached box indeed do have a significant influence on the result. It seems that fluorescence recovers slower for the same value of the diffusion constant if the bleached box is larger (this would make sense as the distance a particle has to travel is larger, too). When developing new models in chapter 11, this fact will become important.

Sample	N	N_A	c_∞
erp572	545859	7767	0.9858
erp5722	545859	6459	0.9882
erp5723	545859	7447	0.9864
erp5724	545859	16886	0.9691

Table 9.7: Asymptotic FRAP values for the different bleached box geometries

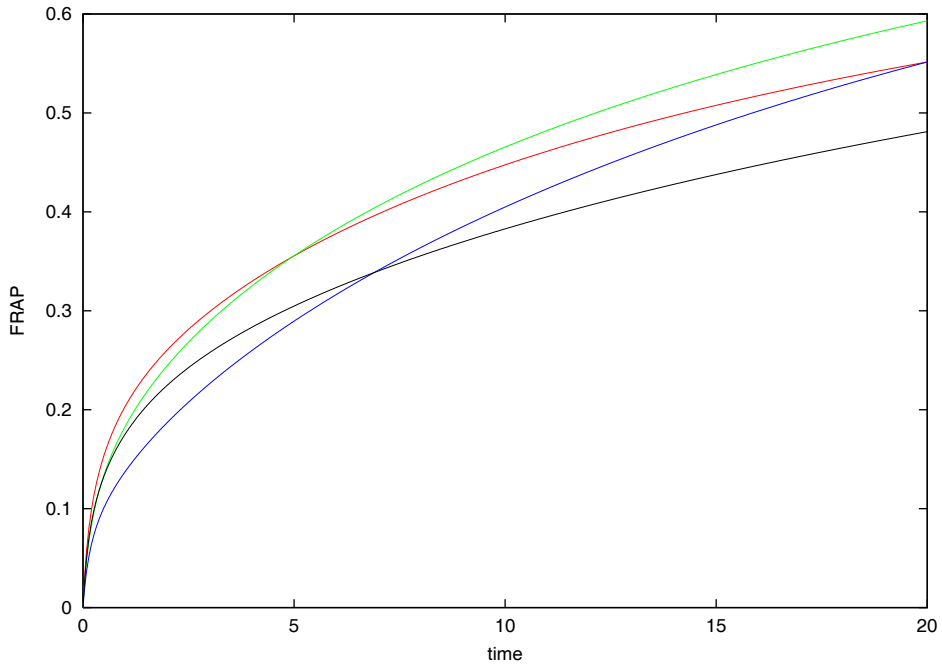


Figure 9.7: Comparison of normalized FRAP curves for different bleached box geometries. (red: erp572, green: erp5722, blue: erp5723, black: erp5724)

9.3 Timings and parallel speed-up estimation

As the PSE simulation code is parallelized using MPI, it is insightful to conduct some benchmark runs on different parallel machines to get comparative speed-up figures. The two computers used are:

- **asgard:** A distributed memory machine of type Beowulf running SuSE Linux 6.3 (kernel 2.2.14-SMP) and MPICH 1.2.1. It consists of 251 nodes with 2 Intel Pentium III 500 MHz processors and 1 GB main memory each, making a total number of 502 processors, interconnected by a 100 MBit/s fast Ethernet. The theoretical peak performance is 550 MFLOPS per processor.
- **prometeo:** A shared memory NEC SX-5/10A vector supercomputer running SUPER-UX R11.1 and MPI/SX 6.3.0. It has 10 processors sharing 64 GB of main memory. The peak performance is 8 GFLOPS per processor.

PSE simulations of the ER sample erp572 (545859 particles) are run for 20 time steps with a step size of $\delta t = 0.01$ on both machines using different numbers of processors. The time measurement on asgard is done using the `MPI_Walltime` call, on prometeo, the system's tracing facility is used. When more than one processor is used, the timing for the slowest one is reported since this is the speed-limiting factor (all other processors will have to wait for the slowest one before a new time step can be started). Tables 9.8 and 9.9 give the measured elapsed time per time step for the PSE calculations, excluding all I/O and auxiliary routines. The third column lists the *speed-up* figures as defined by:

$$S(n) = \frac{t_E(1)}{t_E(n)} \tag{9.2}$$

where $t_E(n)$ means the execution time on n processors. The last column gives the *parallel speed-up efficiency*:

$$e(n) = \frac{S(n)}{n}$$

number of processors	seconds per time step	speed-up	parallel efficiency
1	545.928		
2	282.319	1.934	96.7%
4	241.087	2.264	56.6%
8	238.932	2.285	28.6%
16	194.855	2.802	17.5%
32	127.988	4.265	13.3%
64	63.994	8.531	13.3%

Table 9.8: Parallel scalability on asgard

number of processors	seconds per time step	speed-up	parallel efficiency
1	37.327		
2	19.542	1.910	95.5%
4	13.977	2.671	66.8%
6	11.723	3.184	53.1%

Table 9.9: Parallel scalability on prometeo

Figure 9.8 visualizes the timing results for asgard in a log-lin plot. Figure 9.9 shows a plot of the parallel speed-up as defined by equation 9.2 vs. the number of processors. The dashed blue curves indicate the ideal levels for perfect scalability. Figures 9.10 and 9.11 show the same for prometeo. It can be seen that both machines scale almost perfectly when going from 1 to 2 processors. Using a higher number of processors on asgard does not really improve things any more until going above 16 CPUs. On prometeo, a more or less steady speed-up is observed even for 4 and 6 processors. This is suspected to be caused by the fact that asgard is a distributed memory machine whereas prometeo has a shared memory architecture. MPI communication on asgard has to pass the slow and potentially congested Ethernet links, on prometeo it just consists of copying within the main memory. Therefore it can be assumed that MPI does not add any significant overhead to the program on prometeo (see tracing results below), on asgard however it does. This accounts for the different scaling behavior of the two machines. To explain the deviation from the blue optimum line even on prometeo, one has to consider the quality of load balancing.

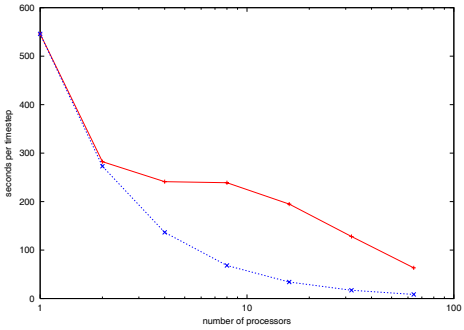


Figure 9.8: CPU time per time step for asgard (red) vs. ideal line (blue)

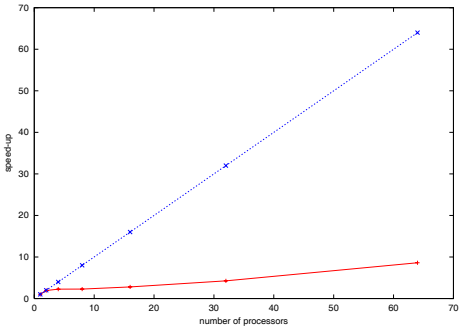


Figure 9.9: Parallel speed-up on asgard (red) compared to ideal speed-up (blue)

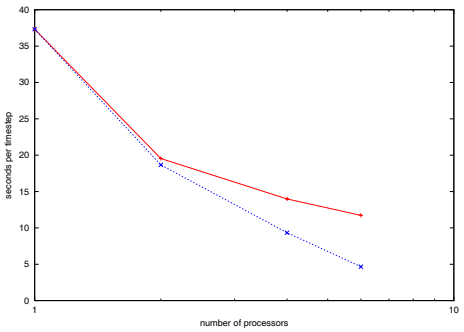


Figure 9.10: CPU time per time step for prometeo (red) vs. ideal line (blue)

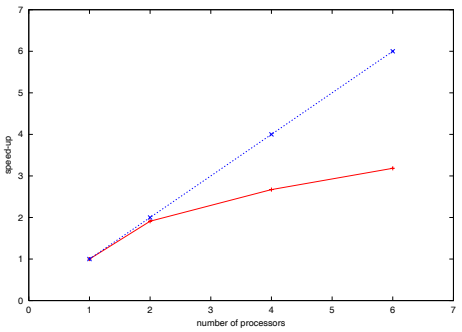


Figure 9.11: Parallel speed-up on prometeo compared to ideal speed-up (blue)

The goal of load balancing is to equally distribute the amount of work to be done among a certain number of processors. If one processor gets assigned more work than its colleagues, they will have to sit idle and wait for it at the end of the time step when the results are communicated.

One parameter to measure the quality of load balancing is the *load distribution coefficient*:

$$\ell = \frac{\max_i t_i}{\min_i t_i} \qquad i = 1, \ldots, n$$

where t_i is the time spent on processor i out of n. Perfect load balancing would thus result in value of $\ell = 1$. Table 9.10 lists the numbers for prometeo. In fact, the load balance is almost perfect for 2 processors but then deteriorates rapidly. Already using 6 processors causes an imbalance of a factor of 10. This explains the speed-up curve shown in figure 9.11. Consider the following snapshot of total execution time (i.e. for all 20 time steps) on two neighboring processors of the 4 processor case:

```
processor            pse                  diag
-------------------------------------------------
   1             103.682 sec          175.214 sec
   2             279.548 sec            0.248 sec
```

It is noticeable that the difference of the two numbers for the `pse` routine pretty much corresponds to the time spent in `diag` on the first processor. `diag` is the subroutine that is called directly after `pse` and that collects the results from the different processors to calculate the diagnostics (e.g. the FRAP value). It is therefore the first MPI communication call after the PSE calculation. Above figures show that processor 1 – which gets assigned less work and therefore completes the PSE earlier – has to wait for the results of processor 2 to arrive, thus spending a lot of idle CPU time in the subroutine `diag` which normally only takes about 0.2 s.

The reason for such a poor load balance is the use of MPI's built-in cartesian domain decomposition routines. They only take into account the edge lengths of the bounding box but not the particle densities. Since the bounding box of the present ER problem is very flat compared to its x and y dimensions (cf. table 9.1) and the particles are not homogeneously distributed inside it (but only where there is ER geometry), this is not a good way to decompose the domain. Some processors might not get any particles at all while others might have densely filled sub-domains. Load balancing thus is an issue to consider when trying to improve the code's performance. A custom load balancing algorithm would not equally distribute the volume of the bounding box but rather the number of particles.

number of processors	$\min_i t_i$ (i)	$\max_i t_i$ (i)	ℓ
2	17.525 (2)	19.542 (1)	1.115
4	5.184 (1)	13.977 (2)	2.696
6	1.170 (1)	11.723 (4)	10.020

Table 9.10: Load balance on prometeo

Using a single processor only, prometeo is about 14.6 times faster than asgard. This is in good agreement with the fact that its peak performance is 14.5 times the one of asgard, meaning that on both machines the code sustains about the same fraction of the peak performance. To estimate this fraction, the program is traced on prometeo using the `-ftrace` compiler option. The resulting output for a representative run is:

```
/tmp.speed/ivo/PSEtrace ftrace -f ftrace.out.0.0
*-------------------------*
  FLOW TRACE ANALYSIS LIST
*-------------------------*

Execution : Tue Feb  5 09:42:35 2002
Total CPU : 0:13'03"761
```

PROG.UNIT	FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	I-CACHE MISS	O-CACHE MISS	BANK CONF
pse	20	760.123(97.0)	38006.135	3647.5	2004.6	99.10	134.9	0.0706	0.9937	32.3171
init	1	12.147(1.5)	12146.724	85.2	0.4	1.92	5.2	1.9725	0.6545	0.0000
do_io	1	8.541(1.1)	8540.770	92.7	0.7	2.69	12.5	1.2781	0.0673	0.0000
sortp	20	1.908(0.2)	95.408	700.4	69.3	84.18	255.9	0.0004	0.4233	0.0013
updatep	20	0.641(0.1)	32.071	1671.5	34.0	92.67	255.9	0.0024	0.1088	0.0027
bc	21	0.183(0.0)	8.692	4631.0	378.2	99.35	255.8	0.0004	0.0002	0.0005
diag	21	0.084(0.0)	3.999	4954.6	819.3	99.22	255.5	0.0090	0.0031	0.0000
partalloc	63	0.081(0.0)	1.291	4896.5	0.0	98.70	255.9	0.0015	0.0007	0.0020
pwrite	189	0.039(0.0)	0.206	130.2	0.0	0.84	25.7	0.0053	0.0012	0.0000
readctrl	1	0.006(0.0)	5.921	127.7	0.0	1.34	40.4	0.0009	0.0002	0.0000
pse3d	1	0.002(0.0)	2.027	26.8	0.3	11.53	120.3	0.0011	0.0004	0.0000
aniso_coeff	1	0.002(0.0)	1.664	2641.6	0.0	99.35	255.9	0.0000	0.0000	0.0000
mktable	1	0.002(0.0)	1.659	8415.3	4897.4	99.51	255.8	0.0000	0.0000	0.0000
chkabort	20	0.001(0.0)	0.056	16.7	0.0	0.00	0.0	0.0007	0.0003	0.0000
mkmsh	1	0.001(0.0)	0.831	58.1	0.9	10.41	35.7	0.0003	0.0001	0.0000
finalise	1	0.001(0.0)	0.807	59.3	0.0	0.06	1.1	0.0002	0.0002	0.0000
inputarg	1	0.000(0.0)	0.317	155.4	0.0	0.06	8.0	0.0000	0.0000	0.0000
alloccp	21	0.000(0.0)	0.015	27.2	0.0	0.00	0.0	0.0001	0.0001	0.0000
substop	220	0.000(0.0)	0.001	30.9	0.0	0.00	0.0	0.0000	0.0000	0.0000
substart	219	0.000(0.0)	0.001	40.5	0.0	0.00	0.0	0.0000	0.0000	0.0000
chkbc	1	0.000(0.0)	0.114	29.7	0.1	0.36	3.0	0.0001	0.0000	0.0000
meshalloc	1	0.000(0.0)	0.084	49.9	0.1	6.89	72.5	0.0000	0.0000	0.0000
rmabort	1	0.000(0.0)	0.053	14.1	0.0	0.00	0.0	0.0000	0.0000	0.0000
mkmshsz	1	0.000(0.0)	0.052	30.7	0.3	0.38	1.0	0.0000	0.0000	0.0000
uppercase	17	0.000(0.0)	0.002	63.3	0.0	0.00	0.0	0.0000	0.0000	0.0000
defaults	1	0.000(0.0)	0.028	93.4	0.0	0.46	6.0	0.0000	0.0000	0.0000
const	1	0.000(0.0)	0.012	24.6	4.7	0.00	0.0	0.0000	0.0000	0.0000
mp_time	2	0.000(0.0)	0.004	15.7	0.3	0.00	0.0	0.0000	0.0000	0.0000
defneigh	1	0.000(0.0)	0.005	70.1	0.0	0.00	0.0	0.0000	0.0000	0.0000
updbc	1	0.000(0.0)	0.004	33.7	0.0	0.00	0.0	0.0000	0.0000	0.0000
neighproc	1	0.000(0.0)	0.003	31.1	0.0	0.00	0.0	0.0000	0.0000	0.0000
total	871	783.762(100.0)	899.841	3545.0	1944.5	99.02	135.0	3.3439	2.2542	32.3237

It can be seen that the PSE subroutine takes 97% of all execution time and almost all main memory (as indicated by `BANK CONF`), making it clearly the most performance relevant part of the code. It is called 20 times (according to the 20 time steps), takes around 38 seconds per call and has a sustained performance of 2 GFLOPS, thus 25% of the peak performance. It is interesting to note that the subroutine `mktable` gets 4.9 GFLOPS (61% of peak). This is due to the fact that it vectorizes almost perfectly (99.51% vectorized with an average vector length of 255.8). The task of this subroutine is to build a look-up table for the exponential function to speed-up the numerous evaluations of the PSE kernel function. Also the PSE core routine `pse` vectorizes 99.1%, but the average vector length is only 134.9 while 256.0 would be the maximum. On average, this routine thus uses only 52.7% of the vector registers, which explains the difference in the operation count. It would be possible to increase the vector length of the core routine by building particle interaction lists prior to doing the actual interactions. This way, all interactions of one particle would be combined into a huge vector and one could possibly improve the over-all performance by a factor of two. Also the communication and memory allocation subroutines `updatep`, `bc`, `diag` and `partalloc` vectorized almost completely.

Another interesting detail is to notice that the routines `init` and `do_io` produce a significant amount of instruction cache misses. This is due to the fact that they do I/O on changing data. However they are only called once (`init` at the beginning, `do_io` at the end of the simulation) so it does not really harm.

Chapter 10

Comparison to experiments

The simulations conducted in chapter 9 will be validated against experiments in this chapter. A FRAP experiment as well as a stack of micrograph sections at $\Delta z = 0.1\,\mu$m has been made from the same cell by Anna Mezzacasa according to the following protocol:

Cell culture:
VERO cells were cultured essentially as described in (Pelkmans et. al., Inst. of Biochemistry, ETHZ), using a 100 mm tissue-culture grade plastic dish at 37°C with 5% CO_2 in DMEM cell culture media containing 10% FCS, and then passaged to 18 mm glass cover slips. At 70% confluence, a reporter gene containing the ER targeting signal sequence fused to GFP and the ER retention sequence (KDEL) was transiently transfected using Superfect (Sigma) for expression in mammalian cells. 12 – 16 hours post transfection, live cells were exchanged into CO_2-independent buffer for imaging.

Confocal microscopy:
Live cells were maintained at 37°C during the FRAP experiment using a Zeiss stage warmer. Confocal images were acquired with an inverted Zeiss LSM 510 (X63 PlanApo Nikon objective, N.A. 1.4) using the 488 nm argon laser line and detected using a BP505-530 emission filter (488/543 dichroic) (pinhole setting at 1 Airy unit, image pixel size 80 nm). Images were acquired as 8-bit tiff files (512×512 pixel frame), analyzed using Openlab 3.0.4 and processed in Photoshop 5.0 (Adobe).

***z*-series and Fluorescence Recovery After Photobleaching (FRAP):**
Prior to FRAP, 55 0.1 μm optical z-sections were collected and processed for 3D reconstruction. Experimental details of FRAP have been described in the protocols in [Sbalzarini (2001)]. To a wild type cell expressing the fusion protein: ss-GFP-KDEL, image scans of the entire cell were acquired at low laser power (30% power, 0.3% transmission). A defined region of interest (ROI) ($5\times5\,\mu m^2$) was photobleached at full laser power (100% power, 100% transmission, 50 ms sampling interval); recovery of fluorescence was monitored by scanning the whole cell at low laser power (30% power, 0.3% transmission).

As indicated in the protocol, the cell was transfected to express a protein called *ssGFP-KDEL*. Hereby, "ss" stands for "signal sequence", a polypeptide needed for the ribosomes to recognize the transcript and start synthesizing it into the ER lumen. "GFP" means the familiar green fluorescent protein and "KDEL" is a targeting sequence for ER resident proteins, which causes the protein to be retained in the ER once it is synthesized (or being transferred back to the ER if it escapes). The ssGFP-KDEL protein is soluble meaning that it freely diffuses within the ER lumen (as opposed to membrane-bound proteins). This makes the results of the simulations comparable as the PSE algorithm simulates diffusion *inside* the reconstructed ER surface rather than diffusion *on* that surface. Figure 10.1 shows a sample micrograph of a VERO[1] cell with the ER shining in green. After filtering the data with a Gaussian filter of kernel size 46.576, the reconstruction is done according to chapter 2 with voxels of size $66.54 \times 66.54 \times 27.0$. Figure 10.2 shows a shaded surface view of the resulting triangulated 3D surface.

[1] Cell from green monkey intestine epithelium

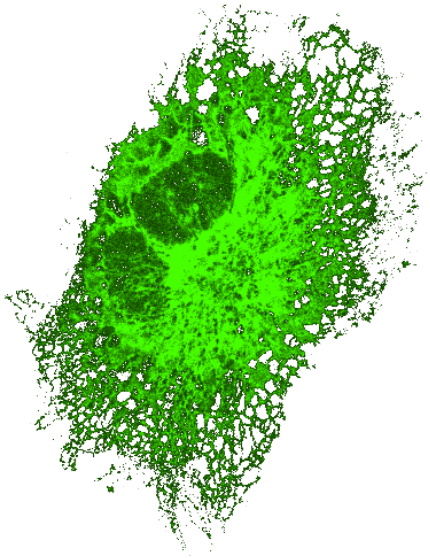


Figure 10.1: Micrograph of the ER sample 8s (courtesy of Anna Mezzacasa)

Figure 10.2: 3D reconstruction of ER surface

The bounding box of the triangulated ER geometry is:

$$[6146.58, 27349.1] \times [4502.35, 31204.9] \times [-40.2303, 40.2303]$$

In the lateral directions x and y, one unit of length corresponds to $1.2\,\mathrm{nm}$[2], in the z direction the unit is $67.1\,\mathrm{nm}^3$. The lateral resolution of the microscope is 220 nm, which means that all the pictures have been taken at a 2.75-fold oversampling. The depth resolution of the microscope is 500 nm. Since the slices are taken at 100 nm distance, they actually overlap, leading to a slightly blurred representation in the third direction. Due to these resolution limitations and the fact that the unit length is two orders of magnitude smaller than for all the runs in section 9.1, the global geometry tolerance TOL is increased from 10^{-10} to 10^{-8}.

To get a meaningful validation, two runs (and experiments) for different bleached boxes are performed using this geometry. Table 10.1 lists the coordinates of the bleached boxes used in the experiments and the subsequent simulations. The first run, termed *8s* features a small bleached box at the periphery of the cell, the second run (*8.2*) a larger one closer to the nucleus.

Run	a	c	b	d
8s	23500	25000	26000	27500
8.2	16500	13000	20500	17000

Table 10.1: Comparison of bleached box geometries

[2]One pixel of the micrograph image represents a $80 \times 80\,\mathrm{nm}$ square, the image has 320×407 pixel and the bounding box of the triangulation is 21202×26703.

[3]The z-distance between each of the 55 micrograph slices is $0.1\,\mu\mathrm{m}$ and the z-length of the bounding box of the geometry is 80.46.

For the subsequent FRAP curve, the fluorescence intensity in the originally bleached box was measured at a sampling interval of 50 ms. The quantification has been done by Anna Mezzacasa after properly normalizing the images and subtracting any background noise. The resulting measured data are listed below and the FRAP curves are shown in figure 10.3.

time	FRAP 8.2	FRAP 8s
0.00	0.00000000	0.000000000
0.05	0.642359994	0.768091498
0.10	0.761357527	0.85031668
0.15	0.825385128	0.875191667
0.20	0.85971692	0.925180215
0.25	0.891158161	0.946877274
0.30	0.914321773	0.946250857
0.35	0.925693525	0.946139788
0.40	0.924324516	0.978810841
0.45	0.937138416	0.96555209
0.50	0.954323345	0.985550568
0.55	0.946796145	0.999120908
0.60	0.937829731	0.98808123
0.65	0.958437983	1.029331826
0.70	0.952619479	1.033946096
0.75	0.962860212	
0.80	0.962289884	
0.85	0.961190977	
0.90	0.983247206	
0.95	0.96625929	
1.00	0.973192383	
1.05	0.976437893	
1.10	0.965600989	

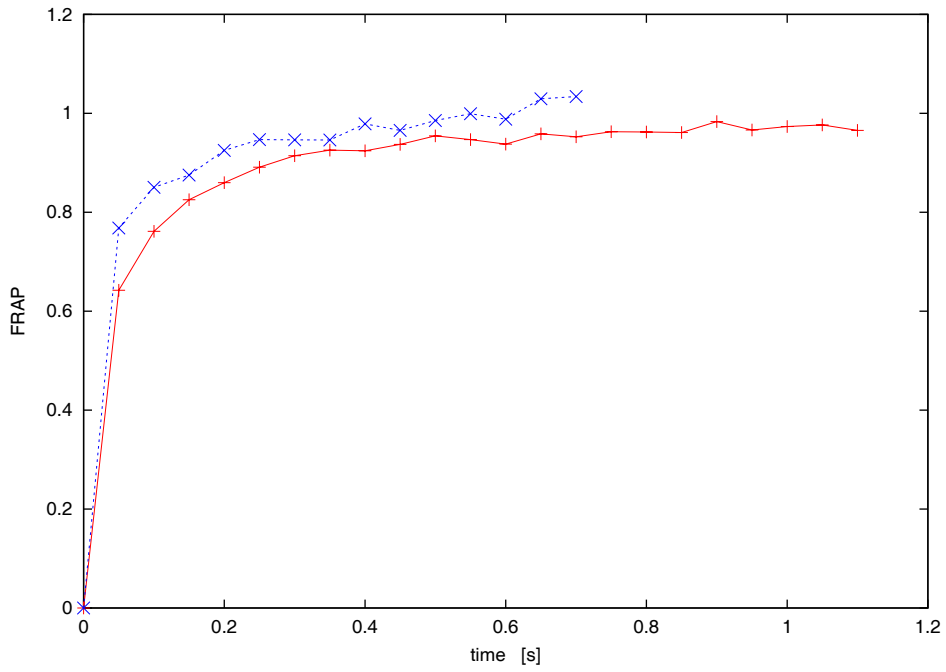


Figure 10.3: Measured FRAP curves for experiments 8s (red) and 8.2 (blue) (courtesy of Anna Mezzacasa)

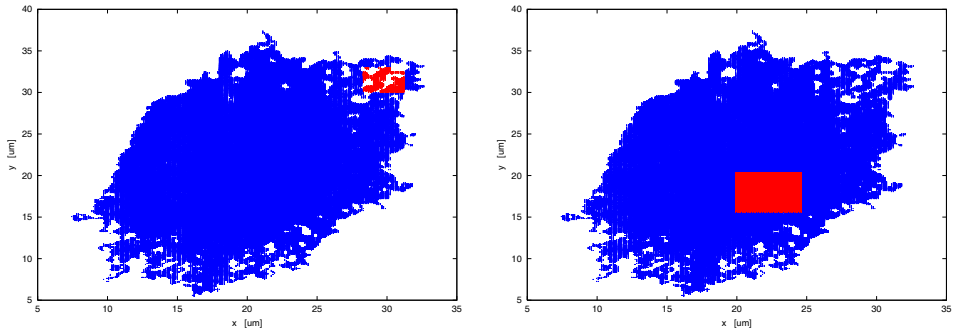
One problem with the protein used for the experiments is its fast diffusion due to its small size of ~ 30 kDa and the free space diffusion. This means that the FRAP curve will recover very quickly, asking for a large diffusion constant in the simulations. According to the stability criterion given in section 9.1 this asks for a very small time step which in turn requires a large number of time steps to be simulated. To keep the simulations computationally feasible, the number of particles therefore is drastically reduced compared to all previous runs. The inter-particle spacings are chosen to be $h_{1,2,3} = [100.0,\ 100.0,\ 20.0]$ which results in a total number of 46486 particles. This is roughly 1/10 of the numbers used so far. According to the timing results given in section 9.3 this means that one time step will complete in about 1.5 s on prometeo. The required number of 450000 time steps will thus approximately need 187.5 hours on 4 processors (amounting to 750 CPU-hours !).

Table 10.2 lists the resulting numbers of particles inside the bleached area (N_B), the number of grid points inside the bleached box (N_g) as well as the asymptotic steady-state FRAP value (c_∞) and the volume-filling coefficient (ϕ_0). Definitions and explanations of all of these properties have been given in section 9.1. It can be seem from the volume-filling coefficient that the second run takes place in an area where the ER is much denser since it is closer the the nucleus.

Sample	N	N_B	N_g	c_∞	ϕ_0
8s	46486	112	2500	0.9976	0.0448
8.2	46486	3556	6400	0.9235	0.5556

Table 10.2: Asymptotic FRAP values and volume-filling coefficients

According to the algorithms presented in section 7.2.5, the particles are initialized and placed inside the reconstructed ER geometry. Figure 10.4 shows the corresponding plots for both examples. As before, blue points signify particles of initial concentration 1, red points such of concentration 0.

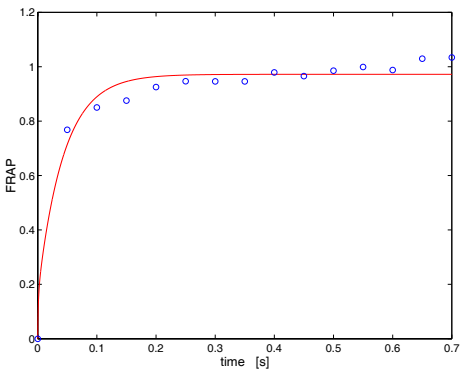


8s 8.2

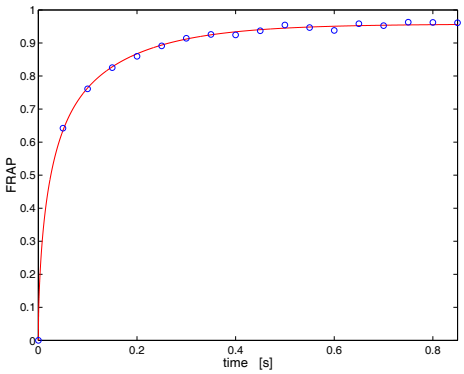
Figure 10.4: Initial particle distributions for comparative simulations. Blue points correspond to particles of concentration 1, red points to those of concentration 0.

The diffusion constant is arbitrarily set to $D = 750.0$ for all runs. This requires the time step size to be less than 0.267 for the simulation to be stable. A time step of $\delta t = 0.1$ is chosen for all runs. The simulations are performed using the techniques introduced in chapter 7 in order to obtain simulated FRAP curves. Using MATLAB, the normalized FRAP curves are then fitted to the experimental data such as to minimize the quadratic error. This is done by allowing time scaling only which is –

as shown in section 9.1 – equivalent to varying the diffusion constant. Figure 10.5 shows the results for both examples along with the resulting time stretching factors s_t. Both factors are about $s_t = 2 \cdot 10^{-5}$, indicating that both examples have been run using the same diffusion constant. This is in fact true and can be considered a consistency check for this validation.



8s: $s_t = 2.0 \cdot 10^{-5}$



8.2: $s_t = 1.9 \cdot 10^{-5}$

Figure 10.5: Simulated FRAP curve (red) compared to experimental measurement data (blue circles). The time stretching factors are listed below the corresponding figure.

The fitting error for sample 8s is 0.021 and thus much larger than the one for the 8.2 sample of $6.389 \cdot 10^{-4}$. This has different reasons, among them:

- Fluorescence recovery in the first example is very fast, resulting in an almost complete recovery after only three measurement intervals. This leads to large measurement uncertainties since the speed of the process under observation challenges the measurement equipment.
- Due to the large inter-particle spacing that had to be chosen in order to finish the simulation runs on time, the resolution is quite poor having only 112 particles inside the bleached box (compared to 3556 for the 8.2 example).
- The simulation is done in an area where the ER is not very dense. This leads to a high sensitivity to geometry errors due to picture noise or reconstruction uncertainties.
- The measured FRAP curve for the first example recovers to values > 1, which is not possible. This indicates a generally higher level of measurement uncertainty for this run.

Taking these points into account, one has to state that the second run (8.2) is more significant and reliable than the first one. The simulation should therefore be validated against the 8.2 run and the 8s run should be taken as a consistency check only. It is noticeable that the resulting time stretching factors are about the same, despite the larger fitting error for 8s. This could mean that the method presented is quite robust against measurement errors and simulation resolution. However, to make a statement about this, more samples have to be considered of course. Another explanation for the deviations in the first case could be that diffusion in reality is anisotropic and thus the isotropic simulation results fail to match it exactly. Running anisotropic simulations with different ratios of anisotropy would be needed to address this question in future work (cf. chapter 13).

Using the time stretch information and the known length scale of the micrographs it is now even possible to calculate a numerical estimation for the diffusion constant. The simulation has been run with a diffusion constant of 750 (units length)^2/(unit time). From the fitting of the resulting curve to the experimental data we know that (unit time)=$2 \cdot 10^{-5}$ s and from the microscope settings (see page 94) we know (unit length)=1.2 nm for the lateral directions. Inserting this yields a diffusion constant of $D = 54.0\,\mu\text{m}^2/\text{s}$. To compare with, the reported diffusion constant of pure GFP in water is $87\,\mu\text{m}^2/\text{s}$.

In conclusion, one can say that the simulations have been shown to perform reasonably well, even under unfavorable conditions. They generally exhibit good coincidence with experimental data and make estimations of the diffusion constant possible.

Chapter 11

Towards a novel FRAP data model

It has been shown in chapter 10 that it is possible to obtain numerical values for the diffusion constant by fitting the FRAP curve from a simulation in the very same geometry to the measured intensity values. However, it would be too time consuming and require too much knowledge and resources to do so on a regular basis in productive laboratory work. Therefore, we are looking for a mathematical model that describes the time behavior of diffusion in the ER (i.e. the FRAP curve as a function of time). The parameters of this model will then be identified using the simulated FRAP curves for which we know the value of the diffusion constant. The model should then be able to interpolate and – if based on first physical principles – also extrapolate to new experimental FRAP curves of unknown diffusion constant in order to extract the value thereof.

11.1 Review of current models

To start with, briefly recall the standard models used today as already presented in [Sbalzarini (2001)]. Note that besides the ones presented hereafter, other models for special cases such as strip bleaching experiments exist ([Ellenberg et al. (1997)]). The first standard model is the *exponential model* emerging from the solution of the diffusion equation on a square 2D plate assuming homogeneous and isotropic diffusion. The fluorescence intensity in an initially bleached square region will then recover as:

$$F(t) = F_\infty \left(1 - e^{-\alpha D t}\right) \tag{11.1}$$

where $F(t)$ is the intensity (FRAP value) at time t, F_∞ is the asymptotic intensity for $t \to \infty$ and α is a model parameter.

Another model commonly used is a purely *empirical correlation* originally proposed by [Dayel, Hom & Verkman (1999)] and [Reits & Neefjes (2001)]. It is defined by:

$$F(t) = F_a + \frac{\left[F_a + R\left(F_0 - F_a\right)\right]\left(t/t_{1/2}\right)^\alpha}{1 + \left(t/t_{1/2}\right)^\alpha} \tag{11.2}$$

where F_0 is the intensity just before bleaching, F_a is the intensity just after bleaching and α and $t_{1/2}$ are model parameters. The *mobile fraction* R is defined as:

$$R = \frac{F_\infty - F_a}{F_0 - F_a}$$

where F_∞ is the asymptotic fluorescence value for $t \to \infty$.

Both models described above lack a solid physical foundation and are thus questionable in their extrapolation capabilities. Due to this fundamental problem, there are a number of practical insufficiencies. The models are for example not capable to account for the fact that different ER geometries will yield different FRAP curves, even at the same value of the diffusion constant (cf. figure 9.3 in section 9.1). They will thus predict a difference in the apparent diffusion constant even though there is none. This is simply due to the fact that they do not contain any information about the geometry at hand (more accurate: about the restriction of diffusion to the geometry at hand) and will interpret all variations in the FRAP curve as caused by the diffusion constant.

Another problem with these models is that they do not include any information about the position or the size of the bleached box. As shown in section 9.2 this is however an important piece of information that indeed does have an influence on the resulting FRAP curve (see figure 9.7). Current models will thus predict a dependence of the diffusion constant on the geometry of the bleached box. This however is completely unphysical and an artifact of the model used. The diffusion constant is physically given by nature and it cannot be that it depends itself on the method used to measure it. New models should thus take this into account.

11.2 New models

Now knowing the currently used models and their insufficiencies, two new models as developed during this project are presented.

11.2.1 A power law model

In order to have a model that takes the ER's local geometry into account, the concepts of diffusion on fractal sets are used. It has been shown in section 6.2 that the dimension of the walk is a sufficient parameter to capture the influences of geometry on the time behavior of diffusion. It has also been shown in the same section that the mean square displacement of a particle undergoing Brownian motion on a fractal is given by:

$$\mathsf{E}\left(\left|X\left(t+\delta t\right)-X\left(t\right)\right|^2\right) \asymp \delta t^{2/d_w}$$

where d_w is the dimension of the walk for the specific geometry under consideration. Using this parameter in a model would thus incorporate the sought-after geometrical information and moreover it does this very efficiently using just a single parameter. For unrestricted diffusion in a Euclidean shape, d_w is always equal to 2. For fractals (or geometrically restricted diffusion), we have $d_w \neq 2$. For the first model, we will thus assume that the FRAP value is proportional to t^α for some parameter α. In order to meet the global model constraints that the FRAP curve starts at $(0, 0)$ and asymptotically goes to a steady-state value F_∞, we set:

$$F(t) = F_\infty\left(1 - t^{-\alpha}\right)$$

The problem now is that this would be $-\infty$ for $t = 0$, so we shift the origin by one, replacing t with $(t+1)$. To account for the fact that the mean square displacement is only proportional to t^{2/d_w} but not equal, we also introduce a proportionality factor β. The final *power law model* is thus given by:

$$F(t) = F_\infty\left(1 - (\beta t + 1)^{-\alpha}\right) \tag{11.3}$$

with α and β being its parameters. Compared to the current models of section 11.1, this now incorporates some physical knowledge about the process and also takes into account the influences of the local ER geometry on diffusion. However, it still fails to include any information about the geometry of the bleached box and will thus still be insufficient.

11.2.2 A second order physical model

The task is to now include some information about the bleached box into the model. This is done by starting from a very simple and basic physical reasoning. Consider the situation in the vicinity of the bleached box as depicted in figure 11.1. Instead of looking at diffusion of fluorescent protein *into* the box, consider the complementary problem of diffusion of bleached protein *out of* the box. This process is analogous to the original one and also happens in reality, governed by the same laws and the same diffusion coefficient. Considering it has however the advantage that the asymptotics of the model will automatically come out right whereas a model for diffusion *into* the box would need a fix (just as the previous power law model needed) to prevent it from diverging to infinity.

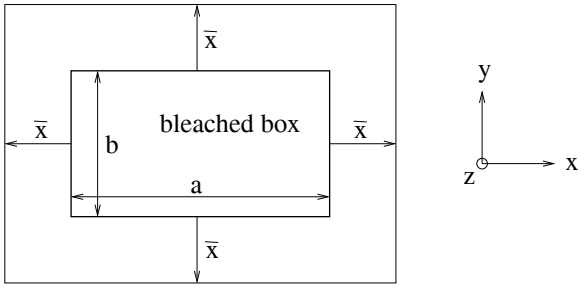


Figure 11.1: Geometrical situation around the bleached box

Let a, b and c be the lengths of the edges of the bleached box in all three dimensions. Without loss of generality, assume that the initial concentration of bleached protein inside the box is 1. The total mass of bleached protein in the box is thus equal to abc. After some time elapsed, the concentration front will have traveled a mean distance of $\overline{x}$ in both the x and y direction due to diffusion. The new volume in which the bleached proteins can distribute themselves is thus given by $(a + 2\overline{x})(b + 2\overline{x})c$. Assuming a homogeneous distribution and using the conservation of mass, the new mean concentration in the bleached box hence becomes:

$$\frac{abc}{(a + 2\overline{x})(b + 2\overline{x})c}$$

The second assumption to be made is that there is a finite and constant number of places for particles to be. This means that if a bleached protein moves out of the bleached box, a fluorescent one will come in for it in exchange. Under this assumption, the fluorescence intensity (or the FRAP value) is given by the complement of above concentration. At the same time we also rescale the model to an asymptotic level of F_∞ instead of 1, so it becomes:

$$F(\overline{x}) = F_\infty \left(1 - \frac{ab}{(a + 2\overline{x})(b + 2\overline{x})}\right)$$

To also take into account the influences of the local ER geometry, we set $\overline{x}^2 = \alpha^2 t^{2\beta}$ using the same concepts and proofs of diffusion on fractal sets as for the power law

model. Inserting this into above equation leaves us with the final formulation of the *second order physical model*:

$$F(t) = F_\infty \left(1 - \frac{ab}{ab + 2\,(a+b)\,\alpha t^\beta + 4\alpha^2 t^{2\beta}}\right) \tag{11.4}$$

with parameters α and β. This model now includes both information about the ER geometry (using the fractal concepts) and the bleached box (in a and b). Moreover, it is based on first physical principles and can thus be expected to even have some extrapolation capabilities. It is termed *second order* physical model since it contains a second order polynomial in αt in the denominator.

To get the half-time of recovery from this model, one simply sets $F(t) = \frac{1}{2}F_\infty$ and solves for t taking into account that $t > 0$. This yields:

$$t_{1/2} = \left[\frac{\sqrt{(a+b)^2\alpha^2 + 4ab\alpha^2} - (a+b)\alpha}{4\alpha^2}\right]^{1/\beta} \tag{11.5}$$

This can now be used to compare the results to the ones of the empirical model given by equation 11.2 or to draw certain physical conclusions (cf. also section 11.4).

11.3 Identification of the model parameters

In order for the models described in the previous sections to be of any use, their parameters have to be identified (i.e. assigned some numerical value). This is done by fitting them to the simulated FRAP curves obtained in section 9.1.

11.3.1 The gradient descent algorithm

To fit the (nonlinear) models to simulation or experimental data, a gradient descent algorithm is used. Let $F^*(t_n)$ be the given data to match at time $t_n = (n-1)\delta t$ and $F(t = t_n, \alpha, \beta)$ the corresponding value of the model under consideration. Now define the matching error as:

$$E = \sum_{n=1}^{n_{max}} E(t_n) = \frac{1}{2}\sum_{n=1}^{n_{max}} \left[F^*\left(t_n\right) - F\left(t = t_n, \alpha, \beta\right)\right]^2 \tag{11.6}$$

The problem now consists of finding the pair of model parameters (α, β) that minimizes the error E. This is done by following the gradient of E with respect to α and β:

$$\frac{\partial E}{\partial \alpha} = \sum_{n=1}^{n_{max}} \frac{\partial E(t_n)}{\partial \alpha}$$
$$\frac{\partial E}{\partial \beta} = \sum_{n=1}^{n_{max}} \frac{\partial E(t_n)}{\partial \beta}$$

The algorithm thus proceeds as follows:

Algorithm 11.1 (Gradient descent).

Step 1: Choose starting values α_0 and β_0 35-36

Step 2: Iterate for $k = 1, 2, \ldots$

2.1: Update α and β according to:

$$\alpha_k = \alpha_{k-1} - \eta_\alpha \sum_{n=1}^{n_{max}} \frac{\partial E(t_n)}{\partial \alpha_{k-1}}$$
$$\beta_k = \beta_{k-1} - \eta_\beta \sum_{n=1}^{n_{max}} \frac{\partial E(t_n)}{\partial \beta_{k-1}}$$

112-139

2.2: If $\frac{\partial E}{\partial \alpha_{k-1}} < TOL$ and $\frac{\partial E}{\partial \beta_{k-1}} < TOL$: exit 149-151

The *learning rates* η_α and η_β are chosen to maximize convergence but prevent instability. The larger they are, the faster the algorithm will converge but at some point is starts to become unstable or to miss the minimum because it takes too large steps. Above algorithm is implemented as a Fortran program called `nlfit.f90`. See appendix C.4.2.1 for its source code.

11.3.2 Model gradients

To be able to apply the gradient descent algorithm, the gradients of the error function for all models are needed for all time steps. For the derivation of some of the gradients, the following law of logarithmic differentiation has been used:

$$\frac{\partial}{\partial \beta}\left(t^{c\beta}\right) = \frac{\partial}{\partial \beta}\left(e^{c\beta \log t}\right) = c \log t e^{c\beta \log t} = c t^{c\beta} \log t$$

All the model equations including their gradients will in turn be given hereafter.

Exponential model

$$F(t) = F_\infty \left(1 - e^{-\alpha D t}\right)$$
$$\frac{\partial E(t_n)}{\partial \alpha} = \frac{\partial E(t_n)}{\partial F(t)} \frac{\partial F(t)}{\partial \alpha} = \left(F(t_n) - F^*(t_n)\right)\left(F_\infty - F(t_n)\right) t_n$$

Empirical model

To represent the situation of the simulations, the empirical model has to be taken with $F_a = 0$ and $F_0 = 1$. Therefore, we have $R = F_\infty$ and the model becomes:

$$F(t) = \frac{F_\infty \left(t/t_{1/2}\right)^\alpha}{1 + \left(t/t_{1/2}\right)^\alpha}$$

$$\begin{aligned}\frac{\partial E(t_n)}{\partial \alpha} = F_\infty \left(F(t_n) - F^*(t_n)\right) \Big[&\left(t_n/t_{1/2}\right)^\alpha \log\left(t_n/t_{1/2}\right) \left[1 + \left(t_n/t_{1/2}\right)^\alpha\right]^{-1} \\ &- \left(t_n/t_{1/2}\right)^{2\alpha} \left[1 + \left(t_n/t_{1/2}\right)^\alpha\right]^{-2} \log\left(t_n/t_{1/2}\right)\Big]\end{aligned}$$

$$\begin{aligned}\frac{\partial E(t_n)}{\partial t_{1/2}} = F_\infty \left(F(t_n) - F^*(t_n)\right) \Big[&\left(t_n/t_{1/2}\right)^\alpha \left[1 + \left(t_n/t_{1/2}\right)^\alpha\right]^{-2} \frac{\alpha}{t_{1/2}^2} \left(t_n/t_{1/2}\right)^{\alpha-1} \\ &- \frac{\alpha}{t_{1/2}^2} \left(t_n/t_{1/2}\right)^{\alpha-1} \left[1 + \left(t_n/t_{1/2}\right)^\alpha\right]^{-1}\Big]\end{aligned}$$

Power law model

$$F(t) = F_\infty \left(1 - (\beta t + 1)^{-\alpha}\right)$$

$$\frac{\partial E(t_n)}{\partial \alpha} = \left(F(t_n) - F^*(t_n)\right) \left(F_\infty - F(t_n)\right) \log\left(\beta t_n + 1\right)$$

$$\frac{\partial E(t_n)}{\partial \beta} = \left(F(t_n) - F^*(t_n)\right) \left(F_\infty - F(t_n)\right) \frac{\alpha t_n}{\beta t_n + 1}$$

Second order physical model

$$F(t) = F_\infty \left(1 - \frac{ab}{ab + 2\left(a+b\right)\alpha t^\beta + 4\alpha^2 t^{2\beta}}\right)$$

$$\begin{aligned}\frac{\partial E(t_n)}{\partial \alpha} = F_\infty ab \left(F(t_n) - F^*(t_n)\right) &\left[ab + 2\left(a+b\right)\alpha t_n^\beta + 4\alpha^2 t_n^{2\beta}\right]^{-2} \\ &\cdot \left[2\left(a+b\right) t_n^\beta + 8\alpha t_n^{2\beta}\right]\end{aligned}$$

$$\begin{aligned}\frac{\partial E(t_n)}{\partial \beta} = F_\infty ab \left(F(t_n) - F^*(t_n)\right) &\left[ab + 2\left(a+b\right)\alpha t_n^\beta + 4\alpha^2 t_n^{2\beta}\right]^{-2} \\ &\cdot \left[2\left(a+b\right)\alpha t_n^\beta \log t_n + 8\alpha^2 t_n^{2\beta} \log t_n\right]\end{aligned}$$

11.3.3 Properties of the error functions

To get an idea of what the error functions for the different model fits looks like, the parameter space is sampled on a regular cartesian grid of 500×500 grid points for the models having two parameters and on 500 linear interval points for the exponential model having just one parameter. At each grid point (α_i, β_j), the RMS
67-99 matching error is calculated as:

$$E_{\mathrm{RMS}} = \sqrt{\frac{1}{n_{max}} \sum_{n=1}^{n_{max}} \left[F^*\left(t_n\right) - F\left(t = t_n, \alpha_i, \beta_j\right)\right]^2} \tag{11.7}$$

and its logarithm is plotted. The ER sample clx has been taken for all visualizations. Figure 11.2 shows the resulting functional shape for the exponential model, figures

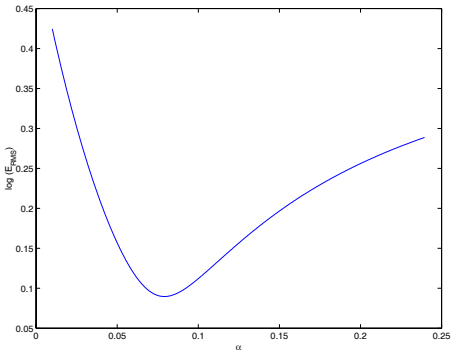


Figure 11.2: Logarithmic RMS matching error for the exponential model

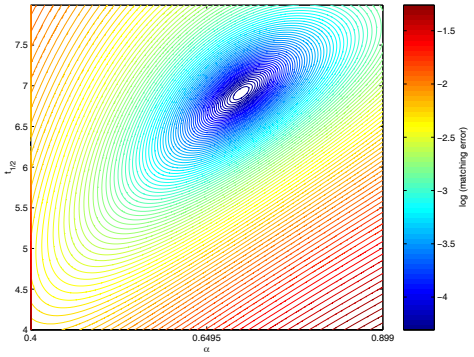


Figure 11.3: Logarithmic RMS matching error for the empirical model

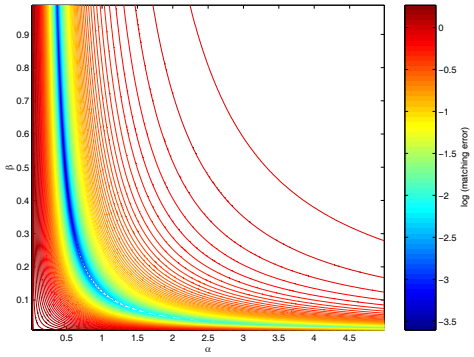


Figure 11.4: Logarithmic RMS matching error for the power law model

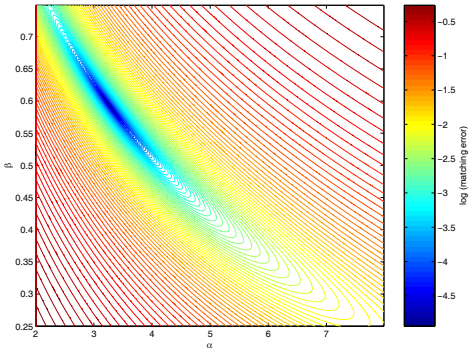


Figure 11.5: Logarithmic RMS matching error for the second order physical model

11.3 to 11.5 show the contour plots for the empirical, power law and second order physical model, respectively.

It can be seen from the plots that within the area of interest there is only one minimum. This justifies the application of a gradient descent algorithm as there is no danger for it to get stuck in local minima.

Figure 11.6 shows the model matching error for the current empirical model (blue dashed line) and the second order physical model (red) as a function of the iteration step number of the gradient descent algorithm. To be able to see anything at all, logarithmic scaling is used on both axes.

Another interesting detail is that the error function for the power law model resembles the Rosenbrock banana function[1] in two characteristic properties:

1. It is very ill-conditioned with a narrow valley that has a soft slope on one side of the minimum and a very steep one on the other side (notice the logarithmic scaling).

2. The valley makes a banana-like turn.

These properties make the function very difficult to handle for optimization algorithms as they might overshoot, miss the minimum or fail to follow the narrow

[1] A popular multimodal test function for minimization algorithms. In its 2-parameter form: $f(x) = 100(\beta - \alpha^2)^2 + (1 - \alpha)^2$

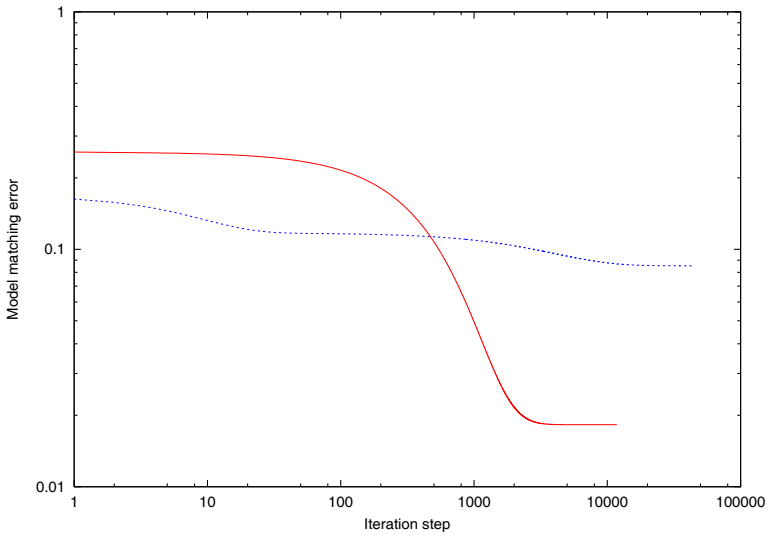


Figure 11.6: Model matching error for empirical model (blue dashed) and second order physical model (red). Notice the double logarithmic scaling.

turning valley. Therefore, Rosenbrock's function is very popular as a test function for optimization techniques. The function shown in figure 11.4 exhibits the same properties. However, it emerges from the real-world problem of fitting fractal diffusion models to FRAP curves whereas the Rosenbrock function is something completely artificial.

Looking at the same function one also notices that gradient descent is one of the best methods to employ due to its robustness. Faster algorithms such as conjugate gradients would not be better as the main problem is the non-linearity of the function and not its dimensionality (CG is optimal for linear problems of high dimension). Figure 11.3 however shows a different picture. This function is very flat and regular with elongated ellipses as contour lines, such that a gradient descent will have a very poor convergence rate and need a lot of iterations. Looking at the results in appendix B in fact shows this.

11.3.4 Fitting results for all ER samples

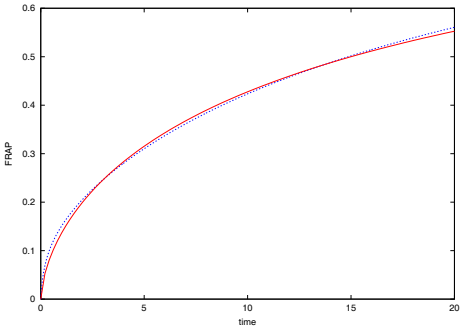
Using all the knowledge about the models and the functional shapes, the second order physical model is fitted to all experimental FRAP curves obtained in chapter 9. The power law model will not be pursued any further as it fails to account for the influences of the bleached box geometry. All fitting runs are started from the same initial point in parameter space, namely:

$$\alpha_0 = 4.0 \qquad\qquad \beta_0 = 0.5$$

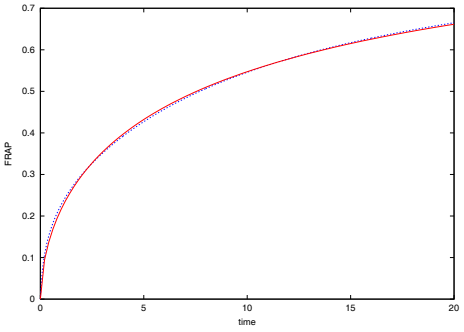
in order to be able to extract comparative information from the iteration counts. The learning rates are fixed to $\eta_\alpha = \eta_\beta = 10^{-3}$ for all runs and the tolerance for the termination criterion is set to $TOL = 10^{-6}$, meaning that both derivatives of the error function have to be less than 10^{-6} for the solution to be considered converged. The simulated FRAP curves all consist of $n_{max} = 2001$ data points for $t \in [0 \ldots 20]$.

Figure 11.7 shows the results. The simulated FRAP curves are shown in dashed blue, the best model fit in red and the minimum value of the RMS fitting error as

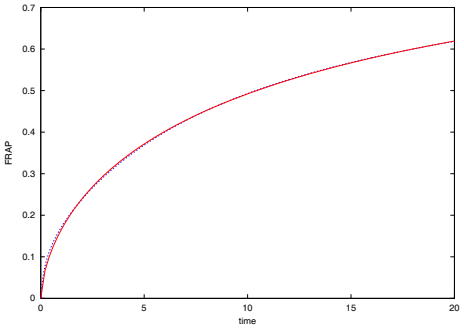
defined by equation 11.7 is included in the captions. The numerical values for the best model parameters as well as the number of iterations it took for the solution to converge are listed in appendix B.



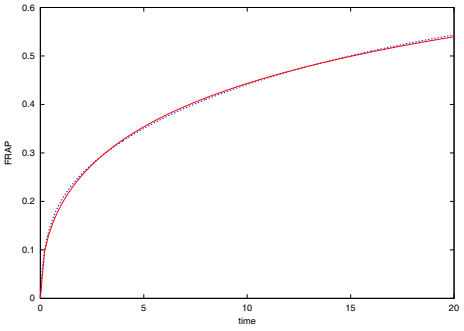
bip2: $E_{\mathrm{RMS}} = 4.259 \cdot 10^{-3}$



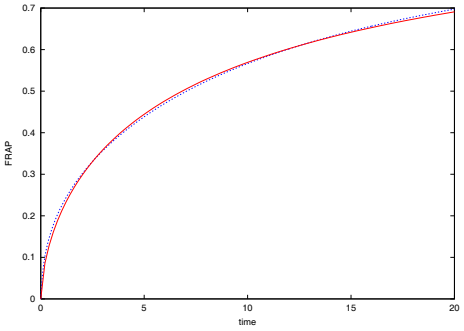
clx: $E_{\mathrm{RMS}} = 3.021 \cdot 10^{-3}$



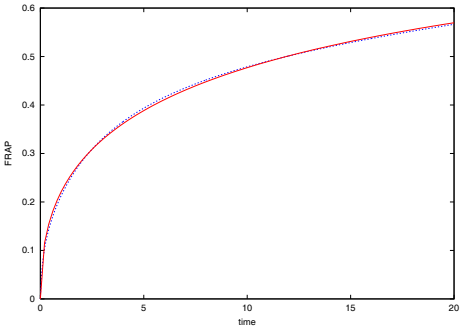
erp57: $E_{\mathrm{RMS}} = 1.818 \cdot 10^{-3}$



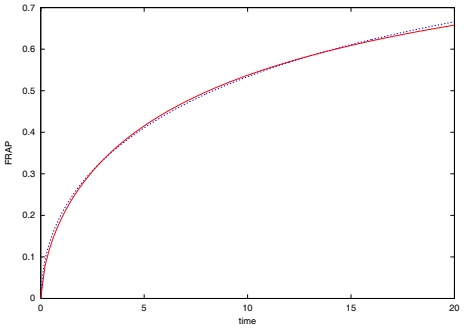
erp572: $E_{\mathrm{RMS}} = 2.374 \cdot 10^{-3}$



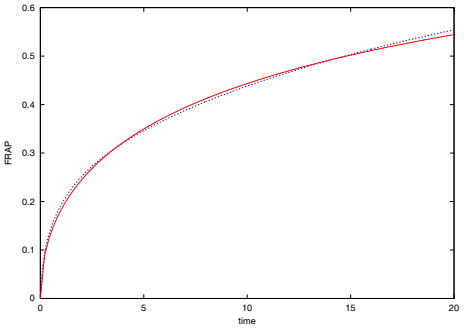
erp573_1: $E_{\mathrm{RMS}} = 3.917 \cdot 10^{-3}$



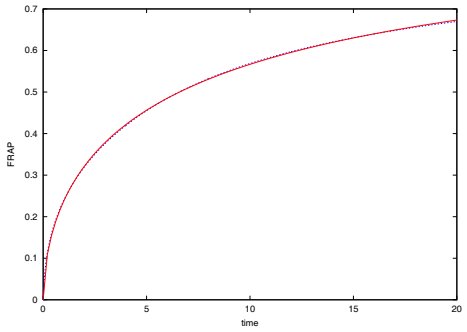
erp573_2: $E_{\mathrm{RMS}} = 2.886 \cdot 10^{-3}$



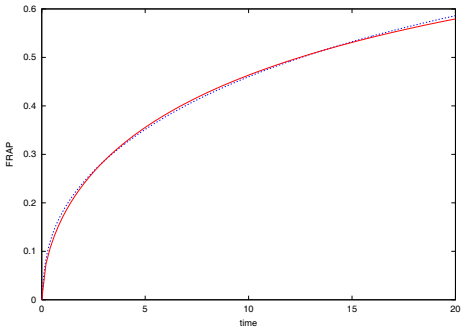
erp573_3: $E_{\mathrm{RMS}} = 2.587 \cdot 10^{-3}$



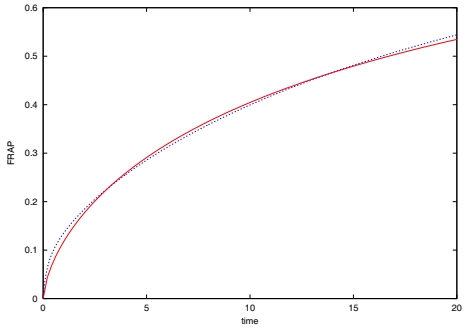
erp574_1: $E_{\mathrm{RMS}} = 3.734 \cdot 10^{-3}$



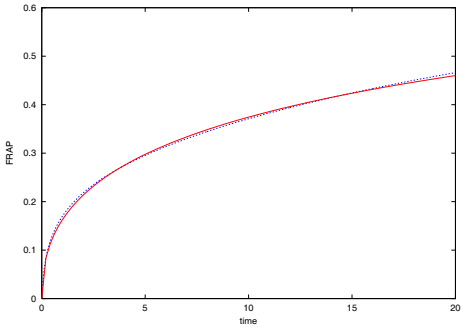
erp574_2: $E_{\mathrm{RMS}} = 1.236 \cdot 10^{-3}$



erp5722: $E_{\mathrm{RMS}} = 3.255 \cdot 10^{-3}$



erp5723: $E_{\mathrm{RMS}} = 5.009 \cdot 10^{-3}$



erp5724: $E_{\mathrm{RMS}} = 2.493 \cdot 10^{-3}$

Figure 11.7: Second order physical model fit (red) to simulated FRAP curves (dashed blue)

The plots show that the new model almost perfectly fits all of the ER samples and even the runs erp5722, erp5723 and erp5724 with different bleached box geometries are well captured. Recall from table 8.2 in section 8.4 that the RMS error of the PSE simulations as compared to the analytic solution on the box test case is at the order of $5.7 \cdot 10^{-3}$ for the resulting FRAP curve. Since the fitting errors for the new model are at the same order of magnitude (in fact always slightly below), they are not significant. It is in principle even possible that the model perfectly describes the physics and that above RMS errors stem from the PSE simulations only. It could however also be possible that some contribution to the error comes from the PSE and another one from the model. In the worst case they are both grossly wrong but their error contributions cancel so to make the model fits look good. However, since the PSE is well validated, it seems safe to assume that its error is small and thus the model not only fits the simulation results but also the physical reality pretty well.

Chapter 12 will be concerned with comparative studies of all the old and new models described in this chapter for the box test case, for a sample ER structure and for experimental data.

11.4 The link between model parameters and the diffusion constant

Now having a new model that fits the simulated FRAP data, the question arises of how its parameters α and β reflect the diffusion constant and the dimension of the walk. Only this link would finally allow to draw information about real-world quantities from the data model fits. We are thus looking for the following functions for the second order physical model:

$$D = f_1(\alpha, \beta)$$
$$d_w = f_2(\alpha, \beta)$$

The first function is the one we are actually interested in so as to be able to use the new model for quantitative evaluations of real FRAP experiments in the laboratory. The second one is just a "side effect" but nonetheless interesting. As mentioned in section 6.3, there exists no algorithm to measure the spectral dimension or the dimension of the walk for an arbitrary given geometry. Finding f_2 however would enable us to measure both the dimension of the walk and – since we can estimate Hausdorff's dimension using the box counting algorithm of section 5.2 – the spectral dimension by simulating a FRAP experiment inside the geometry in question and fitting the new model to the result.

The following constraints for the functions f_1 and f_2 are known from various computer experiments:

- an increase in the diffusion constant leads to an increase in α and vice versa: $D \uparrow \iff \alpha \uparrow$
- an increase in the dimension of the walk leads to a decrease in β and vice versa: $d_w \uparrow \iff \beta \downarrow$
- β does not depend on D but only on the geometry d_w since two runs in the same geometry with different diffusion constants yield the same values for β: $d_w = f_2(\beta)$
- α depends on both D and the geometry: $D = f_1(\alpha, \beta)$
- α depends quadratically on D: $\alpha_1^2/\alpha_2^2 = D_1/D_2$

Furthermore, it turned out that f_1 is not just a simple polynomial in α and β, nor is it a polynomial in αt^β or any real rational function of such polynomials.

For the second order physical model to be dimensionally consistent, it has to yield something that has the same measurement unit as intensity. Looking at equation 11.4 one notices that this means that the term in parentheses has to be dimensionless. The number 1 trivially meets this requirement. The numerator of the following fraction is of dimension (length)^2, so the denominator must be of the same dimension. This requires α to be of dimension (length)$\times$(time)$^{-\beta}$. β has to be dimensionless as it is only found as an exponent to time. The simplest functions that fulfill these requirements and meet all constraints stated above are:

$$D = f_1(\alpha, \beta) = \frac{\left(\alpha T^\beta\right)^2}{\tau}$$
$$d_w = f_2(\beta) = \frac{\kappa}{\beta}$$

for some constants $\tau > 0$ [time], $T > 0$ [time] and $\kappa > 0$ [dimensionless]. Unfortunately, this set-up fails to recover the correct diffusion constants for the simulated FRAP curves.

Due to time resource limitations it has not been possible to find the correct functions f_1 and f_2 during this work. However, enough simulations in different ER geometries have been conducted and it is possible to produce a large data set of (α, β) and corresponding D values by proper time scaling of those simulated FRAP curves. As mentioned in section 9.1 varying the diffusion constant simply corresponds to rescaling the time and no new PSE simulation is necessary. This way, a few hundred FRAP curves with different but known D values could be created for the various ER samples considered. The model can then be fitted to each of these curves to get the corresponding parameter values. This would finally yield enough data to use techniques like neural networks, data assimilation or machine learning methods to find the sought-after functional relationships.

Meanwhile, the new model can already be used to get recovery half-times according to equation 11.5. Statements about relative differences and ratios of diffusion constants *in the same geometry* are possible directly from the half-times as they are linearly related to each other. However, this does not work across different geometries since the geometry also has an influence on the functional relationship between the recovery half-time and the diffusion constant. This becomes obvious when calculating the half-times for all the model fits of section 11.3.4 giving $11.78^{+12.40}_{-6.08}$. However since all of them are simulation runs with the same diffusion constant, they should be equal if geometry had no influence. Existing models such as the one by [Dayel, Hom & Verkman (1999)] however completely neglect this fact using the same calibration measurement for all experiments[2]. This still allows the diffusion constants to be compared among each other for the same cell (since only ratios are considered) but it makes them completely incomparable among different cells that have differently shaped ERs or even among different groups that use different calibration points.

[2]One such calibration point is that a recovery half-time of 100 ms corresponds to a diffusion constant of $5 - 10\,\mu\text{m}^2/\text{s}$

Chapter 12

Comparison to currently used models

The new models developed in section 11.2 will be compared to the current models as stated in section 11.1. All four models are to be compared to each other both for the box test case as defined in section 8.1 and a simulated ER sample from section 9.1. In addition, the empirical and the second order physical model are compared to each other on the experimental data of chapter 10. All model fits are made using the gradient descent algorithm as descibed in section 11.3.1.

12.1 Comparison on the box test case

Table 12.1 gives the initial parameters, learning rates and tolerance settings used for fitting the various models to the $n_{max} = 601$ data points of the analytical solution of the box test case. Table 12.2 summarizes the optimal model parameters found, the minimum value of the quadratic fitting error as defined by equation 11.6 and the number of iterations it took for the gradient descent algorithm to converge below the prescribed tolerance.

Model	Initial parameters	Learning rates	Tolerance
Exponential	$\alpha_0 = 0.4$	$\eta_\alpha = 10^{-5}$	10^{-6}
Empirical	$\alpha_0 = 0.4$, $t_{1/2,0} = 2.0$	$\eta_\alpha = \eta_{t_{1/2}} = 10^{-3}$	10^{-6}
Power law	$\alpha_0 = 0.5$, $\beta_0 = 0.9$	$\eta_\alpha = \eta_\beta = 10^{-3}$	10^{-6}
2^{nd} order physical	$\alpha_0 = 2.0$, $\beta_0 = 0.5$	$\eta_\alpha = \eta_\beta = 10^{-3}$	10^{-6}

Table 12.1: Parameter settings for box test case model fits

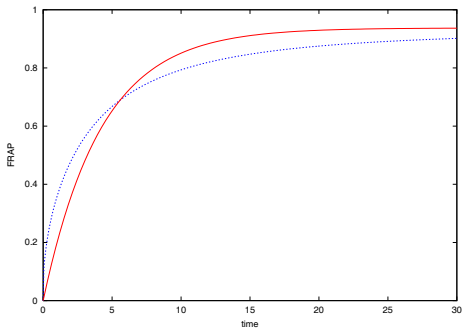
Model	Optimal parameters	E_{min}	N_{Iter}
Exponential	$\alpha_{opt} = 0.23811$	1.225543	8422
Empirical	$\alpha_{opt} = 0.94695$, $t_{1/2,opt} = 1.58038$	0.140867	14158
Power law	$\alpha_{opt} = 1.11405$, $\beta_{opt} = 0.43515$	0.085755	15115
2^{nd} order physical	$\alpha_{opt} = 0.12707$, $\beta_{opt} = 0.78848$	0.030658	1704

Table 12.2: Optimal parameters for box test case model fits

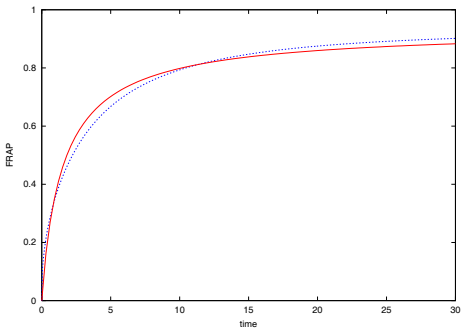
Figure 12.1 shows the resulting plots of the fitted models (red curves) to the analytical solution for the box test case (blue curves). The corresponding minimum value

of the quadratic fitting error is listed in each figure's caption. It can be seen that even for the simple (non-fractal, homogeneous and isotropic) test case, the current models seem to be insufficient. For the exponential model this is due to the fact that it is based on 2D plate diffusion whereas the box test case is a 3D problem. The effects of dimensionality as investigated in [Sbalzarini (2001)] are thus reflected in the resulting fitting error. In fact (after proper temporal scaling), the exponential model's plot below looks very similar to figure 5.2 (page 29) in [Sbalzarini (2001)] which compares a 2D and a 3D random walk simulation. This is evidence that the influence of dimensionality largely accounts for the error in the exponential model fit. The empirical model on the other hand is completely based on measurement data for the ER and lacks physical background. It is therefore not surprising that its performance on the box test case (which is different from ER diffusion) is not convincing as it fails to extrapolate to new geometries.

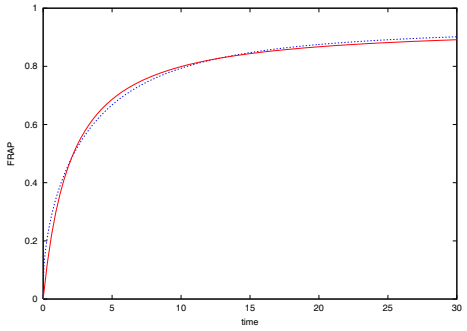
Both new models perform better than the current ones since they are based on physical principles. In fact the RMS fitting error for the second order physical model is again at the order of 10^{-3} as for all the ER samples considered in section 11.3.4. This indicates that it manages to extrapolate over a wide range of geometries without grossly deteriorating in performance. It fits about one order of magnitude better than the empirical model and two orders of magnitude better than the exponential model. For the box test case, also the power law model seems quite usable. However, we will see later that it gets slightly worse for the ER samples.



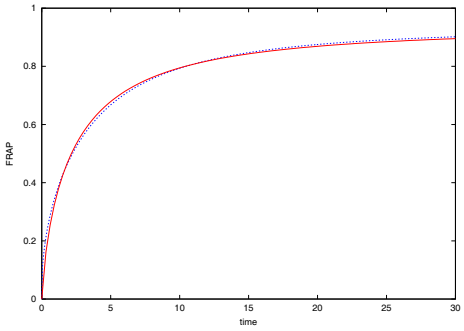
Exponential model: $E_{min} = 1.226$



Empirical model: $E_{min} = 0.141$



Power law model: $E_{min} = 0.086$



Second order physical model: $E_{min} = 0.031$

Figure 12.1: Best fit of different models (red) to the analytic solution for the box test case (blue dashed)

12.2 Comparison on ER samples

Although many of the models' features can already be observed on the box test case, their performance on real simulated FRAP data from ER samples is what actually counts. Therefore, the models are also fitted to the simulation results obtained in section 9.1. Exemplarily, the results for the erp5724 run are given hereafter since this run turned out to pose the hardest problem to the models because it has a different bleached box size than all the other ones (cf. section 9.2). Table 12.3 lists the initial parameters, learning rates and tolerance settings used for fitting the various models to the $n_{max} = 2001$ data points of the simulated FRAP curve. Table 12.4 summarizes the optimal model parameters found, the minimum value of the quadratic fitting error as defined by equation 11.6 and the number of iterations it took for the gradient descent algorithm to converge below the prescribed tolerance.

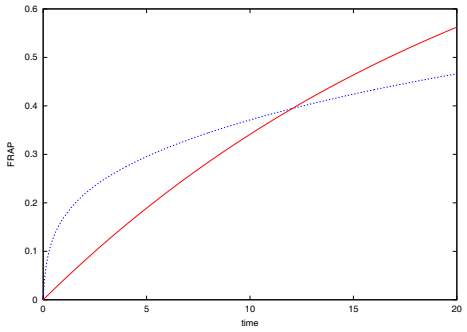
Model	Initial parameters	Learning rates	Tolerance
Exponential	$\alpha_0 = 0.1$	$\eta_\alpha = 10^{-5}$	10^{-6}
Empirical	$\alpha_0 = 0.7$, $t_{1/2,0} = 6.5$	$\eta_\alpha = \eta_{t_{1/2}} = 10^{-3}$	10^{-6}
Power law	$\alpha_0 = 0.5$, $\beta_0 = 1.0$	$\eta_\alpha = \eta_\beta = 10^{-4}$	10^{-6}
2^{nd} order physical	$\alpha_0 = 4.0$, $\beta_0 = 0.5$	$\eta_\alpha = \eta_\beta = 10^{-3}$	10^{-6}

Table 12.3: Parameter settings for erp5724 model fits

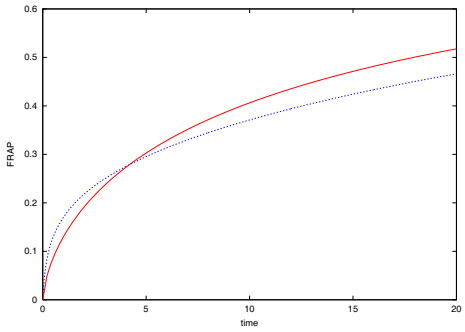
Model	Optimal parameters	E_{min}	N_{Iter}
Exponential	$\alpha_{opt} = 0.04342$	6.433889	24
Empirical	$\alpha_{opt} = 0.66775$, $t_{1/2,opt} = 16.28989$	1.36757	991209
Power law	$\alpha_{opt} = 0.18428$, $\beta_{opt} = 1.38605$	0.166035	344597
2^{nd} order physical	$\alpha_{opt} = 3.61430$, $\beta_{opt} = 0.45771$	0.012437	18318

Table 12.4: Optimal parameters for erp5724 model fits

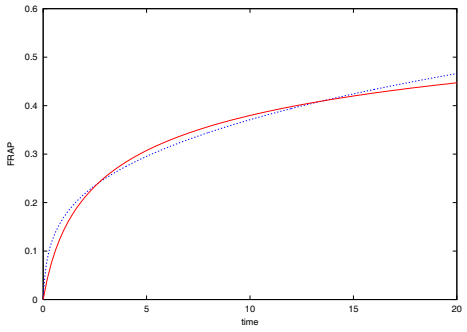
Figure 12.2 again shows the resulting plots of the fitted models (red curves) to the simulated FRAP data (blue curves). The minimum values of the quadratic fitting error are listed along with the figures' captions. The picture is even more striking as it was for the box test case. The exponential model now seems to completely fail to capture the characteristics of the curve. Comparing this picture to the one of the box test case shows that the influence of geometry accounts for an error of 5.208 whereas the influence of dimensionality caused an error of 1.226. The geometrical structures (precise: the restriction of diffusion to them) thus have a large influence on the resulting FRAP curve even at the same value of the diffusion constant. The empirical model is also worse than it was for the box test case. One should however note, that it usually performs better on ER samples (cf. table B.2 in appendix B). This sample however features a bleached box that deviates in size and position from the ones of the other samples. The large error of the empirical model for this run indicates that it is incapable of dealing with different bleached box geometries, as already stated in section 11.1. Another interesting detail is the fact that the empirical model takes about 50 times as many iterations to converge as the second order physical one. This is in agreement with the findings of section 11.3.3.



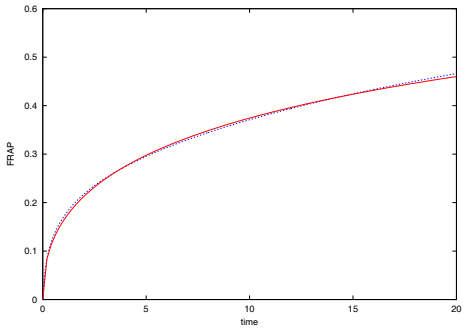
Exponential model: $E_{min} = 6.434$



Empirical model: $E_{min} = 1.368$



Power law model: $E_{min} = 0.166$



Second order physical model: $E_{min} = 0.012$

Figure 12.2: Best fit of different models (red) to simulated erp5724 FRAP curve (blue dashed)

Again, both new models perform significantly better[1] than the empirical model, even for this worst case ER sample. The RMS error of the second order physical model fit is again below the RMS error of the PSE simulations indicating a possibly perfect match.

To allow further comparisons between the best currently used model (i.e. the empirical model) and the best new model (the second order physical model), the fitting results for all ER samples are given in appendix B. Apart from the exception erp573_2, the new model constantly fits one to two orders of magnitude better than the empirical model.

Figure 12.3 shows the model matching errors for the empirical model (blue dashed curve) and the second order physical model (red curve) as a function of the iteration step number k of the gradient descent algorithm 11.1. Both axes use logarithmic scaling to get a clear plot.

[1] one order of magnitude for the power law model, two orders of magnitude for the second order physical model

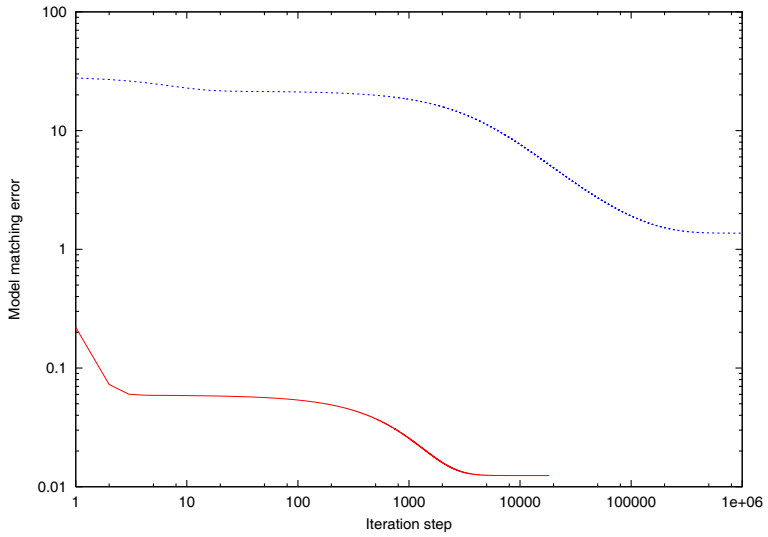


Figure 12.3: Model matching error for sample erp5724 and empirical model (blue dashed) as well as second order physical model (red). Notice the double logarithmic scaling.

12.3 Comparison on experimental data

Finally, the two best models, i.e. the empirical one and the new second order physical model, are compared with real experimental data (courtesy of Anna Mezzacasa). Therefore, the same two experiments as presented in chapter 10 are used. Example 8s has 15 data points and 8.2 has 23 data points. Both experiments have been made with a bleached spot of size $5\,\mu\text{m}\times 5\,\mu\text{m}$. The initial parameters and optimization settings are listed in table 12.5. Table 12.6 gives the optimal model parameters, the minimum value of the quadratic fitting error and the number of iterations needed for the 8s sample and table 12.7 for the 8.2 sample.

Model	Initial parameters	Learning rates	Tolerance
Empirical	$\alpha_0 = 1.0$, $t_{1/2,0} = 0.05$	$\eta_\alpha = 10^{-4}$, $\eta_{t_{1/2}} = 5 \cdot 10^{-5}$	10^{-6}
2^{nd} order physical	$\alpha_0 = 5.0$, $\beta_0 = 0.8$	$\eta_\alpha = 0.1$, $\eta_\beta = 5 \cdot 10^{-4}$	10^{-6}

Table 12.5: Parameter settings for experimental data model fits

Model	Optimal parameters	E_{min}	N_{Iter}
Empirical	$\alpha_{opt} = 11.19487$, $t_{1/2,opt} = 0.01775$	$4.49922 \cdot 10^{-3}$	1118631
2^{nd} order physical	$\alpha_{opt} = 25.93876$, $\beta_{opt} = 0.77531$	$3.90525 \cdot 10^{-3}$	2794396

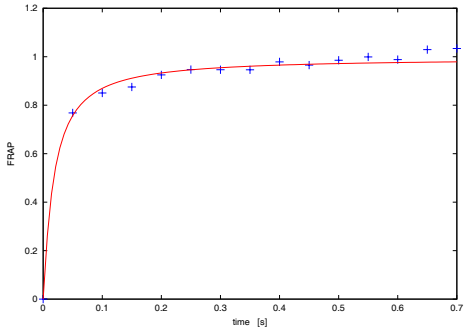
Table 12.6: Optimal parameters for 8s model fits

Figure 12.4 shows the corresponding plots. The measured data points are shown as blue crosses, the optimally fitted model is drawn as a red line. Again, the second order physical model performs better than the empirical one. However, the

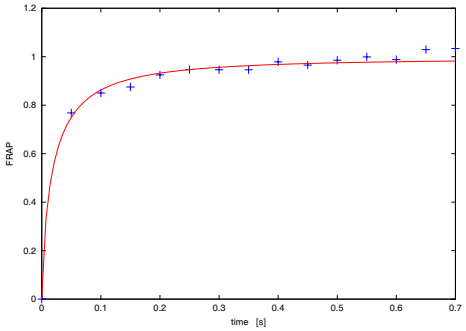
Model	Optimal parameters	E_{min}	N_{Iter}
Empirical	$\alpha_{opt} = 11.02793$, $t_{1/2,opt} = 0.02912$	$1.34563 \cdot 10^{-3}$	523349
2^{nd} order physical	$\alpha_{opt} = 17.49833$, $\beta_{opt} = 0.78028$	$7.44625 \cdot 10^{-4}$	677758

Table 12.7: Optimal parameters for 8.2 model fits

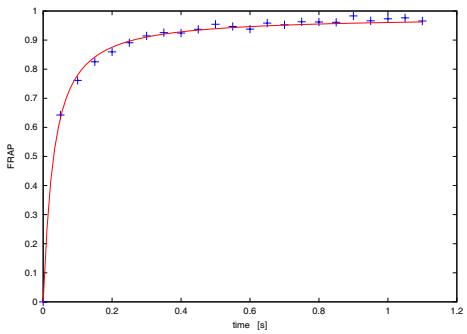
difference is smaller than for the simulated data sets. This was to be expected as the empirical model is based on such experimental data and has been trained to fit it (including its measurement errors and noise).



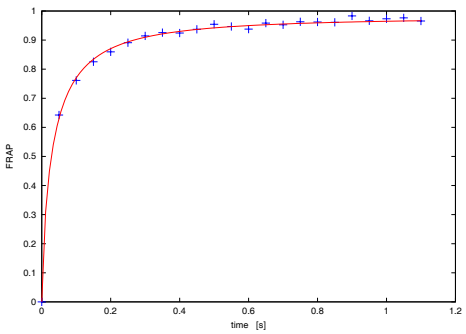
8s with empirical model:
$E_{min} = 4.499 \cdot 10^{-3}$



8s with second order physical model:
$E_{min} = 3.905 \cdot 10^{-3}$



8.2 with empirical model:
$E_{min} = 1.346 \cdot 10^{-3}$

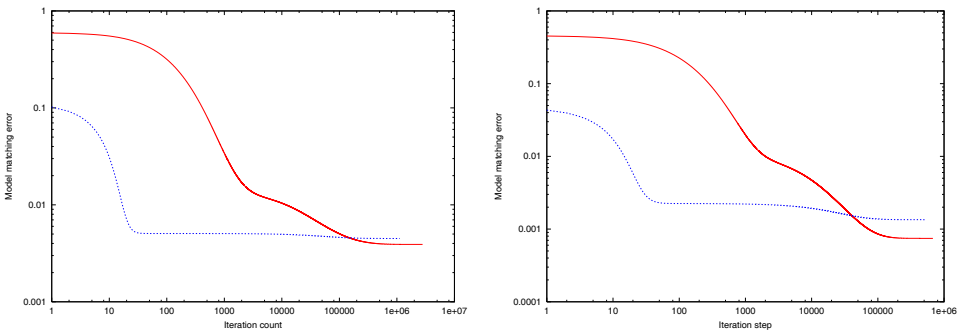


8.2 with second order physical model:
$E_{min} = 7.446 \cdot 10^{-4}$

Figure 12.4: Best fit of different models (red) to experimental FRAP data (blue crosses)

Again, the model matching error is plotted against the iteration step number of the gradient descent algorithm using a double logarithmic scaling. The results for both experimental data sets are shown in figure 12.5 where the blue dashed curve stands for the empirical model and the red curve for the second order physical one.

According to equation 11.5, the recovery half-times for the second order physical model can also be calculated. Table 12.8 lists the results and the relative deviation from the empirical model. The models are in agreement insofar as their mutual difference is less than the statistical scatter (as both experiments have been made using the same ER, the diffusion constant and therefore the half-times for both samples should be equal).



8s 8.2

Figure 12.5: Model matching error for empirical model (blue dashed) as well as second order physical model (red) for both experiments. Notice the double logarithmic scaling.

Sample	$t_{1/2}$ empirical	$t_{1/2}$ physical	relative deviation
8s	17.75 ms	15.70 ms	11.5%
8.2	29.12 ms	26.69 ms	8.3%
average	23.4 ± 5.7 ms	21.2 ± 5.5 ms	9.4%

Table 12.8: Recovery half-times and their deviation

As stated in section 11.4, this would now allow one to deduce the diffusion constant if the influence of geometry was known. Wrongly applying the calibration point of [Dayel, Hom & Verkman (1999)][2] to the geometry of our samples yields $D = 21.8 - 43.5\,\mu\text{m}^2/\text{s}$ for the empirical model and $D = 23.6 - 47.2\,\mu\text{m}^2/\text{s}$ for the second order physical model. Comparing these numbers to the result $D = 54\,\mu\text{m}^2/\text{s}$ obtained from direct numerical simulation in chapter 10 shows that they differ by a factor of 1.2 to 2.5 for the empirical model and 1.1 to 2.3 for the second order physical model. This difference is caused by neglecting geometrical influences. The fact that the deviation for the second order physical model is slightly smaller is due to its better fitting behavior. Using the empirical model, various papers (e.g. [Dayel, Hom & Verkman (1999)]) report diffusion coefficients between $25\,\mu\text{m}^2/\text{s}$ and $30\,\mu\text{m}^2/\text{s}$ for GFP in the cytoplasm, between $5\,\mu\text{m}^2/\text{s}$ and $10\,\mu\text{m}^2/\text{s}$ in the ER lumen and between $20\,\mu\text{m}^2/\text{s}$ and $30\,\mu\text{m}^2/\text{s}$ in the mitochondrial matrix (see [Lippincott-Schwartz, Snapp & Kenworthy (2001)] for a summary). The difference between these numbers and the one obtained with the old empirical model for the present experiments is thus caused by differences in the cells, proteins or experimental set-ups used and has nothing to do with the models involved.

[2] Stating that $t_{1/2} = 100$ ms corresponds to a diffusion constant of $5 - 10\,\mu\text{m}^2/\text{s}$

Chapter 13

Conclusions and future work

This project successfully extended the work of [Sbalzarini (2001)] to study not only the influence of dimensionality but also the one of geometry on diffusion processes in the endoplasmic reticulum. The use of fractal concepts and latest computer simulation techniques made it possible to move on to three-dimensional realistic ER shapes. Changing the simulation technique from random walk to the method of particle strength exchange allowed accurate treatment of diffusion in arbitrary domains. In order to do so, 3D reconstructions of real ER samples were made as described in chapter 2. The triangulation defining their surface was then checked for various validity criteria in chapter 3.

As stated in the future work list of [Sbalzarini (2001)], a sound mathematical treatment of diffusion on fractal sets was needed in order to proceed with using concepts from fractal geometry to find novel FRAP data models for diffusion in the endoplasmic reticulum. Towards this end, chapter 4 introduced and summarized some of the theoretical foundations about measures and dimensions for later use. In this context, different fractal dimensions such as the box counting dimension or Hausdorff's dimension were properly introduced and defined, extending section 10.1 of [Sbalzarini (2001)]. Chapter 6 then made use of this theoretical background to investigate diffusion on fractal sets. Starting from Brownian motion on the Sierpinski gasket, a proof has been presented that Hausdorff's dimension is not a sufficient parameter to capture the time behavior of diffusion on fractals as was suspected in [Sbalzarini (2001)]. Considering transition densities and eigenvalues of the Laplacian, the *dimension of the walk* has been found to actually be the sought-after sufficient geometry parameter. This finding has also been extended to infinitely ramified fractals and anisotropic diffusion.

Chapter 5 was concerned with measuring the fractal dimension of the ER surface as requested in [Sbalzarini (2001)]. Therefore, an algorithm to transform the triangulated representation to a voxel set was presented and validated. A 3D generalization of the box counting algorithm has then been developed and used to determine the box counting dimension of the ER surface to be 2.8. This value is within the theoretically predicted range of [Sbalzarini (2001)].

Chapter 7 introduced the numerical simulation techniques used in this work. An existing code for the anisotropic particle strength exchange method was successfully extended to handle bounded computational domains of arbitrary shapes with either Neumann or Dirichlet boundary conditions. To this end, a fast geometry preprocessor has been developed as well as proper post-processing and visualization techniques. A new set of algorithms is now available that is much more sophisticated and trustworthy than the 2D finite difference tools applied

by [Siggia, Lippincott-Schwartz & Bekiranov (2000)], [Ellenberg et al. (1997)] and [Dayel, Hom & Verkman (1999)].

To validate the simulation algorithms, a test case has been introduced in chapter 8 and its analytic solution derived and analyzed. A simple finite difference code served as a cross-validation. Random walk and PSE simulations have been shown to yield results close to the exact solution, justifying their use for productive simulations. Comparative error considerations and timings have been made to analyze the codes' performance.

Chapter 9 presented diffusion simulations in a number of ER samples in order to obtain simulated FRAP curves. Moreover, the influence of the position and size of the bleached box was investigated and shown to be relevant. The results obtained for the ER geometry differ significantly from the one of the cubic box considered in [Sbalzarini (2001)]. Comparative timings and parallel speed-up estimations concluded this chapter.

In chapter 10, a simulation has been presented that was run in the exact same geometry as used for a real laboratory experiment. Very good coincidence was observed for sufficiently resolved simulations and still good coincidence for under-resolved runs. Moreover, it has been shown that it is possible to determine diffusion constants by fitting simulation data to experimental measurements. The diffusion constant obtained this way differs from the one predicted by the current empirical model of [Dayel, Hom & Verkman (1999)] by a factor of 1.2 to 2.5. Neglect of the geometrical influences hereby accounts for 1.1 to 2.3, the rest is caused by using a model that does not optimally fit the data.

In chapter 11, new FRAP data models have been proposed after briefly reviewing some existing ones. The new models are based on physical principles. Using the theory of diffusion on fractal sets as outlined in chapter 6, the new models also take into account the local ER geometry which has been shown in chapter 9 to significantly influence the FRAP results. One model also includes basic information about the bleached box geometry which has been shown in section 9.2 to affect the FRAP results. After noting some properties of the model matching error functions, the new model's parameters have been determined for all simulated FRAP curves using a gradient descent method. For the new model, almost perfect correspondence was observed for all ER samples with an RMS error which is below the one of the PSE simulation itself. The chapter was concluded by some considerations on how to interpret the models' parameters in terms of real-world physical properties.

The new models have been compared to the current ones in chapter 12 on the box test case, on ER samples and on experimental data. All comparisons show the same picture: the new physical model is better than the current models while fully accounting for geometry and bleached box influences. Comparing the fitting behavior of the 2D exponential model for the box test case and the ER sample allowed to make the observation that the influences of geometrical structures are some 4 to 5 times more important to the time behavior of diffusion (and thus the FRAP curves) than the influences of dimensionality as investigated in [Sbalzarini (2001)].

In the limited time of this diploma thesis, it was of course not possible to address all the questions in desirable detail. Namely the following points will have to be subjects to future work:

- A functional relationship between the parameters of the new model and the diffusion constant has to be found in order to complete the considerations of section 11.4. Due to the required extrapolation capabilities, it is desirable to find a physical relationship rather than using neural networks or machine learning methods.

- Once such a function is known, a small and stable program for automatic quantitative evaluation of experimental FRAP data has to be developed. Such an application would allow robust and coherent determination of numerical diffusion coefficients in daily laboratory use, making results comparable among different groups.
- More accurate (i.e. higher resolution) simulations should be run in order to compare them to experiments and investigate whether the deviations observed in one case in chapter 10 stem from simulation or measurement errors.
- More accurate initial conditions using an initial fluorophore concentration profile as proposed in [Axelrod et al. (1976)] should be included in the simulations and their influence on the resulting FRAP curve should be investigated.
- More and better FRAP data models have to be found and evaluated.
- Finally, diffusion of membrane-bound proteins should be addressed. An extension of the method of particle strength exchange to surface-bound diffusion will however first have to be theoretically derived. This consists of finding a discretization of the Beltrami operator[1] on particles as well as a set of (local) mappings to theoretically describe the ER's surface. The work of [Spekreijse, Hagmeijer & Boerstoel (1996)] could serve as a starting point.

Ultimately, it is desirable to go beyond simulating existing experiments in order to start investigating things not amenable to them and create novel knowledge by using computer simulations only. Examples of such applications include:

- Investigations on whether the medium filling the ER lumen is anisotropic (what ratio of anisotropy ?) or not. This can be done by conducting anisotropic simulations (recall that all simulation codes already include the functionality needed) and fitting their results to experimental data. If those results fit better than the isotropic ones, this could be evidence that diffusion in reality is anisotropic.
- Investigation on whether the ratio and spatial structure of anisotropy do have an influence on the resulting FRAP curves at all (cf. chapter 10). This consists of running different anisotropic simulations and comparing their FRAP curves.
- Use of the new data model to analyze experimental FRAP curves in order to obtain diffusion coefficients that allow biological conclusions about protein folding and sorting.
- Applications of the fractal diffusion theory to completely different fields such as transport phenomena in porous media, complex electric systems or heat conduction.

Altogether, a large step in understanding diffusion processes in the endoplasmic reticulum as well as geometrical influences on diffusion in general has been made when comparing to the state in [Sbalzarini (2001)] and all assigned goals of the project have been achieved (cf. foreword). Nevertheless, some open points remain and there is still work to do until the new models are ready for productive, unattended laboratory use. A sound basis however exists by now and the hope remains that the final goal is not too far any more.

[1] The Beltrami operator describes diffusion which is restricted to a curved surface in space just as the Laplace operator describes free diffusion in space.

Bibliography

[Alberts et al. (1997)] Alberts, B., Bray, D., Johnson, A., Lewis, J., Raff, M., Roberts, K. & Walter, P., 1997, Essential Cell Biology, Garland publishing, Inc., New York.

[Axelrod et al. (1976)] Axelrod, D, Koppel, D. E., Schlessinger, J., Elson, E. & Webb, W. W., 1976, Mobility Measurement by Analysis of Fluorescence Photobleaching Recovery Kinetics, *Biophysical Journal*, **16**, 1055-1069.

[Baldock & Graham (2000)] Baldock, R. & Graham, J. (Eds.), 2000, Image Processing and Analysis - A Practical Approach, Oxford University Press, New-York.

[Barlow & Bass (1992)] Barlow, M. T. & Bass, R. F., 1992, Transition densities for Brownian motion on the Sierpinski carpet, *Probability Theory and related fields*, **91**, Springer, 307-330.

[Barlow, Hattori, Hattori & Watanabe (1997)] Barlow, M. T., Hattori, K., Hattori, T. & Watanabe, H., 1997, Weak Homogenization of Anisotropic Diffusion on Pre-Sierpinski Carpets, *Communications in Mathematical Physics*, **188**, Springer, 1-27.

[Barlow & Perkins (1988)] Barlow, M. T. & Perkins, E. A., 1988, Brownian Motion on the Sierpinski Gasket, *Probability Theory and related fields*, **79**, Springer, 543-623.

[Beaudoin, Huberson & Rivoalen (2001)] Beaudoin, A., Huberson, S. & Rivoalen, E., Simulation of anisotropic diffusion by means of a diffusion velocity method, *submitted to J. Comp. Phys.*.

[Cheezum, Walker & Guilford (2001)] , Cheezum, M. K., Walker, W. F. & Guilford, W. H., 2001, Quantitative Comparison of Algorithms for Tracking Single Fluorescent Particles, *Biophysical Journal*, **81**, 2378-2388.

[Cottet & Koumoutsakos (2000)] Cottet, G.-H. & Koumoutsakos, P. D., 2000, Vortex Methods - Theory and Practice, Cambridge University Press.

[Dayel, Hom & Verkman (1999)] Dayel, M. J., Hom, E. F. Y. & Verkman, A. S., 1999, Diffusion of Green Fluorescent Protein in the Aqueous-Phase Lumen of the Endoplasmic Reticulum, *Biophysical Journal*, **76**, 2843-2851.

[Degond & Mas-Gallic (1989a)] Degond, P. & Mas-Gallic, S., 1989, The weighted particle method for convection-diffusion equations, Part I: The case of isotropic viscosity, *Math. Comp.*, **53**, 485.

[Degond & Mas-Gallic (1989b)] Degond, P. & Mas-Gallic, S., 1989, The weighted particle method for convection-diffusion equations, Part 2: The anisotropic case, *Math. Comp.*, **53**, 509.

[Dietrich et al. (2002)] Dietrich, C., Yang, B., Fujiwara, T., Kusumi, A. & Jacobson, K., 2002, Relationship of Lipid Rafts to Transient Confinement Zones Detected by Single Particle Tracking, *Biophysical Journal*, **82**, 274-284.

[Ellenberg et al. (1997)] Ellenberg, J, Siggia, E. D., Moreira, J. E., Smith, C. L., Presley, J. F., Worman, H. J. & Lippincott-Schwartz, J., 1997, Nuclear Membrane Dynamics and Reassembly in Living Cells: Targeting of an Inner Nuclear Membrane Protein in Interphase and Mitosis, *Journal of Cell Biology*, **Vol. 138, Nr. 6**, 1193-1206.

[Falconer (1985)] Falconer, K. J., 1990, The Geometry of Fractal Sets, Cambridge University Press.

[Falconer (1990)] Falconer, K. J., 1990, Fractal Geometry - Mathematical Foundations and Applications, John Wiley & Sons, Chichester, UK.

[Falconer (1997)] Falconer, K. J., 1997, Techniques in Fractal Geometry, John Wiley & Sons, Chichester, UK.

[Hattori, Hattori & Watanabe (1994)] Hattori, K., Hattori, T. & Watanabe, H., 1994, Asymptotically one-dimensional diffusions on the Sierpinski gasket and the abc-gaskets, *Probability Theory and related fields*, **100**, Springer, 85-116.

[Hausdorff (1919)] Hausdorff, F., 1919, Dimension und äusseres Mass, *Mathematische Annalen*, **79**, 157-179.

[Hockney & Eastwood (1988)] Hockney, R. W. & Eastwood, J. W., 1988, Computer simulation using particles, Institute of Physics Publishing, Bristol/Philadelphia.

[Hou, Gilmore, Mindlin & Solari (1990)] Hou, X.-J., Gilmore, R., Mindlin, G. B. & Solari, H. G., 1990, An efficient algorithm for fast O(N*log N) box counting, *Physics Letters A*, **Vol. 151, No. 1 2**, 43-46.

[Kamm (2002)] Kamm, R. D., 2002, Cellular Fluid Mechanics, *Annual Revisions in Fluid Mechanics*, **34**, 211-232.

[Kusumi, Sako & Yamamoto (1993)] Kusumi, A., Sako, Y. & Yamamoto, M., 1993, Confined Lateral Diffusion of Membrane Receptors as Studied by Single Particle Tracking (Nanovid Microscopy). Effects of Calcium-Induced Differentiation in Cultured Epithelial Cells, *Biophysical Journal*, **65**, 2021-2040.

[Liebovitch & Toth (1989)] Liebovitch, L. S. & Toth, T., 1989, A fast algorithm to determine fractal dimensions by box-counting, *Physics Letters A*, **Vol. 141, No. 8 9**, 386-390.

[Lippincott-Schwartz, Snapp & Kenworthy (2001)] Lippincott-Schwartz, J., Snapp, E. & Kenworthy, A., 2001, Studying Protein Dynamics in living Cells, *Nature Rev. Cell Biology*, **2**, 444.

[Mandelbrot (1982)] Mandelbrot, B. B., 1982, The fractal geometry of nature, W. H. Freeman and Co., San Francisco.

[McCorquodale, Colella & Johansen (2001)] McCorquodale, P., Colella, P. & Johansen, H., 2001, A Cartesian Grid Embedded Boundary Method for the Heat Equation on Irregular Domains, *Journal of Computational Physics*, **173**, 620-635.

[Mitra, Murthy, Kundu & Bhattacharya (2001)] Mitra, S. K., Murthy, C. A., Kundu, M. K. & Bhattacharya, B. B., 2001, Fractal Image Compression Using Iterated Function System with Probabilities, *Proc. Intl. Conference on Information Technology (ITCC'01)*, IEEE Computer Society, 191-195.

[Monks (2001)] Monks, K., 2001, Definitions and Theorems for Chaos and Fractals, *lecture monograph*, Course Math-320, University of Scranton.

[Numerical Recipes in Fortran 90 (1996)] Press, W. H., Teukolsky, S. A., Vetterling, W. T. & Flannery, B. P., 1996, Numerical Recipes in Fortran 90 - The Art of Parallel Scientific Computing, 2^{nd} edition, Cambridge University Press.

[Ölveczky & Verkman (1998)] Ölveczky, B. P. & Verkman, A. S., 1998, Monte Carlo Analysis of Obstructed Diffusion in Three Dimensions: Application to Molecular Diffusion in Organelles, *Biophysical Journal*, **74**, 2722-2730.

[Reits & Neefjes (2001)] Reits, E. A. J. & Neefjes, J. J., 2001, From fixed to FRAP: measuring protein mobility and activity in living cells, *Nature Cell Biology*, **Vol. 3**, June 2001, E145-E147.

[Sbalzarini (2001)] Sbalzarini, I. F., 2001, On Protein Diffusion in the Endoplasmic Reticulum - a computational approach using particle simulations, *ICoS/ETHZ semester thesis*, July 2001.

[SGI (1992a)] Silicon Graphics, Inc., 1992, How to write an IRIS Inventor File Translator, Release 1.0.

[SGI (1992b)] Silicon Graphics, Inc., 1992, IRIS Inventor Nodes Quick Reference, Release 1.0.

[Shannon (1948)] Shannon, C. E., 1948, A Mathematical Theory of Communication, *The Bell System Technical Journal*, **27**, 379-656.

[Siggia, Lippincott-Schwartz & Bekiranov (2000)] Siggia, E. D., Lippincott-Schwartz, J. & Bekiranov, S., 2000, Diffusion in Inhomogeneous Media: Theory and Simulations Applied to Whole Cell Photobleach Recovery, *Biophysical Journal*, **79**, 1761-1770.

[Smith, Marks, Lang, Sheriff & Neal (1989)] Smith, T. G. Jr., Marks, W. B., Lang, G. D., Sheriff, W. H. Jr. & Neal, E. A., 1989, A Fractal Analysis of Cell Images, *Journal of Neuroscience Methods*, **27**, 173-180.

[Spekreijse, Hagmeijer & Boerstoel (1996)] Spekreijse, S. P., Hagmeijer, R. & Boerstoel, J. W., 1996, Adaptive grid generation by usgin the Laplace-Beltrami operator on a monitor surface, *National Aerospace Laboratory Technical Report*, TP 96297, Amsterdam, The Netherlands.

[Stoll, Stern & Stucki (1996)] Stoll, E. P., Stern, C. & Stucki, P., 1996, Fractals in isotropic systems generated with attracting spheres, *Physica A*, **230**, 11-18.

[Zimmermann, Koumoutsakos & Kinzelbach (2001)] Zimmermann, S., Koumoutsakos, P. & Kinzelbach, W., 2001, Simulation of Pollutant Transport Using a Particle Method, *Journal of Computational Physics*, **173**, 322-347.

Appendix A

Contents of the companion CD-ROMs

All data files that have been produced in digital form during this diploma project are contained on the two attached CD-ROMs.

The CDs contain this report (source texts and final PS and PDF versions as well as all pictures), all Fortran source code files, shell scripts and perl scripts, some movies and the final PowerPoint presentation of the work. Moreover, all original micrographs are included as well as the original input and output files of all the runs presented in this report and all experimental data. Finally, the CDs also contain some 3^{rd} party papers that are related to this work.

All directories on both CDs contain `README` files with further information on the files and their usage.

CD #2 only contains a single directory (runs) which holds compressed archive files (`.tar.gz`) of all the simulation runs. They can be uncompressed using the command

```
$ tar zxvf <run>.tar.gz
```

This will create a directory called `<run>` that contains all the input and output files of the simulation run according to the information in the `README` file.

The file system of CD #1 is structured as follows:

DA02_Sbalzarini_1/	
3div/	3D reconstructions of all ER samples
codes/	Source code files of all programs
analytic/	Calculates the analytic solution (section 8.2)
bdiff3d/	Random walk simulation code (section 7.1)
run/	Input files and utilities
src/	Source code
concat/	Concatenates PSE3D output files
crayfishpak/	Fast finite difference solver libraries
FD/	Finite difference solver (section 8.3)
nlfit/	Gradient descent algorithm (section 11.3.1)
runs/	Results of all model fits made
src/	Source code
preproc/	Geometry preprocessor (section 7.2.5)
run/	Input files and visualization utilities
src/	Source code
PSE3D/	PSE simulation code (section 7.2)
run/	Input files
src/	Source code
experiments/	Experimental data and evaluation tools used in chapter 10
micrographs/	Stained ER micrograph stacks
8s/	Stack of 8s and 8.2 experiments (chapter 10)
movies/	Movies of experiments and exact solution
utilities/	Scripts and helper programs to make movies
papers/	Some relevant 3^{rd} party papers
postproc/	OpenDX converter and visualization tools presented in section 7.3
presentation/	Final project presentation (PowerPoint)
report/	This report
source/	LaTeX source files of this report
code/	Source code files of appendix C
pics/	All EPS and PS figures of this report
erp572/	Sample micrograph stack erp572 (chapter 2)
masters/	Original xfig drawings and pixel images

Appendix B

Fitted model parameters

B.1 Second order physical model fit

The following table lists the optimal model parameters α_{opt} and β_{opt} for the second order physical model given by equation 11.4 in section 11.2.2 found for the different simulation runs of chapter 9. In order not to loose any information, they are given to one more significant digit than the accuracy they were computed with, well aware of the fact that they cannot be trusted for more than 3 digits. The column E_{min} lists the minimum value of the quadratic (not RMS !) fitting error E as defined by equation 11.6 and N_{Iter} is the number of iterations it took for the gradient descent algorithm to converge up to a tolerance of 10^{-6}. Since all runs started from the same initial point in parameter space, this number is a measure for the distance between the initial and the optimal point. Details about the runs as well as comparative plots are contained in section 11.3.4.

Sample	α_{opt}	β_{opt}	E_{min}	N_{Iter}
bip2	1.9440048	0.63324605	$3.629066 \cdot 10^{-2}$	7924
clx	3.3294665	0.58295736	$1.825963 \cdot 10^{-2}$	11773
erp57	2.4120744	0.63037812	$6.61639 \cdot 10^{-3}$	8881
erp572	2.8619264	0.48297255	$1.12797 \cdot 10^{-2}$	10586
erp573_1	3.2154385	0.62770381	$3.07065 \cdot 10^{-2}$	11444
erp573_2	3.4254690	0.46527276	$1.66678 \cdot 10^{-2}$	11758
erp573_3	2.9901725	0.63353217	$2.574665 \cdot 10^{-2}$	11356
erp574_1	2.7352222	0.51042596	$2.789649 \cdot 10^{-2}$	10467
erp574_2	3.8617008	0.56668043	$3.05717 \cdot 10^{-3}$	12512
erp5722	2.4727129	0.57562438	$2.120234 \cdot 10^{-2}$	9303
erp5723	1.6467670	0.66137837	$5.019553 \cdot 10^{-2}$	6922
erp5724	3.6143039	0.45770528	$1.243710 \cdot 10^{-2}$	18318

Table B.1: Fitting results for the second order physical model

B.2 Empirical model fit

Table B.2 lists the optimal model parameters α_{opt} and $t_{1/2,opt}$ for the empirical model given by equation 11.2 in section 11.1 found for the different simulation runs of chapter 9. In order not to loose any information, they are given to one more significant digit than the accuracy they were computed with, well aware of the fact that they cannot be trusted for more than 3 digits. The column E_{min} lists the minimum value of the quadratic (not RMS !) fitting error E as defined by equation 11.6 and N_{Iter} is the number of iterations it took for the gradient descent algorithm to converge up to a tolerance of 10^{-6}. Since all runs started from the same initial point in parameter space, this number is a measure for the distance between the initial and the optimal point.

Sample	α_{opt}	$t_{1/2,opt}$	E_{min}	N_{Iter}
bip2	0.74410405	12.882408	0.22977	401454
clx	0.66395129	6.6161354	0.0852237	43693
erp57	0.71880026	9.6646334	0.0514976	142477
erp572	0.56902663	13.142333	0.127082	735753
erp573_1	0.71491157	6.0489575	0.12589	38403
erp573_2	0.51711024	11.242306	0.0376993	695268
erp573_3	0.72134601	6.7077755	0.106629	46024
erp574_1	0.59855700	12.441742	0.163978	524343
erp574_2	0.65499708	5.4388216	0.0249841	37575
erp5722	0.66276097	11.301493	0.121126	285086
erp5723	0.80072016	14.398334	0.371151	546584
erp5724	0.66775435	16.289888	1.36757	991209

Table B.2: Fitting results for the empirical model

Appendix C

The simulation codes

For future reference and the purpose of complete transparency and reproducibility of the algorithms developed and used in this work, the source code listings of some programs are given in this appendix. Each section contains information about a different program starting with a general description of the code and its usage, syntax and explanation of all input and output files as well as an overview of the code's structure and calling tree (if meaningful). In all syntax examples, mandatory values are enclosed by triangular brackets `<.>`, optional ones by square brackets `[.]`. The dollar sign `$` symbolizes a command line prompt.

These explanations are then followed by a reprint of the source code listing. Each subroutine is presented in a different subsection preceded by a descriptive header stating its purpose and calling syntax. All programs have been written in the Fortran 90 programming language. For easier reference and readability, statement numbers have been included in the listings to which the algorithms described in this report contain references. They are for documentation purposes only and not present in the original "compilable" files. The files as printed hereafter are as of February 21, 2002. Due to the ongoing nature of the project, they can and will be altered and further developed in the future.

Please note that not all programs utilized in this project are given in this appendix. The following codes are omitted for various reasons (but are contained on CD #1 nevertheless):

- The new version of the random walk code because it only contains minor changes compared to the version that has been printed in [Sbalzarini (2001)] and all the changes are fully documented in section 7.1.
- The PSE simulation code because it is too large and has mainly been developed by other people in other projects. Moreover, none of its algorithms are presented in this report in detail making any references to it obsolete.
- The code that calculates FRAP value predictions based on current models because it is too trivial.
- All the small MATLAB files for visualization and least squares fitting because they are straightforward.
- The tools used to concatenate and average data files because they are implemented in an obvious manner.

C.1 Calculating the analytic solution

C.1.1 General description and usage notes

This section contains the program code used to numerically calculate the analytic solution for the box test case as derived in section 8.2. The code only consists of a single source file and has the following input parameters that are directly hard-coded into its header (statements 4 through 15):

Parameter	Meaning
`Len` (L)	Edge length of the cubic computational domain
`a` (a)	Lower x-coordinate of the bleached box
`b` (b)	Upper x-coordinate of the bleached box
`c` (c)	Lower y-coordinate of the bleached box
`d` (d)	Upper y-coordinate of the bleached box
`dt` (δt)	Time step size
`tfinal` (t_{final})	Final time of solution
`t0` (t_0)	Starting time
`Diff` (D)	Diffusion constant
`c0` (c_0)	Initial concentration outside the bleached box
`N` (N)	Number of grid points in each direction
`M` (M)	Number of terms in the series expansion

Table C.1: Input parameters for `analytical.f90`

The output consists of a set of files called `uXXX.out` where `XXX` stands for the corresponding time step number. These files contain the analytical solution at the time steps indicated. Each line contains the solution at one grid point using the following syntax:

```
<x-pos> <y-pos> <z-pos> <solution u>
```

They are used to visualize the exact solution at certain time steps such as in figure 8.3. Two other output files are called `frap.out` and `mass.out` and they contain the FRAP values and the total mass in the system versus time using the syntax:

```
<time> <value>
```

on every line. They are used to create various FRAP curve plots as well as to check the accuracy of the simulation using the principle of conservation of mass. Examples are given in figures 8.4 and 8.5, respectively. Additionally, the program prints a message like

```
Completed time step    20 of   601
```

to the system's standard output every time it completes a time step. This is for progress monitoring purposes only.

C.1.2 Source code listing

The following source code lists the complete program used to calculate the analytic solution given in section 8.2 as an infinite sum over eigenfunctions. It already contains the high performance Fortran directive needed to parallelize it on a vector processor.

C.1.2.1 Analytic solution (analyt.f90)

```
1     program analytical

2     IMPLICIT NONE

      !------------------------------------------------------------------------------
      !analytical  Calculates the analytic solution for the box test case.
      !   ANALYTICAL calculates the analytic solution for diffusion in a
      !   cube of side length L with zero flux boundary conditions. The initial
      !   condition is 1 everywhere except in the cylinder [a,b]x[c,d]x[0,L]
      !   where it is zero initially. The solution is for homogenious, isotropic
      !   diffusion. In addition, the resulting FRAP curve is calculated and
      !   written to the file frap.out.
      !
      !   THIS IS OPTIMIZED FOR PGHPF HIGH PERFORMANCE FORTRAN (PARALLELISM)
      !
      !==============================================================================
      !  DIPLOMA THESIS WS01/02 ICOS                                    ETH-ZUERICH
      !------------------------------------------------------------------------------
      !
      !              PROTEIN DIFFUSION INSIDE THE ENDOPLASMIC RETICULUM
      !
      !============================== ivo f. sbalzarini ==============================
      !

      !------------------------------------------------------------------------------
      ! Declaration of parameters
      !------------------------------------------------------------------------------

      ! precision (4=single, 8=double)
3     INTEGER, PARAMETER                          :: MK = KIND(1.0D0)

      ! edge length of the cube
4     REAL(MK), PARAMETER                         :: Len = 4.0
      ! lower x coordinate of bleached box
5     REAL(MK), PARAMETER                         :: a = 2.0
      ! upper x coordinate of bleached box
6     REAL(MK), PARAMETER                         :: b = 3.0
      ! lower y coordinate of bleached box
7     REAL(MK), PARAMETER                         :: c = 2.0
      ! upper y coordinate of bleached box
8     REAL(MK), PARAMETER                         :: d = 3.0

      ! time step size
9     REAL(MK), PARAMETER                         :: dt = 0.05
      ! final time to calculate solution up to
10    REAL(MK), PARAMETER                         :: tfinal = 30.0
      ! initial time to start from
11    REAL(MK), PARAMETER                         :: t0 = 0.0

      ! Diffusion coefficient
12    REAL(MK), PARAMETER                         :: Diff = 0.03
      ! initial concentration outside bleached box
13    REAL(MK), PARAMETER                         :: c0 = 1.0

      ! number of grid points in each direction to get solution at
14    INTEGER, PARAMETER                          :: N = 41
      ! number of terms to calculate in the solutions series expansion
15    INTEGER, PARAMETER                          :: M = 300

      !------------------------------------------------------------------------------
      ! Declaration of local variables
      !------------------------------------------------------------------------------

      ! loop counters
16    INTEGER                                     :: i, j ,k, l, istep
      ! the four integral parts of the solution
17    REAL(MK)                                    :: I1, I2, I3, I4
      ! intermediate terms
18    REAL(MK)                                    :: t1, t2, t3, h, im, xm
      ! often used sqrt terms
19    REAL(MK)                                    :: w2ul, w2l, w1ul
      ! grid spacing
20    REAL(MK)                                    :: dx
      ! current time
21    REAL(MK)                                    :: t
      ! current position
22    REAL(MK)                                    :: x, y
```

```
      ! solution
23    REAL(MK), DIMENSION(:,:), ALLOCATABLE       :: u
      ! coordinates of points at which solution is avaluated
24    REAL(MK), DIMENSION(:), ALLOCATABLE         :: px
      ! error trap variable
25    INTEGER                                     :: istat
      ! values of x and y eigenfunctions
26    REAL(MK)                                    :: ukx, uly
      ! eigenvalue
27    REAL(MK)                                    :: ll
      ! FRAP data of all time steps
28    REAL(MK), DIMENSION(:), ALLOCATABLE         :: frap
      ! total mass of all times (to check conservation)
29    REAL(MK), DIMENSION(:), ALLOCATABLE         :: mass
      ! name of current output file
30    CHARACTER(LEN=80)                           :: filename
      ! minimum and maximum time step
31    INTEGER                                     :: Tmin, Tmax
      ! index boundaries of bleached box
32    INTEGER                                     :: iblx, ibux, ibly, ibuy
      ! total number of grid points inside bleached box
33    INTEGER                                     :: nb
      ! the inverse length of the domain
34    REAL(MK)                                    :: Leninv
      ! guess what ...
35    REAL(MK)                                    :: PI

      !-----------------------------------------------------------------------------
      ! Set up solution grid and allocate memory
      !-----------------------------------------------------------------------------

36    PI = 4.0_MK*datan(1.0_MK)
37    dx = Len/(N-1.0)                ! grid spacing
38    Tmax = ceiling(tfinal/dt)       ! number of time steps
39    Tmin = ceiling(t0/dt)
40    iblx = ceiling(a/dx)+1          ! boundary indices of bleached box
41    ibux = floor(b/dx)+1
42    ibly = ceiling(c/dx)+1
43    ibuy = floor(d/dx)+1
44    nb = (ibux-iblx+1)*(ibuy-ibly+1) ! number of pts in bleached box

45    istat = 0
46    ALLOCATE(u(N,N), px(N), frap(Tmax-Tmin+1), mass(Tmax-Tmin+1))
47    if (istat .NE. 0) then
48       WRITE(*,'(2A)') 'Unable to allocate memory for solution.'
49       goto 9999
50    end if

51    px(1:N) = real((/(i,i=0,N-1,1)/))*dx    ! coordinates of points

      !-----------------------------------------------------------------------------
      ! Calculate solution for all time steps and grid points
      !-----------------------------------------------------------------------------

      ! some expressions that occur often
52    w2ul = dsqrt(2.0_MK/Len)
53    w2l = dsqrt(2.0_MK*Len)
54    w1ul = dsqrt(1.0_MK/Len)
55    Leninv = 1.0_MK/Len

      !HPF$ INDEPENDENT
56    do istep=Tmin,Tmax
57       t = real(istep)*dt
58       if(t .EQ. 0.0) then
            ! exact initial condition to avoid Gibbs oscillations
59          u(1:N,1:N) = 1.0_MK
60          u(iblx:ibux,ibly:ibuy) = 0.0_MK
61       else

            !-----------------------------------------------------------------------
            ! Calculate solution for all points by Eigenfunction series expansion
            !-----------------------------------------------------------------------

62          do j=0,N-1
63             y = px(j+1)
64             I1 = c*Leninv
65             I4 = (Len-d)*Leninv
66             do l=M,1,-1            ! sum in reverse order to avoid extinction
67                ll = dexp(-Diff*t*(real(l)*PI*Leninv)**2)
68                h = (2.0_MK/(real(l)*PI))*dcos(real(l)*PI*y*Leninv)
```

```
69                I1 = I1 + (h*dsin(real(l)*PI*c*Leninv)*ll)
70                I4 = I4 - (h*dsin(real(l)*PI*d*Leninv)*ll)
71             end do
72             do i=0,N-1
73                x = px(i+1)
74                I2 = 0.0_MK
75                I3 = 0.0_MK
76                do k=M,0,-1         ! sum in reverse order to avoid extinction
77                   if(k .EQ. 0) then
78                      ukx = w1ul
79                      t1 = ukx*a
80                      t2 = ukx*(Len-b)
81                   else
82                      ukx = w2ul*dcos(real(k)*PI*x*Leninv)
83                      h = w2l/(real(k)*PI)
84                      t1 = h*dsin(real(k)*PI*a*Leninv)
85                      t2 = -h*dsin(real(k)*PI*b*Leninv)
86                   end if
87                   do l=M,0,-1       ! sum in reverse order to avoid extinction
88                      ll = (real(k)*PI*Leninv)**2
89                      ll = ll + (real(l)*PI*Leninv)**2
90                      ll = dexp(-Diff*t*ll)
91                      if(l .EQ. 0) then
92                         uly = w1ul
93                         t3 = uly*(d-c)
94                      else
95                         uly = w2ul*dcos(real(l)*PI*y*Leninv)
96                         t3 = dsin(real(l)*PI*d*Leninv)
97                         t3 = t3-dsin(real(l)*PI*c*Leninv)
98                         t3 = (w2l/(real(l)*PI))*t3
99                      end if
100                     im = ukx*uly*t3*ll
101                     I2 = I2 + im*t1
102                     I3 = I3 + im*t2
103                  end do
104               end do
105               u(i+1,j+1) = c0*(I1+I2+I3+I4)
106            end do
107         end do
108      end if

         !------------------------------------------------------------------------
         ! get current FRAP value and total mass
         !------------------------------------------------------------------------

109      frap(istep-Tmin+1) = 0.0_MK
         ! loop over all cells in the bleached box
110      do j=iblx,ibux-1
111         do k=ibly,ibuy-1
112            xm = u(j,k)+u(j+1,k)+u(j,k+1)+u(j+1,k+1)
113            frap(istep-Tmin+1) = frap(istep-Tmin+1)+xm
114         end do
115      end do
         ! multiply with cell volume to get total mass in bleached box
116      frap(istep-Tmin+1) = dx*dx*0.25_MK*frap(istep-Tmin+1)
         ! divide by bleached box volume to get concentration
117      frap(istep-Tmin+1) = frap(istep-Tmin+1)/((b-a)*(d-c))

118      mass(istep-Tmin+1) = 0.0_MK
         ! loop over all cells
119      do j=1,N-1
120         do k=1,N-1
            ! average the concentrations of all 4 corner vertices of the cell
            ! and multiply with the cells volume to get the mass
121            xm = u(j,k)+u(j+1,k)+u(j,k+1)+u(j+1,k+1)
122            mass(istep-Tmin+1) = mass(istep-Tmin+1)+xm
123         end do
124      end do
125      mass(istep-Tmin+1) = dx*dx*0.25_MK*mass(istep-Tmin+1)

         !------------------------------------------------------------------------
         ! Write solution at current time to file
         !------------------------------------------------------------------------

126      WRITE(filename,'(A,I3.3,A)') 'u',istep,'.out'
127      OPEN(30, FILE=filename, STATUS='REPLACE', ACTION='WRITE')
128      do j=1,N
129         do i=1,N
130            WRITE(30,*) px(i), px(j), u(i,j)
131         end do
```

```
132         WRITE(30,'(A)')
133       end do
134       CLOSE(30)

135       WRITE(*,'(A,I5,A,I5)') 'Completed time step ',istep+1,' of ',Tmax+1

136    end do

       !------------------------------------------------------------------------------
       ! Write FRAP data to file
       !------------------------------------------------------------------------------

137    OPEN(30, FILE='frap.out', STATUS='REPLACE', ACTION='WRITE')
138    do i=Tmin+1,Tmax+1
139       WRITE(30,*) (i-1)*dt, frap(i-Tmin)
140    end do
141    CLOSE(30)
142    OPEN(30, FILE='mass.out', STATUS='REPLACE', ACTION='WRITE')
143    do i=Tmin+1,Tmax+1
144       WRITE(30,*) (i-1)*dt, mass(i-Tmin)
145    end do
146    CLOSE(30)

       !------------------------------------------------------------------------------
       ! Deallocate and terminate
       !------------------------------------------------------------------------------

147    if(ALLOCATED(u)) DEALLOCATE(u)
148    if(ALLOCATED(px)) DEALLOCATE(px)
149    if(ALLOCATED(frap)) DEALLOCATE(frap)
150    if(ALLOCATED(mass)) DEALLOCATE(mass)
151    9999 CONTINUE

152    END program analytical
```

C.2 A finite difference code

C.2.1 General description and usage notes

This finite difference code has been used in section 8.3 to validate the analytic solution for the box test case. The code only consists of a single source file and has the input parameters that are directly hard-coded in statements 30 through 49. A list of them is given in table C.2.

The boundary condition types that can be chosen for `LBDCND`, `MBDCND` and `NBDCND` are: 0 (for periodic boundary conditions), 1 (for Dirichlet) or 3 (for Neumann). The boundary values are assumed to be zero but they could be explicitly set in statements 88 to 93. The initial condition is chosen to be given by equation 8.1 if `INCOND` is set to `.FALSE.`. If it is set to `.TRUE.`, the initial condition is read from a file called `init.in` that contains `nv(1)*nv(2)` lines giving the initial values on the grid points as:

```
<x-pos> <y-pos> <value u0>
```

where the inner loop is over `nv(2)`, the outer over `nv(1)`. The initial condition is then extruded in z direction (i.e. it is assumed to be the same on all z-planes). This enables the exact solution to serve as an initial condition as was done for figure 8.8.

The output consists of the three files `init.out`, `fd.frap` and `fd.mass`. `init.out` contains the positions of all the grid points that have an initial value of zero using the following syntax:

```
for i=1,nv(1)
   for j=1,nv(2)
      for k=1,nv(3)
         if(u(i,j,k)=0)
```

```
            <x-pos> <y-pos> <z-pos>
          end if
        end for
      end for
    end for
end for
```

It mainly serves for control purposes. The files `fd.frap` and `fd.mass` contain the FRAP values and the total mass in the system versus time using the syntax:

```
<time> <value>
```

on every line. They are used to create the various FRAP curve plots as well as to check the accuracy of the simulation using the principle of conservation of mass. Examples are given in figures 8.6 and 8.7, respectively. In addition to above output files, the code also prints some messages to the system's standard output. After successful initialization, it prints:

```
Solver initialized.
```

Then, it prints a message like:

```
Completed time step    20 of  6000
```

every time a time step is completed. At the very end, the following output is given:

```
Solver finished.
```

Parameter	Meaning
`D` (D)	Diffusion constant
`bblx` (a)	Lower x-coordinate of the bleached box
`bbux` (b)	Upper x-coordinate of the bleached box
`bbly` (c)	Lower y-coordinate of the bleached box
`bbuy` (d)	Upper y-coordinate of the bleached box
`dt` (δt)	Time step size
`tfinal` (t_{final})	Final time of solution
`mmin(1)`	Lower x-coordinate of cubic computational domain
`mmin(2)`	Lower y-coordinate of cubic computational domain
`mmin(3)`	Lower z-coordinate of cubic computational domain
`mmax(1)`	Upper x-coordinate of cubic computational domain
`mmax(2)`	Upper y-coordinate of cubic computational domain
`mmax(3)`	Upper z-coordinate of cubic computational domain
`nv(1)`	Number of grid points in x direction
`nv(2)`	Number of grid points in y direction
`nv(3)`	Number of grid points in z direction
`LBDCND`	Boundary condition type in x direction
`MBDCND`	Boundary condition type in y direction
`NBDCND`	Boundary condition type in z direction
`INCOND`	Read initial condition from file or not

Table C.2: Input parameters for `fd.f90`

C.2.2 Source code listing

The source code of the program used to calculate the finite difference solution for the box test case in section 8.3 is given hereafter. It is based on the fast commercial solver library CRAYFISHPAK by Green Mountain software, Madeira Beach, FL, which is needed for the program to compile and run.

C.2.2.1 Finite difference solver (fd.f90)

```
1     program fd

2     IMPLICIT NONE

      !------------------------------------------------------------------------------
      !FD  Calculates a high-resolution finite difference solution for the box
      !   FD uses CRAYFISHPAK solvers to get a finite difference solution
      !   for the box test case in order to validate the analytic solution.
      !
      !   THIS NEEDS THE CRAYFISHPAK LIBRARIES
      !   (commercial and thus not included in this reprint)
      !
      !   See also
      !
      !   todo:
      !
      !==============================================================================
      !  DIPLOMA THESIS WS01/02 ICOS                                    ETH-ZUERICH
      !------------------------------------------------------------------------------
      !
      !              PROTEIN DIFFUSION INSIDE THE ENDOPLASMIC RETICULUM
      !
      !============================== ivo f. sbalzarini ==============================
      !

      ! set precision
3     INTEGER, PARAMETER                     :: MK = KIND(1.0D0)

      !------------------------------------------------------------------------------
      ! Declaration of local variables
      !------------------------------------------------------------------------------

      ! loop conters
4     INTEGER                                :: i, j ,k, l
      ! lower and upper boundaries of physical domain
5     REAL(MK), DIMENSION(3)                 :: mmin, mmax
      ! number of grid points in all 3 directions
6     INTEGER, DIMENSION(3)                  :: nv
      ! boundary condition types (0: periodic, 1: dirichlet, 3: neumann)
7     INTEGER                                :: LBDCND
8     INTEGER                                :: MBDCND
9     INTEGER                                :: NBDCND
      ! boundary data arrays
10    REAL(MK), DIMENSION(:,:), ALLOCATABLE  :: BDXS, BDXF, BDYS, BDYF, BDZS, BDZF
      ! dimensions
11    INTEGER                                :: LDIMF, MDIMF
      ! lambda in Helmholz equation
12    REAL(MK)                               :: lambda
      ! error trap
13    INTEGER                                :: info
      ! solution array
14    REAL(MK), DIMENSION(:,:,:), ALLOCATABLE :: u
      ! work arrays for CRAYFISHPAK
15    REAL(MK), DIMENSION(:), ALLOCATABLE    :: wrk1, wrk2
      ! time step size
16    REAL(MK)                               :: dt
      ! final time of simulation
17    REAL(MK)                               :: tfinal
      ! number of time steps
18    INTEGER                                :: TMAX
      ! diffusion coefficient
19    REAL(MK)                               :: D
      ! dimension of work array needed by CRAYFISHPAK
20    INTEGER                                :: dim
      ! Perturbation returned from solver
21    REAL(MK)                               :: PERTRB
      ! grid spacing in x, y and z
```

```
22    REAL(MK)                                :: dx, dy, dz
      ! lower/upper x/y coordinates of bleached box
23    REAL(MK)                                :: bblx, bbux, bbly, bbuy
      ! lower and upper indices of bleached box in solution grid
24    INTEGER                                 :: il, ih, jl, jh
      ! number of points inside bleached box
25    INTEGER                                 :: nb
      ! array to hold frap data
26    REAL(MK), DIMENSION(:), ALLOCATABLE     :: frap
      ! array to hold total mass at each time step
27    REAL(MK), DIMENSION(:), ALLOCATABLE     :: mass
      ! read initial condition from file?
28    LOGICAL                                 :: INCOND
      ! moving sum for total mass
29    REAL(MK)                                :: xm

      !-----------------------------------------------------------------------------
      ! Problem initialization (THIS IS USER INPUT)
      !-----------------------------------------------------------------------------

      ! diffusion coefficient
30    D = 0.03_MK

      ! bleached box
31    bblx = 2.0_MK
32    bbux = 3.0_MK
33    bbly = 2.0_MK
34    bbuy = 3.0_MK

      ! time step and final time
35    dt = 0.005_MK
36    tfinal = 30.0_MK

      ! physical domain size
37    mmin(1) = 0.0_MK
38    mmin(2) = 0.0_MK
39    mmin(3) = 0.0_MK
40    mmax(1) = 4.0_MK
41    mmax(2) = 4.0_MK
42    mmax(3) = 4.0_MK

      ! number of grid vertices in all three directions
43    nv(1) = 81
44    nv(2) = 81
45    nv(3) = 81

      ! boundary condition: Neumann (3) on all sides
      ! (1 would be Dirichlet and 0 periodic)
46    LBDCND  = 3
47    MBDCND  = 3
48    NBDCND  = 3

      ! read initial condition from file ?
49    INCOND = .false.

      !-----------------------------------------------------------------------------
      ! Initialize derived properties (NO USER INPUT BELOW THIS LINE)
      !-----------------------------------------------------------------------------

      ! grid spacings
50    dx = (mmax(1)-mmin(1))/real(nv(1)-1)
51    dy = (mmax(2)-mmin(2))/real(nv(2)-1)
52    dz = (mmax(3)-mmin(3))/real(nv(3)-1)

      ! number of time steps
53    TMAX = ceiling(tfinal/dt)

      ! lambda in Helmholtz equation
54    lambda = -1.0_MK/(D*dt)

      ! some dimensions (ex CRAYFISHPAK documentation)
55    LDIMF = nv(1) + 4
56    MDIMF = nv(2) + 4
57    dim   = (nv(1)+1)*(nv(2))*(nv(3)) + &
                7*(nv(1)-1) + 3*(nv(2)-1) + 4*(nv(3)-1) + 76

      !-----------------------------------------------------------------------------
      ! Allocate the memory needed
      !-----------------------------------------------------------------------------
```

```
58    ALLOCATE(u(LDIMF,MDIMF,nv(3)), STAT=info)
59    if(info .NE. 0) then
60       WRITE(*,'(A)') 'Error allocating memory for solution array u.'
61       goto 9999
62    end if

63    ALLOCATE(wrk1(dim), wrk2(dim), STAT=info)
64    if(info .NE. 0) then
65       WRITE(*,'(A)') 'Error allocating memory for work arrays.'
66       goto 9999
67    end if

68    ALLOCATE(BDXS(nv(2)+4,nv(3)+4), BDXF(nv(2)+4,nv(3)+4), STAT=info)
69    if(info .NE. 0) then
70       WRITE(*,'(A)') 'Error allocating memory for x boundary arrays.'
71       goto 9999
72    end if

73    ALLOCATE(BDYS(nv(1)+4,nv(3)+4), BDYF(nv(1)+4,nv(3)+4), STAT=info)
74    if(info .NE. 0) then
75       WRITE(*,'(A)') 'Error allocating memory for y boundary arrays.'
76       goto 9999
77    end if

78    ALLOCATE(BDZS(nv(1)+4,nv(2)+4), BDZF(nv(1)+4,nv(2)+4), STAT=info)
79    if(info .NE. 0) then
80       WRITE(*,'(A)') 'Error allocating memory for z boundary arrays.'
81       goto 9999
82    end if

83    ALLOCATE(frap(TMAX+1), mass(TMAX+1), STAT=info)
84    if(info .NE. 0) then
85       WRITE(*,'(A)') 'Error allocating memory for frap and mass arrays.'
86       goto 9999
87    end if

      !------------------------------------------------------------------------------
      ! Set boundary and initial conditions
      !------------------------------------------------------------------------------

      ! boundary condition values (s: lower boundary, f: upper boundary)
      ! here meaning zero gradient at the boundaries
88    BDXS = 0.0_MK
89    BDXF = 0.0_MK
90    BDYS = 0.0_MK
91    BDYF = 0.0_MK
92    BDZS = 0.0_MK
93    BDZF = 0.0_MK

      ! set initial condition
94    u = 1.0_MK
95    il = ceiling(bblx/dx)+1     ! boundaries of bleached box
96    ih = floor(bbux/dx)+1
97    jl = ceiling(bbly/dy)+1
98    jh = floor(bbuy/dy)+1
99    nb = (ih-il+1)*(jh-jl+1)*nv(3)  ! number of particles in bleached box
100   if(INCOND) then
101      OPEN(40, FILE='init.in', STATUS='OLD', ACTION='READ')
         ! read initial condition from file
102      do i=1,nv(1)
103         do j=1,nv(2)
104            READ(40,*) xm, xm, u(i,j,1)
105         end do
106      end do
107      CLOSE(40)
         ! extrude to 3D
108      do i=1,nv(1)
109         do j=1,nv(2)
110            do k=2,nv(3)
111               u(i,j,k) = u(i,j,1)
112            end do
113         end do
114      end do
115   else
116      u(il:ih,jl:jh,:) = 0.0_MK
117   end if

      ! write initial condition to file for check
118   OPEN(40,FILE='init.out',STATUS='REPLACE',ACTION='WRITE')
119   do i=1,LDIMF
```

```
120      do j=1,MDIMF
121         do k=1,nv(3)
122            if(u(i,j,k) .EQ. 0) WRITE(40,*) (i-1)*dx,(j-1)*dy,(k-1)*dx
123         enddo
124      enddo
125   enddo
126   CLOSE(40)

      !------------------------------------------------------------------------
      ! Initialize solver (CALL TO CRAYFISHPAK LIBRARY)
      !------------------------------------------------------------------------

127   CALL H3GCIS(mmin(1),mmax(1),nv(1)-1,LBDCND,   &
                  mmin(2),mmax(2),nv(2)-1,MBDCND,   &
                  mmin(3),mmax(3),nv(3)-1,NBDCND,   &
                  lambda,LDIMF,MDIMF,info,wrk1)

128   if(info .NE. 0) then
129      WRITE(*,'(A)') 'ERROR: Initialization of solver failed!'
130      WRITE(*,'(A,I5)') 'H3GCIS returns: ',info
131      goto 9999
132   else
133      WRITE(*,'(A)') 'Solver initialized.'
134   end if

      !------------------------------------------------------------------------
      ! Main time step loop
      !------------------------------------------------------------------------

135   do i=1,TMAX

136      frap(i) = 0.0_MK
         ! loop over all cells in the bleached box
137      do j=il,ih-1
138         do k=jl,jh-1
139            do l=1,nv(3)-1
140               xm = u(j,k,l)+u(j+1,k,l)+u(j,k+1,l)+u(j,k,l+1)  &
                       +u(j+1,k+1,l)+u(j,k+1,l+1)+u(j+1,k,l+1)+u(j+1,k+1,l+1)
141               frap(i) = frap(i) + xm
142            end do
143         end do
144      end do
         ! multiply with cell volume to get total mass in bleached box
145      frap(i) = dx*dy*dz*0.125_MK*frap(i)
         ! divide by bleached box volume to get concentration
146      frap(i) = frap(i)/((bbux-bblx)*(bbuy-bbly)*(mmax(3)-mmin(3)))

147      mass(i) = 0.0_MK
         ! loop over all cells
148      do j=1,nv(1)-1
149         do k=1,nv(2)-1
150            do l=1,nv(3)-1
               ! average the concentrations of all 8 corner vertices of the cell
151               xm = u(j,k,l)+u(j+1,k,l)+u(j,k+1,l)+u(j,k,l+1)  &
                       +u(j+1,k+1,l)+u(j,k+1,l+1)+u(j+1,k,l+1)+u(j+1,k+1,l+1)
152               mass(i) = mass(i) + xm
153            end do
154         end do
155      end do
         ! multiply with the cells volume to get the mass
156      mass(i) = dx*dy*dz*0.125_MK*mass(i)

         !---------------------------------------------------------------------
         ! Call solver (CALL TO CRAYFISHPAK LIBRARY)
         !---------------------------------------------------------------------

         ! reset perturbation
157      PERTRB = 0.0_MK
         ! new right hand side
158      u = lambda*u
         ! solve
159      CALL H3GCSS(BDXS,BDXF,BDYS,BDYF,BDZS,BDZF,LDIMF,MDIMF,u(1,1,1),  &
                     PERTRB,wrk1,wrk2)
160      if(PERTRB .NE. 0.0_MK) then
161         WRITE(*,'(A,I3,A,E15.5)') 'Time step ',i,': Solver perturbation = ',PERTRB
162      end if
163      WRITE(*,'(A,I5,A,I5)') 'Completed time step ',i,' of ',TMAX
164   end do

      ! get last value too
```

```
165   frap(TMAX+1) = 0.0_MK
166   do j=il,ih-1
167      do k=jl,jh-1
168         do l=1,nv(3)-1
169            xm = u(j,k,l)+u(j+1,k,l)+u(j,k+1,l)+u(j,k,l+1)  &
                    +u(j+1,k+1,l)+u(j,k+1,l+1)+u(j+1,k,l+1)+u(j+1,k+1,l+1)
170            frap(TMAX+1) = frap(TMAX+1) + xm
171         end do
172      end do
173   end do
      ! multiply with cell volume to get total mass in bleached box
174   frap(TMAX+1) = dx*dy*dz*0.125_MK*frap(TMAX+1)
      ! divide by bleached box volume to get concentration
175   frap(TMAX+1) = frap(TMAX+1)/((bbux-bblx)*(bbuy-bbly)*(mmax(3)-mmin(3)))
176   mass(TMAX+1) = 0.0_MK
177   do j=1,nv(1)-1
178      do k=1,nv(2)-1
179         do l=1,nv(3)-1
180            xm = u(j,k,l)+u(j+1,k,l)+u(j,k+1,l)+u(j,k,l+1)  &
                    +u(j+1,k+1,l)+u(j,k+1,l+1)+u(j+1,k,l+1)+u(j+1,k+1,l+1)
181            mass(TMAX+1) = mass(TMAX+1) + xm
182         end do
183      end do
184   end do
185   mass(TMAX+1) = dx*dy*dz*0.125_MK*mass(TMAX+1)

      ! write frap data to file
186   OPEN(40, FILE='fd.frap', STATUS='REPLACE', ACTION='WRITE')
187   do i=1,TMAX+1
188      WRITE(40,*) (i-1)*dt, frap(i)
189   end do
190   CLOSE(40)
191   OPEN(40, FILE='fd.mass', STATUS='REPLACE', ACTION='WRITE')
192   do i=1,TMAX+1
193      WRITE(40,*) (i-1)*dt, mass(i)
194   end do
195   CLOSE(40)

      !------------------------------------------------------------------------------
      ! Free memory and terminate
      !------------------------------------------------------------------------------

196   9999  CONTINUE

197   if(ALLOCATED(mass)) DEALLOCATE(mass)
198   if(ALLOCATED(frap)) DEALLOCATE(frap)
199   if(ALLOCATED(BDZF)) DEALLOCATE(BDZF)
200   if(ALLOCATED(BDZS)) DEALLOCATE(BDZS)
201   if(ALLOCATED(BDYF)) DEALLOCATE(BDYF)
202   if(ALLOCATED(BDYS)) DEALLOCATE(BDYS)
203   if(ALLOCATED(BDXF)) DEALLOCATE(BDXF)
204   if(ALLOCATED(BDXS)) DEALLOCATE(BDXS)
205   if(ALLOCATED(wrk2)) DEALLOCATE(wrk2)
206   if(ALLOCATED(wrk1)) DEALLOCATE(wrk1)
207   if(ALLOCATED(u)) DEALLOCATE(u)

208   WRITE(*,'(A)') 'Solver finished.'

209   END program fd
```

C.3 Post-processing

C.3.1 General description and usage notes

As mentioned in section 7.3, the output files of the PSE simulation code have to be converted to an OpenDX grid file format in order to visualize the concentration fields. The code that implements the corresponding algorithm is given hereafter. It consists of a single source file and does not need any input parameters. The files one wishes to convert are passed to it as command line arguments upon invocation:

```
$ res2dx <file1.res> [file2.res [file3.res ...]]
```

Any number of files can be processed at the same call, shell wildcards are allowed. The parameters `NBX`, `NBY` and `NBZ` on lines 4 to 6 give the numbers of cells in each direction to be used for the linked list algorithm. They can be tuned for better performance. The input files are PSE3D result files containing all the particles and their strengths at a certain time step of the simulation. Each line in such a file contains the information about one particle using the syntax:

```
<x-pos> <y-pos> <z-pos> <strength>
```

For each input file processed, the code outputs the two files `<infile>.grid` and `<infile>.general`. The former contains just one number per line giving the concentration value at that grid position. Hereby it first loops over the z-direction, then over the y-direction and last over the x-direction:

```
for i=1,Nx
   for j=1,Ny
      for k=1,Nz
         <value(i,j,k)>
      end for
   end for
end for
```

The latter contains the OpenDX header according to section 7.3. This is the file that is actually opened in OpenDX. No output is written to the standard output.

C.3.2 Source code listing

The source code of the converter program is given below. For a description of its algorithms including the fast linked list sorting, see section 7.3 of this report.

C.3.2.1 OpenDX grid file converter (res2dx.f90)

```
1     program res2dx

2     IMPLICIT NONE

      !------------------------------------------------------------------------------
      !RES2DX  Converts PSE3D result files to OpenDX grid input format.
      !   res2dx reads a file containing positions and strengths of particles
      !   and converts them to a regular cartesion grid file for OpenDX. It
      !   writes both the data file *.grid and the *.general file containing
      !   all the headers for OpenDX. The list of files to be processed is
      !   passed to res2dx as command line arguments. This program uses a
      !   fast o(N) box sorting with linked particle lists.
      !
      !   See also PSE3D
      !
      !   todo:
      !
      !==============================================================================
      !  DIPLOMA THESIS WS01/02 ICOS                                   ETH-ZUERICH
      !------------------------------------------------------------------------------
      !
      !              PROTEIN DIFFUSION INSIDE THE ENDOPLASMIC RETICULUM
      !
      !============================== ivo f. sbalzarini ==============================
      !

      !------------------------------------------------------------------------------
      ! Declaration of external functions
      !------------------------------------------------------------------------------

      ! get number of command line arguments
3     INTEGER, EXTERNAL                       :: iargc
```

```
      !------------------------------------------------------------------------------
      ! Declaration of local variables
      !------------------------------------------------------------------------------

      ! number of boxes in x,y,z for chaining mesh
4     INTEGER, PARAMETER                        :: NBX = 10
5     INTEGER, PARAMETER                        :: NBY = 10
6     Integer, PARAMETER                        :: NBZ = 10
      ! accuarcy
7     INTEGER, PARAMETER                        :: MK = KIND(1.0D0)
      ! loop counters
8     INTEGER                                   :: i,j,k,l,m
      ! grid spacings in x,y,z
9     REAL(MK)                                  :: dx, dy, dz
      ! number of grid points in x,y,z
10    INTEGER                                   :: Nx, Ny, Nz
      ! bounding box
11    REAL(MK)                                  :: xl,xh,yl,yh,zl,zh
      ! number of points to be read in
12    INTEGER                                   :: Np, Nw
      ! position
13    REAL(MK), DIMENSION(3)                    :: x, xold, xdiff
      ! strength
14    REAL(MK)                                  :: s
      ! input file, output file, header file
15    CHARACTER(LEN=80)                         :: infile, outfile, headfile
      ! name of the program
16    CHARACTER(LEN=80)                         :: prgname
      ! number of command line arguments (input files)
17    INTEGER                                   :: nargc
      ! position and mass data
18    REAL(MK), DIMENSION(:,:), ALLOCATABLE     :: xp
19    REAL(MK), DIMENSION(:), ALLOCATABLE       :: mass
      ! threshold for nearest grid point
20    REAL(MK)                                  :: t
      ! is this grid point found in the data?
21    LOGICAL                                   :: found
      ! error status
22    INTEGER                                   :: istat
      ! head-of-chain for each box
23    INTEGER, DIMENSION(:,:,:), ALLOCATABLE    :: HOC
      ! linked particle list
24    INTEGER, DIMENSION(:), ALLOCATABLE        :: ll
      ! box sizes
25    REAL(MK)                                  :: dbx, dby, dbz
      ! current box index
26    INTEGER                                   :: idx, idy, idz

      !------------------------------------------------------------------------------
      ! Process command line arguments
      !------------------------------------------------------------------------------

      ! read program name from command line
27    CALL getarg(0,prgname)

      ! get number of input files
28    nargc = iargc()
29    if(nargc .LT. 1) then
30       print*,'Missing input file name(s). Usage: ',TRIM(prgname),'  &
    inputfile [inputfile ...]'
31       goto 9999
32    end if

      !------------------------------------------------------------------------------
      ! Loop over all input files and translate them
      !------------------------------------------------------------------------------

33    do i=1,nargc
         ! get input file name
34       CALL getarg(i,infile)
         ! derive other file names
35       WRITE(outfile,'(2A)') TRIM(infile),'.grid'
36       WRITE(headfile,'(2A)') TRIM(infile),'.general'
37       WRITE(*,'(2A)') 'processing file: ',TRIM(infile)

         !---------------------------------------------------------------------------
         ! Scan file and determine grid geometry
         !---------------------------------------------------------------------------

38       WRITE(*,'(A)') 'Parsing data to determine grid geometry ...'
```

```
39       OPEN(40, FILE=infile, STATUS='OLD', ACTION='READ')
         ! first pass: scan file and count number of particles to be
         ! read in. also determine bounding box and grid spacings
40       Np = 0
41       xh = -HUGE(xh)
42       xl = HUGE(xl)
43       yh = -HUGE(yh)
44       yl = HUGE(yl)
45       zh = -HUGE(zh)
46       zl = HUGE(zl)
47       dx = HUGE(dx)
48       dy = HUGE(dy)
49       dz = HUGE(dz)
50       do
51          READ(40,'(4E16.8)',END=110) x(1:3), s
52          if(Np .EQ. 0) xold = x
53          Np = Np + 1
54          xdiff = abs(x-xold)
55          if(x(1) .GT. xh) xh = x(1)
56          if(x(1) .LT. xl) xl = x(1)
57          if(x(2) .GT. yh) yh = x(2)
58          if(x(2) .LT. yl) yl = x(2)
59          if(x(3) .GT. zh) zh = x(3)
60          if(x(3) .LT. zl) zl = x(3)
61          if(xdiff(1) .LT. dx .AND. xdiff(1) .GT. 0.0_MK) dx = xdiff(1)
62          if(xdiff(2) .LT. dy .AND. xdiff(2) .GT. 0.0_MK) dy = xdiff(2)
63          if(xdiff(3) .LT. dz .AND. xdiff(3) .GT. 0.0_MK) dz = xdiff(3)
64          xold = x
65       end do
66    110 CLOSE(40)

         ! determine number of grid points
67       Nx = int((xh-xl)/dx)+1
68       Ny = int((yh-yl)/dy)+1
69       Nz = int((zh-zl)/dz)+1

         ! print terminal output
70       WRITE(*,'(3(A,F8.4))') 'xl = ',xl,' / yl = ',yl,' / zl = ',zl
71       WRITE(*,'(3(A,F8.4))') 'dx = ',dx,' / dy = ',dy,' / dz = ',dz
72       WRITE(*,'(3(A,I8))') 'Nx = ',Nx,' / Ny = ',Ny,' / Nz = ',Nz
73       WRITE(*,'(3(A,F8.4))') 'xh = ',xh,' / yh = ',yh,' / zh = ',zh

         ! write OpenDX *.general header file
74       OPEN(40, FILE=headfile, STATUS='REPLACE', ACTION='WRITE')
75       WRITE(40,'(2A)') 'file = ', TRIM(outfile)
76       WRITE(40,'(A,I4,A,I4,A,I4)') 'grid = ',Nx,' x ',Ny,' x ',Nz
77       WRITE(40,'(A)') 'format = ascii'
78       WRITE(40,'(A)') 'interleaving = record'
79       WRITE(40,'(A)') 'majority = row'
80       WRITE(40,'(A)') 'field = u'
81       WRITE(40,'(A)') 'structure = scalar'
82       WRITE(40,'(A)') 'type = float'
83       WRITE(40,'(A)') 'dependency = positions'
84       WRITE(40,'(A,5(F8.3,A),F8.3)') 'positions = regular,regular,regular,  &
            ',xl,',',dx,',',yl,',',dy,',',zl,',',dz
85       WRITE(40,'(A)')
86       WRITE(40,'(A)') 'end'
87       CLOSE(40)

         !------------------------------------------------------------------------
         ! Allocate memory and read all data
         !------------------------------------------------------------------------

         ! allocate arrays for positions and masses
88       ALLOCATE(xp(3,Np), mass(Np), STAT=istat)
89       if(istat .NE. 0) then
90          WRITE(*,'(A)') 'Error allocating memory for data.'
91          goto 9999
92       end if

         ! allocate arrays for particle sorting and chaining mesh
93       ALLOCATE(HOC(NBX,NBY,NBZ), ll(Np), STAT=istat)
94       if(istat .NE. 0) then
95          WRITE(*,'(A)') 'Error allocating memory for linked list.'
96          goto 9999
97       end if

         ! nullify lists
98       HOC(1:NBX,1:NBY,1:NBZ) = 0
99       ll(1:Np) = 0
```

```
        ! second pass: read points
100     WRITE(*,'(A)') 'Reading data points ...'
101     OPEN(40, FILE=infile, STATUS='OLD', ACTION='READ')
102     do j=1,Np
103        READ(40,'(4E16.8)') xp(1:3,j), mass(j)
104     end do
105     CLOSE(40)

        !------------------------------------------------------------------------
        ! Set up chaining mesh and sort the particles into boxes
        !------------------------------------------------------------------------

106     WRITE(*,'(A)') 'Sorting particles in linked lists ...'

        ! determine box sizes
107     dbx = (xh-xl)/real(NBX)
108     dby = (yh-yl)/real(NBY)
109     dbz = (zh-zl)/real(NBZ)

        ! loop over all particles
110     do j=1,Np
           ! determine index of box we are in
111        idx = ceiling((xp(1,j)-xl)/dbx)
112        if(idx .EQ. 0) idx = 1
113        if(idx .EQ. NBX+1) idx = NBX
114        idy = ceiling((xp(2,j)-yl)/dby)
115        if(idy .EQ. 0) idy = 1
116        if(idy .EQ. NBY+1) idy = NBY
117        idz = ceiling((xp(3,j)-zl)/dbz)
118        if(idz .EQ. 0) idz = 1
119        if(idz .EQ. NBZ+1) idz = NBZ
           ! add particle to linked list of this box
120        ll(j) = HOC(idx,idy,idz)
121        HOC(idx,idy,idz) = j
122     end do

        !------------------------------------------------------------------------
        ! Write regular grid output
        !------------------------------------------------------------------------

        ! write output file
123     WRITE(*,'(A)') 'Writing regular grid output file ...'
124     OPEN(50, FILE=outfile, STATUS='REPLACE', ACTION='WRITE')
125     Nw = 0
        ! threshold for nearest grid point
126     t = (0.5*minval((/dx,dy,dz/)))**2
      !  t = 0.1
        ! loop over all grid points
127     do j=1,Nx
128        do k=1,Ny
129           do l=1,Nz
              ! current position
130           xold(1) = xl+(j-1)*dx
131           xold(2) = yl+(k-1)*dy
132           xold(3) = zl+(l-1)*dz
              ! determine box index
133           idx = ceiling((xold(1)-xl)/dbx)
134           if(idx .EQ. 0) idx = 1
135           if(idx .EQ. NBX+1) idx = NBX
136           idy = ceiling((xold(2)-yl)/dby)
137           if(idy .EQ. 0) idy = 1
138           if(idy .EQ. NBY+1) idy = NBY
139           idz = ceiling((xold(3)-zl)/dbz)
140           if(idz .EQ. 0) idz = 1
141           if(idz .EQ. NBZ+1) idz = NBZ
              ! search for this grid point in the data
142           found = .FALSE.
              ! traverse linked list for all particles in the same box
143           m = HOC(idx,idy,idz)
144           do while (m .NE. 0 .AND. found .EQ. .FALSE.)
                 ! if particle is close enough to grid point, take it and
                 ! exit loop
145              if(sum((xp(1:3,m)-xold(1:3))**2) .LT. t) then
146                 WRITE(50,*) mass(m)
147                 Nw = Nw + 1
148                 found = .TRUE.
149              end if
150              m = ll(m)
151           end do
```

```
                ! if not found, output zero
152             if(.NOT. found) WRITE(50,*) 0.0_MK
153           end do
154         end do
155       end do
156       CLOSE(50)

          !-------------------------------------------------------------------------
          ! Terminate and deallocate memory
          !-------------------------------------------------------------------------

          ! terminal output
157       WRITE(*,'(2(A,I8))') 'Particles read: ',Np,' / Particles written: ',Nw
158       WRITE(*,'(A)')

          ! deallocate memory
159       DEALLOCATE(HOC, ll)
160       DEALLOCATE(xp, mass)

161     end do

162     9999 CONTINUE
163       if(ALLOCATED(HOC)) DEALLOCATE(HOC)
164       if(ALLOCATED(ll)) DEALLOCATE(ll)
165       if(ALLOCATED(xp)) DEALLOCATE(xp)
166       if(ALLOCATED(mass)) DEALLOCATE(mass)

167     END program res2dx
```

C.4 Gradient descent fitting

C.4.1 General description and usage notes

The program code for the gradient descent algorithm described in section 11.3.1 and used in section 11.3.4 and chapter 12 to fit the various FRAP data models to both simulation and experimental data is given next. The program does a nonlinear least squares fit of a model having 1 or 2 parameters called α and β. It uses the slow but robust (cf. the considerations in section 11.3.3) method of gradient descent and consists of a single source file. The following input parameters are specified in statements 5 to 16:

Parameter	Meaning
`TOL` (TOL)	Tolerance for termination criterion
`a`	Size (extension) of the bleached box in x-direction
`b`	Size (extension) of the bleached box in y-direction
`MODEL`	Model to be fitted to the data
`VIZ`	Create visualization output of error function ?
`Nalpha`	Number of visualization points in α-direction
`Nbeta`	Number of visualization points in β-direction
`alpha0`	Lower α bound for visualization
`beta0`	Lower β bound for visualization
`alphamax`	Upper α bound for visualization
`betamax`	Upper β bound for visualization
`POSTOUT`	Create PostScript figures or not

Table C.3: Input parameters for `nlfit.f90`

The parameter `MODEL` specifies which model to use. It can assume the following values: 1 (exponential model given by equation 11.1), 2 (power law model given by equation 11.3), 3 (second order physical model given by equation 11.4) or 4 (empirical model given by equation 11.2). The parameter `VIZ` can be either `.TRUE.` or `.FALSE.`. It specifies whether the error function should be probed on a regular

cartesian grid over the parameter space in order to visualize it prior to the gradient descent. This is for example useful in finding initial parameter values for the optimization or to investigate shape and properties of the error function.

The initial parameter values for α and β need to be specified in statements 35 and 36 in variables `alpha` and `beta`. The corresponding learning rates η_α and η_β are to be defined in statements 37 and 38 in `etaa` and `etab`.

In addition to these parameters, the correct level of the theoretical steady-state FRAP value F_∞ for $t \to \infty$ has to be set in `y0`. The values for all the samples considered in this project are given around statement 34 and one only needs to comment out the right one.

The files containing the data one wishes to fit the model to are passed to the program as command line arguments upon invocation:

```
$ nlfit <file1.frap> [file2.frap [file3.frap ...]]
```

Any number of files can be processed at the same call, shell wildcards are allowed. The input files contain FRAP data points using the following syntax on each line:

```
<time> <FRAP value>
```

The output files that are generated for each such input file are `<infile>.grad`, `<infile>.eta` and `<infile>.mac`. The `.grad` file contains the model fitting error E as defined by equation 11.6 and the error gradients $\frac{\partial E}{\partial \alpha}$ and $\frac{\partial E}{\partial \beta}$ for each iteration step as:

```
<error E> <dEdalpha> <dEdbeta>
```

This information can be used to create convergence plots such as the ones in figures 11.6, 12.3 or 12.5. In addition, these values are also printed to the standard output. The `.eta` file contains the learning rates η_α and η_β of each iteration step as:

```
<etaa> <etab>
```

The `.mac` file finally contains the `gnuplot` macro to plot the data and the best model fit in a picture like figure 11.7 or 12.4. If `POSTOUT` is set to `.TRUE.`, `gnuplot` is instructed to generate PostScript figure files, otherwise the plot is output to a graphics window.

If the parameter `VIZ` is set to `.TRUE.`, an additional output file called `E.out` is written that contains the values of the absolute and relative RMS matching errors on a `Nalpha`×`Nbeta` grid for α between `alpha0` and `alphamax` and β between `beta0` and `betamax`:

```
for i=1,Nbeta
   for j=1,Nalpha
      <alpha> <beta> <E> <Erel>
   end for
end for
```

This data can then be used to create contour plots of the "error landscape" such as the ones in figures 11.3 to 11.5.

C.4.2 Source code listing

The program source code for the gradient descent algorithm and all the visualization is given below.

C.4.2.1 Curve fitting using gradient descent (nlfit.f90)

```
1     program nlfit

2     IMPLICIT NONE

      !-------------------------------------------------------------------------------
      !NLFIT  Does a non-linear model fit using a gradient decent technique.
      !   nlfit reads simulated FRAP data from the file(s) specified on the
      !   command line and estimates the model parameters for various models
      !   to best fit the simulation results in a least squares sense.
      !
      !   See also PSE3D
      !
      !   todo:
      !
      !===============================================================================
      !  DIPLOMA THESIS WS01/02 ICOS                                    ETH-ZUERICH
      !-------------------------------------------------------------------------------
      !
      !              PROTEIN DIFFUSION INSIDE THE ENDOPLASMIC RETICULUM
      !
      !============================= ivo f. sbalzarini ===============================
      !

      !-------------------------------------------------------------------------------
      ! Declaration of external functions
      !-------------------------------------------------------------------------------

      ! get number of command line arguments
3     INTEGER, EXTERNAL                       :: iargc

      !-------------------------------------------------------------------------------
      ! Declaration of parameters
      !-------------------------------------------------------------------------------

      ! precision
4     INTEGER, PARAMETER                      :: MK = KIND(1.0D0)
      ! convergence tolerance (for termination criterion)
5     REAL(MK), PARAMETER                     :: TOL = 1e-6_MK
      ! x and y dimensions of bleached box
6     REAL(MK), PARAMETER                     :: a = 50.0
7     REAL(MK), PARAMETER                     :: b = 50.0
      ! Which model (1=exponential, 2=power law, 3=second order physical,
      ! 4=empirical as in SA,p.14)
8     INTEGER, PARAMETER                      :: MODEL = 3
      ! Do area bombing to visualize function?
9     LOGICAL, PARAMETER                      :: VIZ = .FALSE.
      ! Number of alpha and beta grid points for visualization
10    INTEGER, PARAMETER                      :: Nalpha = 500
11    INTEGER, PARAMETER                      :: Nbeta = 500
      ! lower boundary for alpha and beta for visualization
12    REAL(MK), PARAMETER                     :: alpha0 = 0.1
13    REAL(MK), PARAMETER                     :: beta0 = 0.01
      ! upper boundary for alpha and beta for visualization
14    REAL(MK), PARAMETER                     :: alphamax = 5.0
15    REAL(MK), PARAMETER                     :: betamax = 1.00
      ! Produce postscript gnuplot output?
16    LOGICAL, PARAMETER                      :: POSTOUT = .FALSE.

      !-------------------------------------------------------------------------------
      ! Declaration of local variables
      !-------------------------------------------------------------------------------

      ! loop counters
17    INTEGER                                 :: i, j, k, l
      ! name of the program
18    CHARACTER(LEN=80)                       :: prgname
      ! name of the input and output files
19    CHARACTER(LEN=80)                       :: infile, outfile, gradfile, etafile
      ! number of command line arguments (input files)
20    INTEGER                                 :: nargc
      ! simulated FRAP data from file
21    REAL(MK), DIMENSION(:), ALLOCATABLE     :: frap
      ! number of data points
22    INTEGER                                 :: N
      ! current model value
23    REAL(MK)                                :: y
      ! error status
```

```
24    INTEGER                                 :: istat
      ! model parameters to be fitted
25    REAL(MK)                                :: alpha, beta, alphaold, betaold
      ! gradients of the error function
26    REAL(MK)                                :: dEda, dEdb, dEdaold, dEdbold
      ! time array
27    REAL(MK), DIMENSION(:), ALLOCATABLE     :: t
      ! learning rates for alpha and beta, respectively
28    REAL(MK)                                :: etaa, etab
      ! steady-state value of frap curve
29    REAL(MK)                                :: y0
      ! value of the error function
30    REAL(MK)                                :: E, Erel
      ! sign of the dot product of subsequent gradients
31    INTEGER                                 :: s
      ! intermediates
32    REAL(MK)                                :: den, dyda, dydb
      ! alpha and beta step sizes for visualization
33    REAL(MK)                                :: deltaa, deltab

      !------------------------------------------------------------------------------
      ! steady state values
      !------------------------------------------------------------------------------

      ! y0 = 0.9375                      ! for Box test case
      ! y0 = 481958.0_MK/493850.0_MK    ! for bip2
      ! y0 = 475859.0_MK/488054.0_MK    ! for clx
      ! y0 = 503182.0_MK/509578.0_MK    ! for erp57
      ! y0 = 538092.0_MK/545859.0_MK    ! for erp572
      ! y0 = 463516.0_MK/473466.0_MK    ! for erp573_1
      ! y0 = 482469.0_MK/495373.0_MK    ! for erp573_2
      ! y0 = 319538.0_MK/335480.0_MK    ! for erp573_3
      ! y0 = 469745.0_MK/481922.0_MK    ! for erp574_1
      ! y0 = 481486.0_MK/504770.0_MK    ! for erp574_2
      ! y0 = 539400.0_MK/545859.0_MK    ! for erp5722
      ! y0 = 538412.0_MK/545859.0_MK    ! for erp5723
      ! y0 = 528973.0_MK/545859.0_MK    ! for erp5724
34     y0 = 477774.0_MK/480214.0_MK    ! for 8s9
      ! y0 = 440104.0_MK/480214.0_MK    ! for 8.2
      ! y0 = 0.98_MK

      !------------------------------------------------------------------------------
      ! start values
      !------------------------------------------------------------------------------

35    alpha = 1.0_MK
36    beta = 0.05_MK
37    etaa = 1e-4_MK
38    etab = 5e-5_MK

      !------------------------------------------------------------------------------
      ! Process command line arguments
      !------------------------------------------------------------------------------

      ! read program name from command line
39    CALL getarg(0,prgname)

      ! get number of input files
40    nargc = iargc()
41    if(nargc .LT. 1) then
42       print*,'Missing input file name(s). Usage: ',TRIM(prgname),  &
            ' inputfile [inputfile ...]'
43       goto 9999
44    end if

      !------------------------------------------------------------------------------
      ! Delete old result file
      !------------------------------------------------------------------------------

45    OPEN(40, FILE='All_Params.out', ACTION='WRITE')
46    CLOSE(40, STATUS='DELETE')

      !------------------------------------------------------------------------------
      ! Loop over all input files
      !------------------------------------------------------------------------------

47    do i=1,nargc

         !---------------------------------------------------------------------------
         ! Scan file and read its contents
```

```
        !------------------------------------------------------------------------

        ! get input file name
48      CALL getarg(i,infile)
        ! first pass: count points
49      OPEN(40, FILE=infile, STATUS='OLD', ACTION='READ')
50      N = 0
51      do
52         READ(40,*,END=110) y, y
53         N = N + 1
54      end do
55   110 CLOSE(40)
        ! allocate memory
56      ALLOCATE(frap(N), t(N), STAT=istat)
57      if(istat .NE. 0) then
58         WRITE(*,'(A)') 'Error allocating memory for frap data.'
59         goto 9999
60      end if
        ! second pass: read data
61      OPEN(40, FILE=infile, STATUS='OLD', ACTION='READ')
62      do j=1,N
63         READ(40,*) t(j), frap(j)
64      end do
65      CLOSE(40)

        !------------------------------------------------------------------------
        ! Visualization of error function
        !------------------------------------------------------------------------

66      if(VIZ) then
67         OPEN(40, FILE='E.out', STATUS='REPLACE', ACTION='WRITE')
68         deltab = (betamax-beta0)/real(Nbeta-1,MK)
69         deltaa = (alphamax-alpha0)/real(Nalpha-1,MK)
70         do j=1,Nbeta
71            beta = beta0+(j-1)*deltab
72            do k=1,Nalpha
73               alpha = alpha0+(k-1)*deltaa
74               E = 0.0_MK
75               Erel = 0.0_MK
76               do l=1,N
                    ! current model value
77                  if(MODEL .EQ. 2) then
                       ! POWER MODEL
78                     y = y0*(1.0_MK-(beta*t(l)+1)**(-alpha))
79                  elseif(MODEL .EQ. 1) then
                       ! EXPONENTIAL MODEL
80                     y = y0*(1.0_MK-exp(-alpha*t(l)))
81                  elseif(MODEL .EQ. 3) then
                       ! SECOND ORDER PHYSICAL MODEL
82                     den = (a*b+2.0_MK*(a+b)*alpha*t(l)**beta  &
                            +4.0_MK*alpha**2*t(l)**(2.0_MK*beta))
83                     y = y0*(1.0_MK-a*b/den)
84                  elseif(MODEL .EQ. 4) then
                       ! EMPIRICAL MODEL FROM SA,P.14
85                     den = (t(l)/beta)**alpha
86                     y = y0*den/(1.0_MK+den)
87                  else
88                     print*,'Unknown model parameter specified !'
89                     goto 9999
90                  end if
                    ! error function
91                  E = E + (frap(l)-y)**2
92                  Erel = Erel + ((frap(l)-y)/(y0-frap(l)))**2
93               end do
94               E = sqrt(E/real(N))
95               Erel = sqrt(Erel/real(N))
96               WRITE(40,'(4E12.4)') alpha, beta, E, Erel
97            end do
98         end do
99         CLOSE(40)
100     end if

        ! open some output files
101     WRITE(gradfile,'(2A)') TRIM(infile),'.grad'
102     OPEN(40, FILE=gradfile, STATUS='REPLACE', ACTION='WRITE')
103     WRITE(etafile,'(2A)') TRIM(infile),'.eta'
104     OPEN(50, FILE=etafile, STATUS='REPLACE', ACTION='WRITE')

        !------------------------------------------------------------------------
        ! Gradient decent
```

```
      !------------------------------------------------------------------------

105   k = 0

      ! iterate for alpha and beta
106   do
         ! sum the gradients over all time points
107      dEdaold = dEda
108      if(MODEL .GT. 1) dEdbold = dEdb
109      dEda = 0.0_MK
110      dEdb = 0.0_MK
111      E = 0.0_MK
112      do j=2,N
            ! current model value
113         if(MODEL .EQ. 1) then
               ! EXPONENTIAL MODEL
114            y = y0*(1.0_MK-exp(-alpha*t(j)))
115            dEda = dEda + (y-frap(j))*(y0-y)*t(j)
116         elseif(MODEL .EQ. 2) then
               ! POWER MODEL
117            y = y0*(1.0_MK-(beta*t(j)+1)**(-alpha))
118            dEda = dEda + (y-frap(j))*(y0-y)*log(beta*t(j)+1)
119            dEdb = dEdb + (y-frap(j))*(y0-y)*((alpha*t(j))/(beta*t(j)+1))
120         elseif(MODEL .EQ. 3) then
               ! SECOND ORDER PHYSICAL MODEL
121            den = (a*b+2.0_MK*(a+b)*alpha*t(j)**beta  &
                    +4.0_MK*alpha**2*t(j)**(2.0_MK*beta))
122            y = y0*(1.0_MK-a*b/den)
123            dyda = y0*a*b*den**(-2.0_MK)*(2.0_MK*(a+b)*t(j)**beta   &
                    +8.0_MK*alpha*t(j)**(2.0_MK*beta))
124            dydb = y0*a*b*den**(-2.0_MK)*(2.0_MK*(a+b) &
                    *alpha*t(j)**beta*log(t(j))+8.0_MK*alpha**2*t(j)  &
                    **(2.0_MK*beta)*log(t(j)))
125            dEda = dEda + (y-frap(j))*dyda
126            dEdb = dEdb + (y-frap(j))*dydb
127         elseif(MODEL .EQ. 4) then
               ! EMPIRICAL MODEL FROM SA,P.14
128            den = (t(j)/beta)**alpha
129            y = y0*den/(1.0_MK+den)
130            dyda = y0*(den*log(t(j)/beta)*(1.0_MK+den)**(-1.0_MK)- &
                    den*(1.0_MK+den)**(-2.0_MK)*den*log(t(j)/beta))
131            dydb = y0*(-alpha/(beta*beta)*(t(j)/beta)**(alpha-1.0_MK) &
                    *(1.0_MK+den)**(-1.0_MK)+den*(1.0_MK+den)**(-2.0_MK) &
                    *alpha/(beta*beta)*(t(j)/beta)**(alpha-1.0_MK))
132            dEda = dEda + (y-frap(j))*dyda
133            dEdb = dEdb + (y-frap(j))*dydb
134         else
135            print*,'Unknown model specified.'
136            goto 9999
137         end if
138         E = E + (frap(j)-y)**2
139      end do
140      E = 0.5_MK*E
141      WRITE(40,*) E, dEda, dEdb
142      WRITE(50,*) etaa, etab
143      WRITE(*,*) E, dEda, dEdb

         ! update model parameters
144      alphaold = alpha
145      if(MODEL .GT. 1) betaold = beta
146      alpha = alphaold - etaa*dEda
147      if(MODEL .GT. 1) beta = betaold - etab*dEdb
148      k = k + 1

         ! terminateion criterion
149      if(abs(dEda) .LE. TOL .AND. abs(dEdb) .LE. TOL) then
150         goto 200
151      end if
152   end do
153   CLOSE(40)
154   CLOSE(50)

      !------------------------------------------------------------------------
      ! Output results to terminal and result file
      !------------------------------------------------------------------------

155   200 WRITE(*,'(3A,I8,A)') 'Input file: ', TRIM(infile), ' contains ',N, &
          ' data points'
156    WRITE(*,'(A,I8,A)') 'Solution converged after ',k,' Iterations.'
157    WRITE(*,'(A,E16.8)') 'Best parameters found: alpha = ', alpha
```

```
158       WRITE(*,'(A,E16.8)') '                        beta  = ', beta

159       OPEN(40, FILE='All_Params.out', POSITION='APPEND', ACTION='WRITE')
160       WRITE(40,'(I4,3E16.8,I7)') i, alpha, beta, E, k
161       CLOSE(40)
162       WRITE(*,'(A)')

163      DEALLOCATE(frap, t)

         !-----------------------------------------------------------------------
         ! Write gnuplot comparison macro
         !-----------------------------------------------------------------------

164      WRITE(outfile,'(2A)') TRIM(infile), '.mac'
165      OPEN(40, FILE=outfile, STATUS='REPLACE', ACTION='WRITE')
166      WRITE(40,'(A)') 'set nokey'
167      if(POSTOUT) then
168         WRITE(40,'(A)') 'set term postscript eps color 14'
169         WRITE(40,'(3A)') 'set output ''',TRIM(infile),'.eps'''
170      end if
171      if(MODEL .EQ. 2) then
            ! POWER MODEL
172         WRITE(40,'(3A,F8.5,A,F8.5,A)') 'p ''',TRIM(infile),''' w p, ',y0,  &
               '*(1.0-(',beta,'*x+1.0)\\'
173         WRITE(40,'(A,F8.5,A)') '**(-',alpha,')) w l'
174      elseif(MODEL .EQ. 3) then
            ! SECOND ORDER PHYSICAL MODEL
175         WRITE(40,'(3A,F8.5,A)') 'p ''',TRIM(infile),''' w p, ',y0,'*(1.0-( \\'
176         WRITE(40,'(6(F8.5,A)') a,'*',b,')/(',a,'*',b,'+2.0*(',a,'+',b,')* \\'
177         WRITE(40,'(3(F8.5,A)') alpha,'*x**',beta,'+4.0*',alpha,'**2*x**( \\'
178         WRITE(40,'(F8.5,A)') 2.0*beta,'))) w l'
179      elseif(MODEL .EQ. 4) then
            ! EMPIRICAL MODEL FROM SA,P.14
180         WRITE(40,'(3A,F8.5,A)') 'p ''',TRIM(infile),''' w p, ',y0,'*((x/ \\'
181         WRITE(40,'(4(F8.5,A))') beta,')**',alpha,')/(1.0+(x/',beta,')**  &
               ',alpha,') w l'
182      elseif(MODEL .EQ. 1) then
            ! EXPONENTIAL MODEL
183         WRITE(40,'(3A,F8.5,A)') 'p ''',TRIM(infile),''' w p, ',y0,'*(1.0- \\'
184         WRITE(40, '(A,F8.5,A)') 'exp(',-alpha,'*x)) w l'
185      else
186         print*,'Unknown model specified.'
187         goto 9999
188      end if
189      WRITE(40,'(A)') 'pause -1'
190      CLOSE(40)

191   end do

192   9999 CONTINUE

193   END program nlfit
```

C.5 Geometry preprocessor

C.5.1 Code structure and calling tree

The geometry preprocessor code implements the algorithms described in chapters 3 and 5 as well as section 7.2.5. It is parallelized using MPI and uses C preprocessor directives to encapsulate both the serial version and the parallel version of the code in the same source files. The parallel version is built if the code is preprocessed with `__MPI__` defined:

```
$ cpp -D__MPI__ -P init_part.f90 __init_part.f90
```

to build the serial version, no defines are needed:

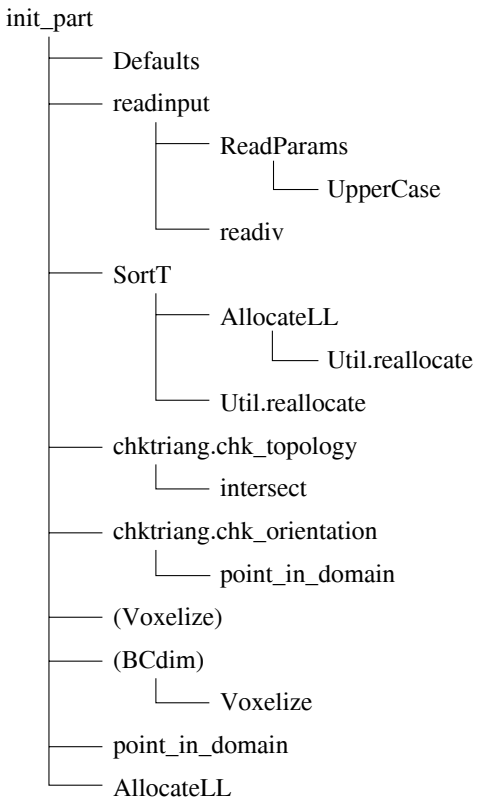
```
$ cpp -P init_part.f90 __init_part.f90
```

`__init_part.f90` then is the program to be compiled. The code consists of the following modules at this stage of development:

Name	Function
`init_part.f90`	Main program
`globals.f90`	Global definitions and variables
`Defaults.f90`	Set default parameter values
`readinput.f90`	Reads all input files
`ReadParams.f90`	Read parameter input file
`readiv.f90`	Read OpenInventor 3D files
`chktriang.f90`	Check triangulation for validity
`intersect.f90`	Intersect a line with a triangle
`SortT.f90`	Sort triangles to bin lists and cell lists
`AllocateLL.f90`	Bin and cell list memory management
`point_in_domain.f90`	Check if a point is inside the computational domain
`Voxelize.f90`	Convert triangulation to voxel representation
`BCdim.f90`	Measure the box counting dimension
`UpperCase.f90`	Convert a string to upper case letters
`Util.f90`	Dynamic list resizing

Table C.4: Modules of the preprocessor code

The code's structure given by its calling tree is:



where "Util.reallocate" means a call to function `reallocate` that is contained in module `Util` and "(BCdim)" means that function `BCdim` can be called if its services are wanted but will generally be commented out (see lines between statement 289 and 290).

C.5.2 Input files and parameters

The code reads two input files: `preproc.in`, which contains all parameters and settings, and a geometry input file containing the triangulated surface description. The parameter input file `preproc.in` uses the same syntax as the one for the PSE code. It contains several lines like:

```
<keyword> = <value>
```

Additionally, it can contain any number of blank lines and comment lines beginning with a hash symbol (`#`). Those lines are ignored by the program. Table C.5 lists all supported keywords, their purpose and default values. If a certain keyword is not explicitly specified in the parameter file, its default value will be assumed. The keyword specifications can occur in any order in the parameter file

Keyword	Meaning	Default value
`surfacefile`	Name of geometry input file to read	box.surf
`restartfile`	Name of particle output file to write	restart.dat
`dx`	Grid spacings in x, y, z	0.1, 0.1, 0.1
`bleachbox`	x_{min}, y_{min}, x_{max} and y_{max} of bleached box	2.0, 2.0, 3.0, 3.0
`cutoff`	PSE cutoff radius (for cell lists)	1.0
`boundarycondition`	BC type: 0=none, 1=Dirichlet, 2=Neumann	0
`gnuout`	Produce `gnuplot` output ?	`.FALSE.`
`checktopology`	Check triangulation for validity ?	`.FALSE.`
`tolerance`	Global geometry tolerance	10^{-10}
`NbinsY`	Number of bins in y-direction (for bin lists)	1
`NbinsZ`	Number of bins in z-direction (for bin lists)	1

Table C.5: Keywords for preprocessor parameter file

For keywords that take more than a single number, a comma separated list is to be given. For `bleachbox`, this list contains the x-coordinate of the lower left corner, the y-coordinate of the lower left corner, the x-coordinate of the upper right corner, the y-coordinate of the upper right corner of the bleached box. As an example, the file for the erp572 simulation run of section 9.1 is given below.

```
# Parameter file for preprocessing tool
# =====================================

# Name of the surface description file to read in:
surfacefile       = ../3div/erp572.iv

# Name of PSE3D restart file to be written
restartfile       = restart.dat

# Lattice spacing in x, y and z direction respectively:
dx                = 2.0, 2.0, 0.4

# Bleached box geometry: lower and upper boundaries in x and y
bleachbox         = 250.0, 125.0, 300.0, 175.0

# Cutoff radius as used in PSE (default: 10*eps)
cutoff            = 22.23

# Type of B.C. (0: none, 1: zero Dirichlet, 2: zero Neumann)
boundarycondition = 2

# Produce additional ASCII output for gnuplot?
```

```
gnuout            = .false.

# Run topology check ?
checktopology     = .true.

# Geometry tolerance (at least 10 times smaller than smallest geometry struct.)
tolerance         = 1e-10

# number of bins in y and z
NbinsY            = 100
NbinsZ            = 100
```

The geometry input file specified using the `surfacefile` keyword in the parameter input file can either be a triangulation `.surf` file as described in [Sbalzarini (2001)] or an OpenInventor 3D file in ASCII format (see [SGI (1992a)] and [SGI (1992b)]). The code recognizes the file type by the first line it contains. If it is

```
#ASCII triangulated surface description
```

the file is considered to be in `.surf`-format according to the specifications in appendix C.2.5 of [Sbalzarini (2001)]. If the first line is

```
#Inventor V2.1 ascii
```

the file is treated as an OpenInventor 3D file and handled by the subroutine `readiv`. The only OpenInventor graphics primitives currently supported are triangulated meshes as for example produced by Imaris (cf. chapter 2).

C.5.3 Output files

The main output file of the preprocessor is the binary restart file to be read by the PSE simulation code. Its name is defined using the `restartfile` keyword in the parameter input file (see above). The restart file contains the positions, initial strengths and attributes[1] of all the particles. Since it is a binary file, one has to make sure that byte order and precision of the target platform are matched. This means that if one wishes to produce initial restart files for a PSE simulation running on a *big endian* machine, the preprocessor has to be compiled using the `-byteswapio` option if it is running on a *little endian* and vice versa. Moreover, it is advisable to use Fortran `KIND` inquiry functions to set the accuracy parameter `MK` (see statements 3 and 4 in `globals.f90`) rather than hard-coding the bit width of a certain machine architecture.

If the parameter `gnuout` in the global parameter input file is set to `.TRUE.`, a set of additional ASCII output files is generated that can be used to visualize the geometry and the particle distribution using `gnuplot` or some other tool. The file `surface.gnu` explicitly contains all the triangles to be visualized in gnuplot using `sp 'surface.gnu'` (see e.g. figures 5.2 and 5.7). The file `mass0.gnu'` contains the positions of all the particles of strength zero (i.e. the ones inside the bleached box) and `mass1.gnu` contains the positions of all those with strength 1. These two files can be used to visualize the initial particle distribution inside the ER geometry such as in figure 9.1. The file `ghost.gnu` finally contains the positions of all boundary condition image particles that are – by definition – in an r_c-neighborhood outside of the computational domain.

[1] > 0 for particles inside the computational domain and < 0 for boundary condition image particles outside the computational domain. The value 0 is reserved for MPI communication layer ghost particles in the PSE code and must not be used

If the routines `Voxelize.f90` and `BCdim.f90` are switched on in order to estimate the box counting dimension of the geometry at hand, three additional files are produced. The first one, `BC.gnu`, contains the values $s_i = \log\left(\frac{1}{\sigma_i}\right)$ and $y_i = \log \eta_i$ for all reduction steps (see section 5.2):

```
<si> <yi>
```

The second one, `plotit.mac`, is a `gnuplot` macro to visualize the measured box counting points and the best linear fit through them (recall that the slope of it corresponds to the box counting dimension of the geometry). The plots in figure 5.11 have for example been made using such automatically generated macros. The third file, `voxels.gnu`, contains the voxel representation of the geometry. Each voxel of value 1 is given by its position in space on a line of the file. The innermost loop is over z, the outermost over x. Thus the file's syntax is

```
for i=1,NVoxx
   for j=1,NVoxy
      for k=1,NVoxz
         <x-pos> <y-pos> <z-pos>
      end for
   end for
end for
```

This voxel representation can then be used to create glyph visualizations (e.g. using OpenDX) such as the ones in figures 5.3 and 5.5.

For debugging and visualization purposes, additional output to create illustrations like the one in figure 7.3 is possible from inside `point_in_domain`. Please see section C.5.4.11 below for further information.

Besides the various file outputs, the program also writes some information about the geometry to the system's standard output (which is a terminal console in most cases). A sample output from the erp572 simulation run of section 9.1 is given hereafter.

```
Initializing particle positions on regular lattice
Spatial resolution (tolerance) is:     0.10E-09

Lattice spacing in x,y,z:   2.00  2.00  0.40

Bleached box has lower left corner at:   250.00  125.00
                      And x-,y-widths:    50.00   50.00

Reading surface description file: ../3div/erp572.iv
Surface triangulation contains  35810 vertices defining  72024 triangles
Triangles ... read.

Bounding box of domain:  xmin =  54.8070  xmax = 414.2720
                         ymin = 102.2590  ymax = 512.7500
                         zmin =  -1.8534  zmax =  17.2687

Calculating centroids and outer normals of triangles ...

Triangles sorted into  100x 100 bins.
Number of entries in largest bin list:    367
Total number of entries in bin lists:   505051

Determining indices of particles inside domain ...
  5%. 10%. 15%. 20%. 25%. 30%. 35%. 40%. 45%. 50%. 55%. 60%. 65%. 70%. 75%. 80%. 85%.
 90%. 95%.100%.
Initializing  545859 particles ...
```

```
Using Neumann boundary conditions.
done.

Wrote restart file restart.dat

Wrote gnuplot files.
```

C.5.4 Source code listings

The following subsections contain the program source of all the modules listed in table C.4. Each section starts with a brief text header explaining the purpose of the routine and highlighting special features of it.

C.5.4.1 Global parameters and variables (globals.f90)

The following module contains the definitions of global parameters and variables used in more than one of the other routines.

```
1     module globals

2     IMPLICIT NONE

      !------------------------------------------------------------------------------
      !GLOBALS  Global variables and definitions for all routines.
      !
      !   See also INIT_PART
      !
      !==============================================================================
      !  DIPLOMA THESIS WS01/02 ICOS                                    ETH-ZUERICH
      !------------------------------------------------------------------------------
      !
      !              PROTEIN DIFFUSION INSIDE THE ENDOPLASMIC RETICULUM
      !
      !============================ ivo f. sbalzarini ===============================
      !

      !------------------------------------------------------------------------------
      ! Global parameters
      !------------------------------------------------------------------------------

      ! Accuracy
3     INTEGER, PARAMETER                        :: MK = KIND(1.0D0)
4     INTEGER, PARAMETER                        :: I4B = SELECTED_INT_KIND(9)

      !------------------------------------------------------------------------------
      ! Global TYPE declarations
      !------------------------------------------------------------------------------

5     TYPE ptr_to_list
6        REAL(MK), DIMENSION(:), POINTER        :: list
7     END TYPE

      !------------------------------------------------------------------------------
      ! Declaration of global variables
      !------------------------------------------------------------------------------

      ! Precision Tolerance for geometry handling (at least 10 times smaller
      ! than smallest occuring geometry structure)
8     REAL(MK)                                  :: TOL
      ! Numer of surface elements
9     INTEGER                                   :: M
      ! Cut-off radius for boundary condition handling (mirroring) and cell list
10    REAL(MK)                                  :: rc
      ! Vertices of all oriented triangles of the domain surface
11    REAL(MK), DIMENSION(:,:,:), ALLOCATABLE   :: triang
      ! centroids of all triangles
12    REAL(MK), DIMENSION(:,:), ALLOCATABLE     :: c
      ! outer normals of all triangles
13    REAL(MK), DIMENSION(:,:), ALLOCATABLE     :: normal
      ! inxed of closest triangle to every particle
14    INTEGER, DIMENSION(:), ALLOCATABLE        :: clotri
      ! distance to boundary of each point
15    REAL(MK), DIMENSION(:), ALLOCATABLE       :: dtb
```

```
      ! Nr. of CPUs and rank of current CPU
16    INTEGER                                 :: ncpu, rank
      ! Precision for MPI
17    INTEGER                                 :: MPI_PREC
      ! Lattice spacings in x, y and z direction
18    REAL(MK)                                :: ldx, ldy, ldz
      ! produce ASCII output for gnuplot?
19    LOGICAL                                 :: GNUOUT
      ! run topology check ?
20    LOGICAL                                 :: CHKTOPO
      ! Bleached box. coordinates of lower left corner and widths
21    REAL(MK)                                :: bbllx,bblly, bbwx,bbwy
      ! upper boundaries in x and y of bleached box
22    REAL(MK)                                :: bbux, bbuy
      ! Name of PSE3D restart file to be written
23    CHARACTER(LEN=80)                       :: resfile
      ! Name of surface description file
24    CHARACTER(LEN=80)                       :: surffile
      ! Type of B.C. (0: none, 1: zero Dirichlet, 2: zero Neumann)
25    INTEGER                                 :: BC
      ! math. constant
26    REAL(MK)                                :: PI
      ! triangle lists for y/z bins (point_in_domain)
27    TYPE(ptr_to_list), DIMENSION(:,:), ALLOCATABLE   :: bin
      ! lengths of y/z lists
28    INTEGER, DIMENSION(:,:), ALLOCATABLE      :: nbin
      ! array of triangle lists for x/y/z cells (boundary condition only)
29    TYPE(ptr_to_list), DIMENSION(:,:,:), ALLOCATABLE :: cell
      ! lengths of x/y/z lists
30    INTEGER, DIMENSION(:,:,:), ALLOCATABLE    :: ncell
      ! Bounding box
31    REAL(MK)                                :: xmin, xmax, ymin, ymax, zmin, zmax
      ! number of bins in y and z
32    INTEGER                                 :: NBINY, NBINZ
      ! y and z sizes of bins
33    REAL(MK)                                :: dby, dbz
      ! number of cells in x,y,z
34    INTEGER                                 :: NCELLX, NCELLY, NCELLZ
      ! x, y and z sizes of cells
35    REAL(MK)                                :: dcx, dcy, dcz
      ! voxel representation of geometry
36    INTEGER, DIMENSION(:,:,:), ALLOCATABLE    :: voxel

37    end module globals
```

C.5.4.2 Main program (init_part.f90)

The main program of the geometry processing tool is given next. It is parallelized using the MPI message passing library. Pay special attention to the MPI testing in statements 45 to 104 which makes sure that all communication channels are up and all processors ready. Load balancing is done in statements 182 to 227 including a small load overview table that is printed to the system's standard output. All calls to MPI are encapsulated in the C preprocessor directives `#ifdef __MPI__` ... `#endif` to make them fully transparent. The region between statements 289 and 290 is commented out and contains the calls to the routines used to translate the triangulation to a voxel representation and to determine its box counting dimension.

```
1     program init_part

2     USE globals

3     IMPLICIT NONE

4     #ifdef __MPI__
5        INCLUDE 'mpif.h'
6     #endif

      !------------------------------------------------------------------------------
      !init_part  Geometry preprocessing and particle initialization for PSE3D.
      !   INIT_PART reads the triangulated geometry description, determines its
      !   bounding box and initializes the particles on a regular lattice.
      !   The particle distribution is then written to a file which can be read
      !   in by PSE3D as a restart file
```

```
      !
      !   See also PSE3D
      !
      !   todo:
      !
      !=========================================================================
      !  DIPLOMA THESIS WS01/02 ICOS                                  ETH-ZUERICH
      !-------------------------------------------------------------------------
      !
      !              PROTEIN DIFFUSION INSIDE THE ENDOPLASMIC RETICULUM
      !
      !============================ ivo f. sbalzarini ==========================
      !

      !-------------------------------------------------------------------------
      ! Declaration of external functions
      !-------------------------------------------------------------------------

      ! checks whether a point is inside the domain
7     INTEGER, EXTERNAL                       :: point_in_domain
      ! Checks the topological validity of the triangulation set
8     LOGICAL, EXTERNAL                       :: chk_topology
      ! Checks the orientation of the triangles (syntactical validity)
9     LOGICAL, EXTERNAL                       :: chk_orientation
      ! returns the box counting dimension of the surface
10    REAL(MK), EXTERNAL                      :: BCdim

      !-------------------------------------------------------------------------
      ! Declaration of local variables
      !-------------------------------------------------------------------------

      ! loop counters
11    INTEGER                                 :: i, j, k, n
      ! total number of particles on local processor and its global sum
12    INTEGER                                 :: Npart, Npsum
      ! number of ghost partcles for B.C. handling and its global sum
13    INTEGER                                 :: Nghost, Ngsum
      ! flag array of which particles are inside the domain
14    INTEGER, DIMENSION(:,:,:), ALLOCATABLE  :: flag
      ! position of current lattice node
15    REAL(MK), DIMENSION(3)                  :: x
      ! positions of particles
16    REAL(MK), DIMENSION(:,:), ALLOCATABLE   :: xp
      ! particle flags
17    INTEGER, DIMENSION(:), ALLOCATABLE      :: ap
      ! concentrations at particle locations
18    REAL(MK), DIMENSION(:), ALLOCATABLE     :: mass
      ! I/O status flag
19    INTEGER                                 :: istat
      ! Number of grid points in x,y,z
20    INTEGER                                 :: Nx, Ny, Nz
      ! number of triangles per slave
21    INTEGER                                 :: Nxslave
      ! local number of triangles
22    INTEGER                                 :: Nxlocal
      ! lower and upper boundary of Nx on local processor
23    INTEGER                                 :: Nxlow, Nxhigh
      ! edge vectors and direction
24    REAL(MK), DIMENSION(3)                  :: a, b
      ! intermediate variables
25    REAL(MK)                                :: rx
      ! Return flag for READIV
26    LOGICAL                                 :: success
      ! percentage done in multiples of 5%
27    INTEGER                                 :: pert
      ! temporary local arrays for dtb and clotri
28    REAL(MK), DIMENSION(:), ALLOCATABLE     :: dtbtemp
29    INTEGER, DIMENSION(:), ALLOCATABLE      :: clotritemp
30    #ifdef __MPI__
         ! MPI init error and MPI status variable
31       INTEGER                           :: ierror, stat(MPI_STATUS_SIZE)
         ! number of characters in processor name
32       INTEGER                           :: ilen
         ! name of current processor
33       CHARACTER(LEN=80)                 :: pname
34    #endif

      ! for the voxel test

35    INTEGER :: voxx, voxy, voxz
```

```
      ! ------------------------

      !-------------------------------------------------------------------------------
      ! Set all global parameters to their default value
      !-------------------------------------------------------------------------------

36    CALL Defaults

      !-------------------------------------------------------------------------------
      ! initialize and test MPI
      !-------------------------------------------------------------------------------

37    #ifdef __MPI__
         ! Define needed precision for MPI
38       if (MK .EQ. 4) then
39          MPI_PREC = MPI_REAL
40       elseif (MK .EQ. 8) then
41          MPI_PREC = MPI_DOUBLE_PRECISION
42       else
43          WRITE (*, '(A,I4)') 'Warning: Unknown precision: ', MK
44       endif

         ! Initialize MPI
45       CALL MPI_Init(ierror)
46       if(ierror .NE. 0) then
47          WRITE(*,'(A)') 'Error: Initializing MPI failed. Aborting.'
48          goto 9999
49       end if
50       CALL MPI_Comm_size(MPI_COMM_WORLD, ncpu, ierror)
51       if(ierror .NE. 0) then
52          WRITE(*,'(A)') 'Error: Cannot determine number of processors. Aborting.'
53          goto 9999
54       end if
55       CALL MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)
56       if(ierror .NE. 0) then
57          WRITE(*,'(A)') 'Error: Cannot determine CPU rank. Aborting.'
58          goto 9999
59       end if
60       if(rank .EQ. 0) then
61          WRITE(*,'(A,I3,A)') 'MPI activated. Starting ',ncpu,' processors ...'
62       end if
63       CALL MPI_Get_Processor_Name(pname, ilen, ierror)
64       WRITE(*,'(3A,I3)') '    -> started on ',pname(1:ilen),' with rank =', rank
65       CALL flush(6)
66       CALL MPI_Barrier(MPI_COMM_WORLD, ierror)
67       if(rank .EQ. 0) WRITE(*,'(A)') 'All processors are up and ready.'

         ! test individual communication
68       if(rank .EQ. 0) then
69          do i=1,ncpu-1
70             WRITE(*,'(A,I3)',ADVANCE='NO') 'Probing rank ',i,' ...      '
71             CALL MPI_SEND(i, 1, MPI_INTEGER, i, i,   &
                    MPI_COMM_WORLD, ierror)
72             CALL MPI_RECV(j, 1, MPI_INTEGER, i, MPI_ANY_TAG,   &
                    MPI_COMM_WORLD, stat, ierror)
73             if(j .EQ. i) WRITE(*,'(A)') '[ passed ]'
74             CALL flush(6)
75          end do
76       else
77          CALL MPI_RECV(j, 1, MPI_INTEGER, 0, MPI_ANY_TAG,   &
                 MPI_COMM_WORLD, stat, ierror)
78          CALL MPI_SEND(j, 1, MPI_INTEGER, 0, rank,   &
                 MPI_COMM_WORLD, ierror)
79       end if

         ! test broadcast
80       i = 0
81       if(rank .EQ. 0) then
82          i = 23
83          Npart = 0
84          WRITE(*,'(A)')
85          WRITE(*,'(A)',ADVANCE='NO') 'Probing broadcast to all ... '
86       end if

87       CALL MPI_BCAST(i, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierror)

88       if(rank .EQ. 0) then
89          do j=1,ncpu-1
90             CALL MPI_RECV(k, 1, MPI_INTEGER, j, MPI_ANY_TAG,   &
                    MPI_COMM_WORLD, stat, ierror)
```

```
91             if(k .EQ. 23) then
92                WRITE(*,'(I3)', ADVANCE='NO') j
93                Npart = Npart + 1
94             end if
95          end do
96          if(Npart .EQ. ncpu-1) then
97             WRITE(*,'(A)') '   [ passed ]'
98          else
99             WRITE(*,'(A)') '   [ failed ]'
100            goto 9999
101         end if
102      else
103         CALL MPI_SEND(i, 1, MPI_INTEGER, 0, rank,   &
                MPI_COMM_WORLD, ierror)
104      end if
105   #else
106      ncpu = 1
107      rank = 0
108   #endif

      !------------------------------------------------------------------------------
      ! Read input files on ROOT
      !------------------------------------------------------------------------------

109   if(rank .EQ. 0) then
110     CALL readinput(success)
111     if(.NOT. success) then
112        WRITE(*,'(A)') 'Failed reading input files. Aborting.'
113        goto 9998
114     end if
        ! dump geometry for GNUPLOT (splot '...' w l)
115     if(GNUOUT) then
116        OPEN(30, FILE='surface.gnu', STATUS='REPLACE')
117        do i=1,M
118           do j=1,3
119              WRITE(30,*) triang(1,j,i), triang(2,j,i), triang(3,j,i)
120           end do
121           WRITE(30,'(A)')
122           WRITE(30,'(A)')
123        end do
124        CLOSE(30)
125     end if
126   end if

      !------------------------------------------------------------------------------
      ! Broadcast geometry to slave processors
      !------------------------------------------------------------------------------

127   #ifdef __MPI__
         ! broadcast number of triangles, global tolerance and grid spacings
128      CALL MPI_BCAST(M, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierror)
129      CALL MPI_BCAST(TOL, 1, MPI_PREC, 0, MPI_COMM_WORLD, ierror)
130      CALL MPI_BCAST(ldx, 1, MPI_PREC, 0, MPI_COMM_WORLD, ierror)
131      CALL MPI_BCAST(ldy, 1, MPI_PREC, 0, MPI_COMM_WORLD, ierror)
132      CALL MPI_BCAST(ldz, 1, MPI_PREC, 0, MPI_COMM_WORLD, ierror)

133      if(rank .NE. 0) then
            ! allocate local memory for triangles on slave
134         istat = 0
            ! first index: x,y,z; second: point number; third: triangle number
135         ALLOCATE(triang(3,3,M), c(3,M), normal(3,M))
136         if (istat .NE. 0) then
137            WRITE(*,'(2A)') 'Unable to allocate memory for triangles on '  &
                  ,pname(1:ilen)
138            goto 9999
139         end if
140      end if

         ! wait for all slaves to allocate memory
141      CALL MPI_Barrier(MPI_COMM_WORLD, ierror)

         ! Broadcast triangles
142      do k=1,M          ! loop over all triangles
143         do j=1,3       ! loop over all vertices
144            CALL MPI_BCAST(triang(:,j,k), 3, MPI_PREC, 0,    &
                  MPI_COMM_WORLD, ierror)
145         end do
146      end do
147   #endif
```

```
     !------------------------------------------------------------------------------
     ! Determine bounding box and set up grid
     !------------------------------------------------------------------------------

148  xmin = HUGE(xmin)
149  xmax = -HUGE(xmax)
150  ymin = HUGE(ymin)
151  ymax = -HUGE(ymax)
152  zmin = HUGE(zmin)
153  zmax = -HUGE(zmax)
154  do i=1,M
155     do j=1,3
156        if (triang(1,j,i) .LT. xmin) xmin = triang(1,j,i)
157        if (triang(1,j,i) .GT. xmax) xmax = triang(1,j,i)
158        if (triang(2,j,i) .LT. ymin) ymin = triang(2,j,i)
159        if (triang(2,j,i) .GT. ymax) ymax = triang(2,j,i)
160        if (triang(3,j,i) .LT. zmin) zmin = triang(3,j,i)
161        if (triang(3,j,i) .GT. zmax) zmax = triang(3,j,i)
162     end do
163  end do

164  if(rank .EQ. 0) then
165     WRITE(*,'(A,F12.4,A,F12.4)') 'Bounding box of domain:  xmin = ',xmin,'  &
           xmax = ',xmax
166     WRITE(*,'(A,F12.4,A,F12.4)') '                         ymin = ',ymin,'  &
           ymax = ',ymax
167     WRITE(*,'(A,F12.4,A,F12.4)') '                         zmin = ',zmin,'  &
           zmax = ',zmax
168     WRITE(*,'(A)')
169  end if

     ! enlarge BBox by tolerance to avoid numerical problems at borders
170  xmin = xmin-TOL
171  xmax = xmax+TOL
172  ymin = ymin-TOL
173  ymax = ymax+TOL
174  zmin = zmin-TOL
175  zmax = zmax+TOL

     ! determine number of particle points
176  Nx = ceiling((xmax-xmin)/ldx)-1
177  Ny = ceiling((ymax-ymin)/ldy)-1
178  Nz = ceiling((zmax-zmin)/ldz)-1
     ! and adjust grid spacing if needed
179  ldx = (xmax-xmin)/real(Nx)
180  ldy = (ymax-ymin)/real(Ny)
181  ldz = (zmax-zmin)/real(Nz)

     ! local chunk sizes
182  Nxslave = nint(real(Nx)/real(ncpu))
183  if(rank .EQ. 0) then
184     Nxlocal = Nx-Nxslave*(ncpu-1)   ! chunk size on ROOT
185     Nxlow = 1               ! lower x index
186     Nxhigh = Nxlocal        ! upper x index
187  else
188     Nxlocal = Nxslave
189     Nxlow = (rank-1)*Nxslave+1+(Nx-Nxslave*(ncpu-1))   ! lower x index
190     Nxhigh = rank*Nxslave+(Nx-Nxslave*(ncpu-1))        ! upper x index
191  end if

192  #ifdef __MPI__
193     if(rank .EQ. 0) then
           ! print nice load allocation table
194        WRITE(*,'(A)',ADVANCE='NO') ' CPU #     |   '
195        do i=0,ncpu-1
196           WRITE(*,'(A,I3,A)',ADVANCE='NO') '   ',i,'  |  '
197        end do
198        WRITE(*,'(A)')
199        WRITE(*,'(A)',ADVANCE='NO') '-----------|-'
200        do i=0,ncpu-1
201           WRITE(*,'(A)',ADVANCE='NO') '----------|'
202        end do
203        WRITE(*,'(A)')
204        WRITE(*,'(A,I6,A)',ADVANCE='NO') ' Nxlocal   |   ',Nxlocal,'  |  '
205        do i=1,ncpu-1
206           WRITE(*,'(I6,A)',ADVANCE='NO') Nxslave,'  |  '
207        end do
208        WRITE(*,'(A)')
209        WRITE(*,'(A,I6,A)',ADVANCE='NO') ' Nxlow     |   ',Nxlow,'  |  '
210        do i=1,ncpu-1
```

```
211           WRITE(*,'(I6,A)',ADVANCE='NO') (i-1)*Nxslave+1+   &
                   (Nx-Nxslave*(ncpu-1)),'  |  '
212        end do
213        WRITE(*,'(A)')
214        WRITE(*,'(A,I6,A)',ADVANCE='NO') ' Nxhigh    |   ',Nxhigh,'  |  '
215        do i=1,ncpu-1
216           WRITE(*,'(I6,A)',ADVANCE='NO') i*Nxslave+(Nx-Nxslave*(ncpu-1)),'  |  '
217        end do
218        WRITE(*,'(A)')
219        WRITE(*,'(A)',ADVANCE='NO') '-----------|-'
220        do i=0,ncpu-1
221           WRITE(*,'(A)',ADVANCE='NO') '----------|'
222        end do
223        WRITE(*,'(A)')
224        WRITE(*,'(A)')
225        CALL flush(6)
226     end if
227  #endif

     !------------------------------------------------------------------------------
     ! Calculate centroids and normals
     !------------------------------------------------------------------------------

228  if(rank .EQ. 0) then
229     WRITE(*,'(A)') 'Calculating centroids and outer normals of triangles ...'
230  end if

231  do i=1,M
        ! calculate centroids of triangles
232     c(:,i) = (1.0/3.0)*(triang(:,1,i)+triang(:,2,i)+triang(:,3,i))
        ! calculate outer normals of triangles
233     a = triang(:,2,i)-triang(:,1,i)
234     b = triang(:,3,i)-triang(:,1,i)
        ! cross product => n
235     normal(1,i) = a(2)*b(3) - a(3)*b(2)
236     normal(2,i) = a(3)*b(1) - a(1)*b(3)
237     normal(3,i) = a(1)*b(2) - a(2)*b(1)
        ! normalize normals
238     rx = sqrt(sum(normal(:,i)**2))
239     normal(:,i) = normal(:,i)/rx
240  end do

     !------------------------------------------------------------------------------
     ! Set up triangle lists and sort triangles
     !------------------------------------------------------------------------------

241  CALL SortT
242  if(rank .EQ. 0) then
243     WRITE(*,'(A)')
244     WRITE(*,'(A,I4,A,I4,A)') 'Triangles sorted into ',NBINY,'x',NBINZ,' bins.'
245     WRITE(*,'(A,I6)') 'Number of entries in largest bin list: ',maxval(nbin)
246     WRITE(*,'(A,I7)') 'Total number of entries in bin lists:  ',sum(nbin)
247     if(BC .GT. 0) then
248        WRITE(*,'(A)')
249        WRITE(*,'(A,I4,A,I4,A,I4,A)') 'Triangles sorted into ',NCELLX,'x',  &
              NCELLY,'x',NCELLZ,' cells.'
250        WRITE(*,'(A,I6)') 'Number of entries in largest cell list: ',maxval(ncell)
251        WRITE(*,'(A,I7)') 'Total number of entries in cell lists:  ',sum(ncell)
252     end if
253     WRITE(*,'(A)')
254  end if

     !------------------------------------------------------------------------------
     ! Check topology and orientation
     !------------------------------------------------------------------------------

255  if(rank .EQ. 0 .AND. CHKTOPO) then
256     WRITE(*,'(A)')
257     if(.NOT. chk_topology()) then
258        WRITE(*,'(A)') 'Triangulation is not topologically valid. Aborting.'
259        goto 9998
260     end if
261     if(.NOT. chk_orientation()) then
262        WRITE(*,'(A)') 'Triangulation is not syntactically valid. Aborting.'
263        goto 9998
264     end if
265     WRITE(*,'(A)')
266  end if

     !------------------------------------------------------------------------------
```

```
      ! Initialize particles on regular lattice
      !-------------------------------------------------------------------------------

      ! determine number of particles
267   if(rank .EQ. 0) then
268      WRITE(*,'(A)') 'Determining indices of particles inside domain ...'
269   end if

      ! allocate temporary flag array
270   if(rank .EQ. 0) then
271      ALLOCATE(flag(Nx,Ny,Nz))
272      if (istat .NE. 0) then
273         WRITE(*,'(A)') 'Unable to allocate memory for flags on ROOT'
274         goto 9998
275      end if
276   else
277      ALLOCATE(flag(Nxlocal,Ny,Nz))
278      if (istat .NE. 0) then
279         WRITE(*,'(A,I3)') 'Unable to allocate memory for flags on CPU',rank
280         goto 9998
281      end if
282   end if

283   if(BC .GT. 0) then
         ! allocate temporary arrays for distances and triangle indices
284      ALLOCATE(dtbtemp(Nxlocal*Ny*Nz), clotritemp(Nxlocal*Ny*Nz))
285      if (istat .NE. 0) then
286         WRITE(*,'(A,I3)') 'Unable to allocate memory for dtb, clotri on CPU',rank
287         goto 9997
288      end if
289   end if

      ! uncomment this to get and save a voxel representation of the
      ! geometry (not needed for determining the box counting dimension
      ! as BCdim will call Voxelize on its own)
      !!$rx = 0.05_MK     ! voxel size
      !!$CALL Voxelize(rx,rx,rx,voxx,voxy,voxz,istat)
      !!$if(ALLOCATED(voxel)) then
      !!$   print*,'vox allocated'
      !!$   print*,'sum of voxels: ',sum(voxel)
      !!$   print*,'number of voxels in x,y,z: ',voxx,voxy,voxz
      !!$   OPEN(50, FILE='voxels.gnu', STATUS='REPLACE', ACTION='WRITE')
      !!$   do i=1,voxx
      !!$      do j=1,voxy
      !!$         do k=1,voxz
      !!$            if(voxel(i,j,k) .EQ. 1) WRITE(50,*) xmin+(i-0.5)*rx,  &
      !!$               ymin+(j-0.5)*rx, zmin+(k-0.5)*rx
      !!$         end do
      !!$      end do
      !!$   end do
      !!$   CLOSE(60)
      !!$   print*,'wrote voxels to file'
      !!$   DEALLOCATE(voxel)
      !!$end if

      ! uncomment this to determine the box counting dimension of the geomerty
      !!$print*,'invoking BCdim ...'
      !!$rx = BCdim((xmax-xmin)/500.0_MK,(ymax-ymin)/500.0_MK,   &
      !!$   (zmax-zmin)/300.0_MK, 0.30_MK, istat)
      !!$if(istat .NE. 0) then
      !!$   WRITE(*,'(A)') 'ERROR in BCdim'
      !!$   STOP
      !!$end if
      !!$print*,'box counting dimension is: ',rx

290   flag = 0   ! 0: grid point outside,
                 ! 1: grid point inside,
                 ! 2: grid point in rc-neighborhood of boundary
291   Npart = 0
292   Nghost = 0
293   pert = 0
294   do i=Nxlow,Nxhigh
295      x(1) = xmin+(real(i)-0.5_MK)*ldx   ! use staggered mesh to avoid
296      do j=1,Ny                          ! having points exactly on the boundary
297         x(2) = ymin+(real(j)-0.5_MK)*ldy! as they would coincide with their
298         do k=1,Nz                       ! images => PSE would be unstable!
299            x(3) = zmin+(real(k)-0.5_MK)*ldz
300            if(BC .GT. 0) then
301               flag(i-Nxlow+1,j,k) = point_in_domain(x,dtbtemp(Npart+1),  &
                        clotritemp(Npart+1))
```

```
302           else
303              flag(i-Nxlow+1,j,k) = point_in_domain(x,rx,n)
304           end if
305           if(flag(i-Nxlow+1,j,k) .GT. 0) Npart = Npart + 1
306           if(flag(i-Nxlow+1,j,k) .EQ. 2) Nghost = Nghost + 1
307           if(rank .EQ. 0) then
                 ! print percentage status every 5%
308              if(((i-1)*Ny*Nz+(j-1)*Nz+k)/(Nxlocal*Ny*Nz+0.0) .GT.   &
                      pert*0.05) then
309                 pert = pert + 1
310                 WRITE(6,'(I3,A)',ADVANCE='NO') pert*5,'%.'
311                 CALL flush(6)        ! force immediate output
312              end if
313           end if
314        end do
315     end do
316  end do
317  WRITE(*,'(A)')

     ! free memory of triangle lists
318  CALL AllocateLL(2,0,0,0,istat)

319  #ifdef __MPI__
        ! wait for all processors to terminate
320     CALL MPI_Barrier(MPI_COMM_WORLD, ierror)
        ! sum Npart onto ROOT
321     CALL MPI_Reduce(Npart, Npsum, 1, MPI_INTEGER, MPI_SUM, 0,   &
             MPI_COMM_WORLD, ierror)
        ! sum Nghost onto ROOT
322     CALL MPI_Reduce(Nghost, Ngsum, 1, MPI_INTEGER, MPI_SUM, 0,   &
             MPI_COMM_WORLD, ierror)
323     if(rank .NE. 0) then
           ! send local part of array flag to ROOT
324        CALL MPI_SEND(flag(:,:,:), Nxlocal*Ny*Nz, MPI_INTEGER, 0, &
              rank, MPI_COMM_WORLD, ierror)
325     else
           ! receive flags from all slaves
326        do i=1,ncpu-1
327           CALL MPI_RECV(flag(Nxlocal+1+(i-1)*Nxslave:Nxlocal+i*Nxslave, &
                 1:Ny,1:Nz),Nxslave*Ny*Nz, MPI_INTEGER, i, i,    &
                 MPI_COMM_WORLD, stat, ierror)
328        end do
329     end if
330  #else
331     Npsum = Npart
332     Ngsum = Nghost
333  #endif

334  if(BC .GT. 0) then
335     if(rank .EQ. 0) then
           ! allocate final arrays of correct size
336        ALLOCATE(dtb(Npsum), clotri(Npsum))
337        if (istat .NE. 0) then
338           WRITE(*,'(A,I3)') 'Unable to allocate memory for dtb, clotri on ROOT'
339           goto 9997
340        end if
           ! copy root part into final arrays
341        dtb(1:Npart) = dtbtemp(1:Npart)
342        clotri(1:Npart) = clotritemp(1:Npart)
343        n = Npart   ! initialize position counter
344  #ifdef __MPI__
           ! receive contributions of all slaves
345        do i=1,ncpu-1
              ! receive number of particles from slave i
346           CALL MPI_RECV(j, 1, MPI_INTEGER, i, i, MPI_COMM_WORLD, stat, ierror)
              ! receive distances to boundary for its points
347           CALL MPI_RECV(dtb(n+1:n+j),j , MPI_PREC, i, i,    &
                 MPI_COMM_WORLD, stat, ierror)
              ! receive indices of closest triangles
348           CALL MPI_RECV(clotri(n+1:n+j),j , MPI_PREC, i, i,    &
                 MPI_COMM_WORLD, stat, ierror)
349        end do
350     else
           ! send my number of particles to ROOT
351        CALL MPI_SEND(Npart, 1, MPI_INTEGER, 0, rank, MPI_COMM_WORLD, ierror)
           ! send the distances to boundary of my particles
352        CALL MPI_SEND(dtbtemp(1:Npart), Npart, MPI_PREC, 0, rank,    &
              MPI_COMM_WORLD, ierror)
           ! send indices of closest triangles
353        CALL MPI_SEND(clotritemp(1:Npart), Npart, MPI_PREC, 0, rank,    &
```

```
                MPI_COMM_WORLD, ierror)
#endif
   end if

   ! deallocate temporary arrays
   DEALLOCATE(dtbtemp)
   DEALLOCATE(clotritemp)
end if

if(rank .EQ. 0) then
   ! be nice and talk to the user
   WRITE(*,'(A,I7,A)') 'Initializing ',Npsum,' particles ...'
   if(BC .EQ. 0) then
      WRITE(*,'(A)') 'Not using boundary conditions.'
   end if
   if(BC .EQ. 1) then
      WRITE(*,'(A,I7,A)') 'Using zero Dirichlet B.C. with ',Ngsum,  &
         ' ghost particles'
   end if
   if(BC .EQ. 2) then
      WRITE(*,'(A,I7,A)') 'Using zero Neumann B.C. with ',Ngsum,  &
         ' ghost particles'
   end if

   ! add ghost particles to the number of particles
   Npsum = Npsum + Ngsum

   ! global array to hold all particles
   istat = 0
   ! first index: x,y,z, second index: particle number
   ALLOCATE(xp(3,Npsum), mass(Npsum), ap(Npsum))
   if (istat .NE. 0) then
      WRITE(*,'(A)') 'Unable to allocate memory for particles on ROOT'
      goto 9997
   end if

   ! initializing particle positions
   n = 0              ! ID of current particle
   Npart = 0          ! total number of particles so far
   do i=1,Nx
      x(1) = xmin+(real(i)-0.5_MK)*ldx
      do j=1,Ny
         x(2) = ymin+(real(j)-0.5_MK)*ldy
         do k=1,Nz
            x(3) = zmin+(real(k)-0.5_MK)*ldz
            if(flag(i,j,k) .GT. 0) then
               n = n + 1
               Npart = Npart + 1
               xp(1:3,Npart) = x(1:3)   ! regular particle
               ap(Npart) = Npart        ! particle ID as attribute
               if(flag(i,j,k) .EQ. 2) then
                  ! boundary particle => needs mirror image as ghost
                  xp(1:3,Npart+1) = x(1:3)+2.0*dtb(n)*normal(1:3,clotri(n))
                  ! negative particle index as attribute
                  ap(Npart+1) = -Npart
                  Npart = Npart + 1
               end if
            end if
         end do
      end do
   end do
end if

! deallocate flags on all processors
DEALLOCATE(flag)

if(rank .EQ. 0) then

   !------------------------------------------------------------------------
   ! initialize particle strengths
   !------------------------------------------------------------------------

   do i=1,Npsum
      if(ap(i) .GT. 0) then    ! regular particles
         if((xp(1,i) .GE. bbllx) .AND. (xp(1,i) .LE. bbllx+bbwx) .AND.    &
              (xp(2,i) .GE. bblly) .AND. (xp(2,i) .LE. bblly+bbwy)) then
            ! no concentration inside bleached box
            mass(i) = 0.0
         else
            ! unit concentration c0 outside
```

```
407              mass(i) = 1.0
408            end if
409          end if
410          if(ap(i) .LT. 0) then                      ! ghost particles
411             if(BC .EQ. 1) mass(i) = -mass(-ap(i))   ! Dirichlet
412             if(BC .EQ. 2) mass(i) = mass(-ap(i))    ! Neumann
413          end if
414       end do
415       WRITE(*,'(A)') 'done.'
416       WRITE(*,'(A)')

          !------------------------------------------------------------------------
          ! Write new restart file for PSE3D
          !------------------------------------------------------------------------

417       OPEN(20, FILE=resfile, ACTION='WRITE', FORM='UNFORMATTED', IOSTAT=istat)
418       CLOSE(20, STATUS='DELETE')
419       OPEN(20, FILE=resfile, ACTION='WRITE', FORM='UNFORMATTED', IOSTAT=istat)
420       WRITE(20)Npsum, 0.0_MK
421       do i=1,Npsum
422          WRITE(20)xp(:,i), mass(i), ap(i)
423       end do
424       CLOSE(20)

425       WRITE(*,'(2A)') 'Wrote restart file ', trim(resfile)
426       WRITE(*,'(A)')

          !------------------------------------------------------------------------
          ! Write gnuplot files for testing purposes
          !------------------------------------------------------------------------

427       if(GNUOUT) then
428          OPEN(20, FILE='mass1.gnu', STATUS='REPLACE', ACTION='WRITE')
429          OPEN(30, FILE='mass0.gnu', STATUS='REPLACE', ACTION='WRITE')
430          OPEN(40, FILE='ghost.gnu', STATUS='REPLACE', ACTION='WRITE')
431          do i=1,Npsum
432             if(ap(i) .GT. 0) then
433                if(mass(i) .EQ. 0.0) then
434                   WRITE(30, *) xp(:,i)
435                else
436                   WRITE(20, *) xp(:,i)
437                end if
438             end if
439             if(ap(i) .LT. 0) then
440                WRITE(40, *) xp(:,i)
441             end if
442          end do
443          CLOSE(20)
444          CLOSE(30)
445          WRITE(*,'(2A)') 'Wrote gnuplot files.'
446       end if

447    end if

       !------------------------------------------------------------------------
       ! Deallocate memory and terminate
       !------------------------------------------------------------------------

448    9997 if(ALLOCATED(dtb)) DEALLOCATE(dtb)
449    if(ALLOCATED(clotri)) DEALLOCATE(clotri)
450    if(ALLOCATED(xp)) DEALLOCATE(xp)
451    if(ALLOCATED(ap)) DEALLOCATE(ap)
452    if(ALLOCATED(mass)) DEALLOCATE(mass)
453    9998 if(ALLOCATED(triang)) DEALLOCATE(triang)
454    if(ALLOCATED(c)) DEALLOCATE(c)
455    if(ALLOCATED(normal)) DEALLOCATE(normal)
456    9999 CONTINUE

457    #ifdef __MPI__
458       CALL MPI_Finalize(ierror)
459    #endif

460    END program init_part
```

C.5.4.3 Set global default values (Defaults.f90)

The following subroutine assigns default values to all global parameters. This is needed to make sure that all parameters contains meaningful values even if they are not explicitly specified in the parameter input file described in section C.5.2.

```
1     subroutine Defaults

2     USE globals

3     IMPLICIT NONE

      !------------------------------------------------------------------------------
      !Defaults  Assigns default values to all parameter variables.
      !   The settings of this file can be overridden by specifying a
      !   different value for a parameter in the parameter input file of
      !   the code.
      !
      !   See also INIT_PART
      !
      !   todo:
      !
      !==============================================================================
      !  DIPLOMA THESIS WS01/02 ICOS                                   ETH-ZUERICH
      !------------------------------------------------------------------------------
      !
      !              PROTEIN DIFFUSION INSIDE THE ENDOPLASMIC RETICULUM
      !
      !============================== ivo f. sbalzarini ==============================
      !

      !------------------------------------------------------------------------------
      !  Specify default values here
      !------------------------------------------------------------------------------

      ! Precision Tolerance for geometry handling (at least 10 times smaller
      ! than smallest occuring geometry structure)
4     TOL                    = 1e-10_MK
      ! Cut-off radius for boundary condition handling (mirroring)
5     rc                     = 1.0_MK
      ! Nr. of CPUs and rank of current CPU
6     ncpu                   = 1
7     rank                   = 0
      ! Lattice spacings in x, y and z direction
8     ldx                    = 0.1_MK
9     ldy                    = 0.1_MK
10    ldz                    = 0.1_MK
      ! produce ASCII output for gnuplot?
11    GNUOUT                 = .FALSE.
      ! run topology check ?
12    CHKTOPO                = .FALSE.
      ! Bleached box. coordinates of lower left corner and widths
13    bbllx                  = 2.0_MK
14    bblly                  = 2.0_MK
15    bbwx                   = 1.0_MK
16    bbwy                   = 1.0_MK
      ! Name of PSE3D restart file to be written
17    resfile                = 'restart.dat'
      ! Name of surface description file
18    surffile               = 'box.surf'
      ! Type of B.C. (0: none, 1: zero Dirichlet, 2: zero Neumann)
19    BC                     = 0
      ! number of bins in y and z
20    NBINY                  = 1
21    NBINZ                  = 1
      ! math. constant
22    if(MK .EQ. KIND(1.0D0)) then
23       PI = 4.0_MK*datan(1.0_MK)   ! double prec version of arctan
24    else
25       PI = 4.0_MK*atan(1.0_MK)    ! single prec version of arctan
26    end if


27    END subroutine Defaults
```

C.5.4.4 Read all input files (readinput.f90)

This routine is responsible for reading all the input files, i.e. the global parameter file and the triangulated geometry. Its argument `res` serves as a status flag. If it is zero upon completion of the routine, no errors occurred. Otherwise the input files contained one or more errors.

```
1     subroutine readinput(res)

2     USE globals

3     IMPLICIT NONE

      !------------------------------------------------------------------------------
      !readinput  Reads the parameter and geometry input files.
      !   READINPUT reads all input files for INIT_PART and initializes
      !   basic variables on the ROOT processor. res will be .TRUE. if
      !   everything was successful, else .FALSE.
      !
      !   See also INIT_PART
      !
      !   todo:
      !
      !==============================================================================
      !  DIPLOMA THESIS WS01/02 ICOS                                    ETH-ZUERICH
      !------------------------------------------------------------------------------
      !
      !              PROTEIN DIFFUSION INSIDE THE ENDOPLASMIC RETICULUM
      !
      !============================ ivo f. sbalzarini ===============================
      !

      !------------------------------------------------------------------------------
      ! Declaration of external functions
      !------------------------------------------------------------------------------

      ! funtion to read and parse the global parameter file
4     INTEGER, EXTERNAL                       :: ReadParams

      !------------------------------------------------------------------------------
      ! Declaration of input/output variables
      !------------------------------------------------------------------------------

      ! .TRUE. if reading was succesful
5     LOGICAL, INTENT(OUT)                    :: res

      !------------------------------------------------------------------------------
      ! Declaration of local variables
      !------------------------------------------------------------------------------

      ! loop counter
6     INTEGER                                 :: i
      ! Return flag for READIV
7     LOGICAL                                 :: success
      ! File format description line
8     CHARACTER(LEN=80)                       :: filefmt
      ! I/O status flag
9     INTEGER                                 :: istat

10    res = .TRUE.

      !------------------------------------------------------------------------------
      ! Read parameter file
      !------------------------------------------------------------------------------

11    if(ReadParams('preproc.in') .NE. 0) then
12       WRITE(*,'(A)') 'Error reading parameter file restart.in! aborting.'
13       res = .FALSE.
14       return
15    end if

      ! if B.C. invalid: set to none
16    if((BC .NE. 0) .AND. (BC .NE. 1) .AND. (BC .NE. 2)) then
17       WRITE(*,'(A)') 'Invalid boundary condition specified. Disabling it.'
18       BC = 0
19    end if
```

```
20    WRITE(*,'(A)')
21    WRITE(*,'(A)') 'Initializing particle positions on regular lattice'
22    WRITE(*,'(A,E12.2)') 'Spatial resolution (tolerance) is: ', TOL
23    WRITE(*,'(A)')
24    WRITE(*,'(A,3F6.2)') 'Lattice spacing in x,y,z: ',ldx,ldy,ldz
25    WRITE(*,'(A)')
26    WRITE(*,'(A,2F8.2)') 'Bleached box has lower left corner at: ',bbllx,bblly
27    WRITE(*,'(A,2F8.2)') '                      And x-,y-widths: ',bbwx,bbwy
28    WRITE(*,'(A)')

      !-------------------------------------------------------------------------
      ! Read in triangulated geometry
      !-------------------------------------------------------------------------

29    WRITE(*,'(2A)') 'Reading surface description file: ', trim(surffile)
30    OPEN(20, FILE=surffile, STATUS='OLD')
      ! determine file format
31    READ(20,'(A)') filefmt
32    if(filefmt .EQ. '#ASCII triangulated surface description') then
33       READ(20,'(A)')
34       READ(20,'(A)')
35       READ(20,'(A)')
36       READ(20,'(A)')
37       READ(20,'(A)')
38       READ(20,'(A)')
39       READ(20,*) M
40       READ(20,'(A)')
41       READ(20,'(A)')
42       READ(20,'(A)')
43       READ(20,'(A)')
44       READ(20,'(A)')
45       READ(20,'(A)')
46       WRITE(*,'(A,I6,A)') 'Surface triangulation contains ',M,' triangles'

47       istat = 0
         ! first index: x,y,z; second: point number; third: triangle number
48       ALLOCATE(triang(3,3,M), c(3,M), normal(3,M))
49       if (istat .NE. 0) then
50          WRITE(*,'(A)') 'Unable to allocate memory for triangles'
51          res = .FALSE.
52          return
53       end if

54       do i=1,M
55          READ(20,*) triang(:,1,i), triang(:,2,i), triang(:,3,i)
56       end do
57       CLOSE(20)
58       WRITE(*,'(A)') 'Triangles ... read.'
59       WRITE(*,'(A)')

60    else if(filefmt(1:9) .EQ. '#Inventor') then
         ! read OpenInventor file format
61       if(filefmt(16:20) .EQ. 'ascii') then
62          CLOSE(20)                        ! readiv expects closed file
63          CALL READIV(surffile, success)   ! read Inventor file
64          if(.NOT. success) then
65             WRITE(*,'(A)') 'Failed reading Inventor file.'
66             res = .FALSE.
67             return
68          end if
69       end if
70       if(filefmt(16:21) .EQ. 'binary') then
71          WRITE(*,'(A)') 'The surface file is a binary Inventor file.'
72          WRITE(*,'(A)') 'Binary files are not supported. Please convert it to an'
73          WRITE(*,'(A)') 'ASCII file using ivcat on a SGI workstation.'
74          res = .FALSE.
75          return
76       end if
77    else
78       WRITE(*,'(A)') 'Unknown geometry file format.'
79       res = .FALSE.
80       return
81    end if

82    CLOSE(20)

83    return

84    END subroutine readinput
```

C.5.4.5 Read parameter input file (ReadParams.f90)

The routine `ReadParams.f90` is called by `readinput.f90` in order to read and parse the global parameter input file according to the syntax rules stated in section C.5.2. The name of the file to be read it passed to it in the argument variable `ctrlfile`. `ReadParams` is a function and it returns zero if no errors occurred, 1 otherwise.

```
1     function ReadParams(ctrlfile)

2     USE globals

3     IMPLICIT NONE

      !------------------------------------------------------------------------------
      !ReadParams  Reads the parameter file "ctrlfile"
      !   returns 0 if no errors occured, 1 otherwise.
      !
      !   See also init_part, readinput
      !
      !   todo:
      !
      !==============================================================================
      !  DIPLOMA THESIS WS01/02 ICOS                                    ETH-ZUERICH
      !------------------------------------------------------------------------------
      !
      !              PROTEIN DIFFUSION INSIDE THE ENDOPLASMIC RETICULUM
      !
      !============================= ivo f. sbalzarini ==============================
      !

      !------------------------------------------------------------------------------
      ! Input/Output arguments
      !------------------------------------------------------------------------------

      ! name of the file to be read
4     CHARACTER(LEN=*), INTENT(IN)          :: ctrlfile
      ! return flag
5     INTEGER                               :: ReadParams

      !------------------------------------------------------------------------------
      ! Declaration of local variables
      !------------------------------------------------------------------------------

      ! loop counters
6     INTEGER                               :: i,j
      ! position index
7     INTEGER                               :: idx,i1,i2
      ! read buffer
8     CHARACTER(LEN=256)                    :: cbuf
      ! file unit to be used for reading the parameter file
9     INTEGER                               :: iUnit
      ! string length, I/O error flag
10    INTEGER                               :: ilen,ios
      ! length of filename
11    INTEGER                               :: ilenctrl
      ! line counter
12    INTEGER                               :: ibc,iline
      ! value and argument of current assignment statement
13    CHARACTER(LEN=256)                    :: cvalue,carg
14    CHARACTER(LEN=256)                    :: bcloc,tvalue
      ! flag to indicate whether the file exists at all
15    LOGICAL                               :: lExist

      !------------------------------------------------------------------------------
      ! Definition of file unit
      !------------------------------------------------------------------------------

16    iUnit = 20

      !------------------------------------------------------------------------------
      ! Check that the parameter file exists
      !------------------------------------------------------------------------------

17    ilenctrl = LEN_TRIM(ctrlfile)         ! length of filename string
18    INQUIRE(FILE=ctrlfile(1:ilenctrl), EXIST=lExist)
19    if(.NOT. lExist) then
20       WRITE(*,'(2A)')'No such file: ',ctrlfile(1:ilenctrl)
```

```
21       ReadParams = 1
22       goto 9999
23    end if

      !--------------------------------------------------------------------------
      ! open the file
      !--------------------------------------------------------------------------

24    OPEN(iUnit, FILE=ctrlfile(1:ilenctrl), IOSTAT=ios, ACTION='READ')
25    if(ios .NE. 0) then
26       WRITE(*,'(2A)')'Failed to open file: ',ctrlfile(1:ilenctrl)
27       ReadParams = 1
28       goto 9999
29    end if

      !--------------------------------------------------------------------------
      ! scan file
      !--------------------------------------------------------------------------

30    iline = 0
31    do
32       iline = iline + 1        ! increment line
33       READ(iUnit,'(A)',END=100,ERR=200) cbuf
34       ilen = LEN_TRIM(cbuf)

         !-----------------------------------------------------------------
         !  Skip comment or empty lines
         !-----------------------------------------------------------------
35       if(ilen .GT. 0 .AND. cbuf(1:1) .NE. '#') then

            !-----------------------------------------------------------------
            !  Remove all spaces
            !-----------------------------------------------------------------
36          j = 0
37          do i=1,ilen
38             if(cbuf(i:i) .NE. ' ') then
39                j = j + 1
40                cbuf(j:j) = cbuf(i:i)
41             end if
42          end do
43          ilen = j     ! update length of string

            !-----------------------------------------------------------------
            !  Find position of =
            !-----------------------------------------------------------------
44          idx = INDEX(cbuf,'=')

            !-----------------------------------------------------------------
            !  Exit if = is missing
            !-----------------------------------------------------------------
45          if(idx .LT. 0) then
46             WRITE(*,'(A,I5)')'Incorrect line: ',iline
47             ReadParams = 1
48             goto 9999
49          end if

            !-----------------------------------------------------------------
            !  Get argument and value
            !-----------------------------------------------------------------
50          carg   = ADJUSTL(cbuf(1:idx-1))
51          cvalue = ADJUSTL(cbuf(idx+1:ilen))

            !-----------------------------------------------------------------
            !  Convert to upper case
            !-----------------------------------------------------------------
52          CALL UpperCase(carg,idx-1)

            !-----------------------------------------------------------------
            !  Parse and read input data
            !-----------------------------------------------------------------
53          if(carg .EQ. 'SURFACEFILE') then
54             READ(cvalue,'(A)',IOSTAT=ios,ERR=200) surffile
55          elseif (carg .EQ. 'RESTARTFILE') then
56             READ(cvalue,'(A)',IOSTAT=ios,ERR=200) resfile
57          elseif (carg .EQ. 'DX') then
58             READ(cvalue,*,IOSTAT=ios,ERR=200) ldx,ldy,ldz
59          elseif (carg .EQ. 'BLEACHBOX') then
60             READ(cvalue,*,IOSTAT=ios,ERR=200) bbllx,bblly,bbux,bbuy
61             bbwx = bbux-bbllx
62             bbwy = bbuy-bblly
```

```
63          elseif (carg .EQ. 'CUTOFF') then
64             READ(cvalue,*,IOSTAT=ios,ERR=200) rc
65          elseif (carg .EQ. 'BOUNDARYCONDITION') then
66             READ(cvalue,*,IOSTAT=ios,ERR=200) BC
67          elseif (carg .EQ. 'GNUOUT') then
68             READ(cvalue,*,IOSTAT=ios,ERR=200) GNUOUT
69          elseif (carg .EQ. 'CHECKTOPOLOGY') then
70             READ(cvalue,*,IOSTAT=ios,ERR=200) CHKTOPO
71          elseif (carg .EQ. 'TOLERANCE') then
72             READ(cvalue,*,IOSTAT=ios,ERR=200) TOL
73          elseif (carg .EQ. 'NBINSY') then
74             READ(cvalue,*,IOSTAT=ios,ERR=200) NBINY
75          elseif (carg .EQ. 'NBINSZ') then
76             READ(cvalue,*,IOSTAT=ios,ERR=200) NBINZ
77          end if
78       end if
79    end do

      !------------------------------------------------------------------------------
      !  Something went wrong if we got here
      !------------------------------------------------------------------------------
80    200   CONTINUE
81    WRITE(*,'(A,I5,2A)') 'Error reading line: ',iline,                    &
         &                 ' of file: ',ctrlfile(1:ilenctrl)
      ilen = LEN_TRIM(cbuf)
82    WRITE(*,'(A)') cbuf(1:ilen)
83    ReadParams = 1
84    goto 9999

      !------------------------------------------------------------------------------
      !  End of file
      !------------------------------------------------------------------------------
85    100   ReadParams = 0

      !------------------------------------------------------------------------------
      !  Close file
      !------------------------------------------------------------------------------
86    CLOSE(iUnit)

      !------------------------------------------------------------------------------
      !  Return
      !------------------------------------------------------------------------------
87    9999  CONTINUE

88    return

89    END function ReadParams
```

C.5.4.6 Read OpenInventor 3D files (readiv.f90)

The following routine implements the elementary OpenInventor 3D file reader needed to be able to read Imaris' triangulated output (cf. chapter 2) according to the syntax standards given in [SGI (1992a)] and [SGI (1992b)]. The name of the OpenInventor file to be read is passed to this subroutine in the argument variable `ivfile`. The other argument `success` is the return value of type `LOGICAL` which contains `.TRUE.` if the file was successfully read, `.FALSE.` otherwise.

```
1     subroutine readiv(ivfile, success)

2     USE globals

3     IMPLICIT NONE

      !------------------------------------------------------------------------------
      !readiv  Reads the ASCII OpenInventor file ivfile.
      !   READIV is a very minimal Inventor file reader that only contains the
      !   Nodes and objects found in imaris output. The variable success is
      !   .TRUE. if reading the file was successful, .FALSE. otherwise.
      !
      !   See also INIT_PART
      !
      !   todo:
      !
      !==============================================================================
```

```
      !  DIPLOMA THESIS WS01/02 ICOS                                      ETH-ZUERICH
      !-------------------------------------------------------------------------------
      !
      !              PROTEIN DIFFUSION INSIDE THE ENDOPLASMIC RETICULUM
      !
      !============================ ivo f. sbalzarini ==============================
      !

      !-------------------------------------------------------------------------------
      ! Declaration of input/output variables
      !-------------------------------------------------------------------------------

      ! Name of file to read
4     CHARACTER(LEN=80), INTENT(IN)                 :: ivfile
      ! Read successful?
5     LOGICAL, INTENT(OUT)                          :: success

      !-------------------------------------------------------------------------------
      ! Declaration of local variables
      !-------------------------------------------------------------------------------

      ! loop counters
6     INTEGER                                       :: i
      ! I/O error trap
7     INTEGER                                       :: istat
      ! read buffer
8     CHARACTER(LEN=80)                             :: buf
      ! separator dummy
9     INTEGER                                       :: dummy
      ! Number of points
10    INTEGER                                       :: Npts
      ! Coordinates of vertices
11    REAL(MK), DIMENSION(:,:), ALLOCATABLE         :: pts
      ! vertex indices of current triangle
12    INTEGER, DIMENSION(3)                         :: vertex

13    success = .TRUE.
14    istat = 0
15    buf(:) = ' '

      !-------------------------------------------------------------------------------
      ! first pass through file to count number of points and triangles
      !-------------------------------------------------------------------------------

16    OPEN(80, FILE=ivfile, STATUS='OLD', ACTION='READ', IOSTAT=istat)
17    if(istat .NE. 0) then
18       WRITE(*,'(A)') 'Could not open file ', trim(ivfile)
19       success = .FALSE.
20       goto 900
21    end if

      ! Skip file header
22    do while(index(buf, 'point', .FALSE.) .EQ. 0)
23       READ(80, '(A)') buf
24    end do

      ! determine number of vertices
25    Npts = 0
26    do while(index(buf, '}', .FALSE.) .EQ. 0)
27       Npts = Npts + 1
28       READ(80, '(A)') buf                    ! read next line
29    end do

      ! skip normals section
30    do while(index(buf, 'coordIndex', .FALSE.) .EQ. 0)
31       READ(80, '(A)') buf
32    end do

      ! determine number of triangles
33    M = 0
34    do while(index(buf, '}', .FALSE.) .EQ. 0)
35       if(index(buf, '-1', .FALSE.) .EQ. index(buf, '-1', .TRUE.)) then
36          M = M + 1
37       else
38          M = M + 2
39       end if
40       READ(80, '(A)') buf
41    end do
42    WRITE(*,'(A,I6,A,I6,A)') 'Surface triangulation contains ',Npts,  &
         ' vertices defining ',M,' triangles'
```

```
43    CLOSE(80)

      !-------------------------------------------------------------------------------
      ! allocate memory for points and triangles
      !-------------------------------------------------------------------------------

44    istat = 0
      ! first index: x,y,z; second: point number; third: triangle number
45    ALLOCATE(triang(3,3,M), c(3,M), normal(3,M), pts(3,Npts))
46    if (istat .NE. 0) then
47       WRITE(*,'(A)') 'Unable to allocate memory for triangles and vertives'
48       success = .FALSE.
49       goto 900
50    end if

      !-------------------------------------------------------------------------------
      ! second pass through file to read data
      !-------------------------------------------------------------------------------

51    OPEN(80, FILE=ivfile, STATUS='OLD', ACTION='READ', IOSTAT=istat)
52    if(istat .NE. 0) then
53       WRITE(*,'(A)') 'Could not open file ', trim(ivfile)
54       success = .FALSE.
55       goto 900
56    end if

      ! Skip file header
57    do while(index(buf, 'point', .FALSE.) .EQ. 0)
58       READ(80, '(A)') buf
59    end do
60    buf = buf(index(buf,'[')+1:)

      ! read point coordinates
61    do i=1,Npts
62       READ(buf,*) pts(1:3,i)             ! parse string for numbers
63       READ(80, '(A)') buf                      ! read next line
64    end do

      ! skip normals section
65    do while(index(buf, 'coordIndex', .FALSE.) .EQ. 0)
66       READ(80, '(A)') buf
67    end do
68    buf = buf(index(buf,'[')+1:)

69    i = 1
70    do while(i .LE. M)
71       if(index(buf, '-1', .FALSE.) .EQ. index(buf, '-1', .TRUE.)) then
            ! only one triangle on this line
72          READ(buf,*) vertex(1:3)
73          triang(1:3,1,i) = pts(1:3,vertex(1))
74          triang(1:3,2,i) = pts(1:3,vertex(2))
75          triang(1:3,3,i) = pts(1:3,vertex(3))
76          i = i + 1
77       else                                     ! two triangles on this line
78          READ(buf,*) vertex(1:3)
            ! +1 because .iv file has C style ordering starting at zero
79          triang(1:3,1,i) = pts(1:3,vertex(1)+1)
80          triang(1:3,2,i) = pts(1:3,vertex(2)+1)
81          triang(1:3,3,i) = pts(1:3,vertex(3)+1)
82          i = i + 1
83          buf = buf(index(buf,'-1')+3:)
84          READ(buf,*) vertex(1:3)
85          triang(1:3,1,i) = pts(1:3,vertex(1)+1)
86          triang(1:3,2,i) = pts(1:3,vertex(2)+1)
87          triang(1:3,3,i) = pts(1:3,vertex(3)+1)
88          i = i + 1
89       end if
90       READ(80, '(A)') buf                      ! read next line
91    end do

92    CLOSE(80)

93    WRITE(*,'(A)') 'Triangles ... read.'
94    WRITE(*,'(A)')

95    900 if(ALLOCATED(pts)) DEALLOCATE(pts)
96    return

97    END subroutine readiv
```

C.5.4.7 Check validity of triangulation (chktriang.f90)

The following file contains the two function `chk_topology` and `chk_orientation`. They check the triangulation for its topological and syntactical validity, respectively, according to the criteria and algorithms given in chapter 3. Both functions return a value of type `LOGICAL` which is `.TRUE.` if all criteria are fulfilled and `.FALSE.` if one of them is not. Notice that `chk_orientation` needs the centroids and normals of all the triangles and cannot be called before they have been calculated in the main program (statements 231 to 240). Moreover, it expects the triangulation to be at least topologically valid and should thus always be called *after* `chk_topology`.

```
1     function chk_topology()

2     USE globals

3     IMPLICIT NONE

      !------------------------------------------------------------------------------
      !chk_topology  Checks topological validity of a triangulation set
      !   CHK_TOPOLOGY checks the following requirements:
      !      A) All edges must be of length > 0
      !      B) No vertex must be inside another triangle
      !      C) Every triangle must have exactly three neighbors sharing 2 vertices
      !      D) No edge of any triangle must intersect any other triangle
      !   if all points are met, the triangulation describes a closed coherent
      !   surface in space. Attn.: this is o(M**2)!
      !
      !   RETURN VALUE: .TRUE. if set is topologically valid, else .FALSE.
      !
      !   See also INIT_PART
      !
      !   todo:
      !
      !==============================================================================
      !  DIPLOMA THESIS WS01/02 ICOS                                    ETH-ZUERICH
      !------------------------------------------------------------------------------
      !
      !              PROTEIN DIFFUSION INSIDE THE ENDOPLASMIC RETICULUM
      !
      !============================ ivo f. sbalzarini ===============================
      !

      !------------------------------------------------------------------------------
      ! Declaration of input arguments and return value
      !------------------------------------------------------------------------------

      ! return value: .TRUE. if set is topologically valid
4     LOGICAL                                 :: chk_topology

      !------------------------------------------------------------------------------
      ! Declaration of external functions
      !------------------------------------------------------------------------------

      ! intersects a line with a triangle
5     REAL(MK), EXTERNAL                      :: intersect

      !------------------------------------------------------------------------------
      ! Declaration of local variables
      !------------------------------------------------------------------------------

      ! loop counters
6     INTEGER                                 :: i ,j, k, l
      ! edge vectors
7     REAL(MK), DIMENSION(3)                  :: a, b, g
      ! determinant of a matrix
8     REAL(MK)                                :: det
      ! Number of common vertices of two triangles
9     INTEGER                                 :: Nvert
      ! Number of neighbors of a triangle
10    INTEGER                                 :: Neigh
      ! Matrix of a 2x2 system
11    REAL(MK), DIMENSION(2,2)                :: S
      ! line coefficients
12    REAL(MK)                                :: alpha, beta
      ! local condition variable
```

```
13    LOGICAL                                  :: lcond

14    chk_topology = .TRUE.

15    WRITE(*,'(A)') 'Checking topology of triangulation set ...'
16    WRITE(*,'(A,I6,A)') 'Set contains ', M, ' triangles.'

      !------------------------------------------------------------------------------
      ! a) All edges must be > 0
      !------------------------------------------------------------------------------

17    lcond = .TRUE.
      ! Main loop over all triangles
18    do i=1,M
19       if(sum((triang(:,2,i)-triang(:,1,i))**2) .LT. TOL) then
20          WRITE(*,'(A,I6,A,E12.2)') '*** WARNING: Edge between vertices 1   &
               and 2 of triangle ',i,' is < ', TOL
21          chk_topology = .FALSE.
22          lcond = .FALSE.
23       end if
24       if(sum((triang(:,3,i)-triang(:,1,i))**2) .LT. TOL) then
25          WRITE(*,'(A,I6,A,E12.2)') '*** WARNING: Edge between vertices 1   &
               and 3 of triangle ',i,' is < ', TOL
26          chk_topology = .FALSE.
27          lcond = .FALSE.
28       end if
29       if(sum((triang(:,3,i)-triang(:,2,i))**2) .LT. TOL) then
30          WRITE(*,'(A,I6,A,E12.2)') '*** WARNING: Edge between vertices 2   &
               and 3 of triangle ',i,' is < ', TOL
31          chk_topology = .FALSE.
32          lcond = .FALSE.
33       end if
34    end do

35    if(lcond) then
36       WRITE(*,'(A)') 'Topology[A] Good: All edges are of length > 0.'
37    end if

      !------------------------------------------------------------------------------
      ! b) No vertex must be inside any other triangle
      !------------------------------------------------------------------------------

38    lcond = .TRUE.
      ! Main loop over all triangles
39    do i=1,M
40       do j=1,M                  ! loop over all the other triangles
41          if(j .NE. j) then      ! loop over all vertices of triangle i
42             do k=1,3
43                a = triang(:,2,j)-triang(:,1,j)
44                b = triang(:,3,j)-triang(:,1,j)
45                det = a(1)*b(2)-a(2)*b(1)
46                if(abs(det) .GE. TOL) then
47                   det = 1.0/det          ! 2x2 subsystem
48                   S(1,1)= b(2)*det
49                   S(1,2)=-b(1)*det
50                   S(2,1)=-a(2)*det
51                   S(2,2)= a(1)*det
52                   g(1:2)= triang(1:2,k,i)-triang(1:2,1,j)
                     ! Solution
53                   alpha = S(1,1)*g(1)+S(1,2)*g(2)
54                   beta  = S(2,1)*g(1)+S(2,2)*g(2)
                     ! point is in same plane if third equation is also
                     ! fullfilled with this solution
55                   if(alpha*a(3)+beta*b(3)+triang(3,1,j)-triang(3,k,i) .LE.  &
                        TOL) then
56                      if((alpha+beta .LT. 1.0+TOL) .AND. (alpha .GE. TOL)    &
                           .AND. (beta .GE. TOL)) then    ! point is insde triangle
57                         WRITE(*,'(A,I1,A,I6,A,I6)') '*** WARNING: Vertex ',k  &
                              ,' of triangle ',i,' lies inside triangle ',j
58                         chk_topology = .FALSE.
59                         lcond = .FALSE.
60                      end if
61    9              end if
62                end if
63             end do
64          end if
65       end do
66    end do

67    if(lcond) then
```

```
68       WRITE(*,'(A)') 'Topology[B] Good: No vertex is inside any other triangle.'
69    end if

      !------------------------------------------------------------------------------
      ! c) Every triangle must have exactly three neighbors sharing 2 vertices
      !------------------------------------------------------------------------------

70    lcond = .TRUE.
      ! Main loop over all triangles
71    do i=1,M
72       Neigh = 0            ! reset neighbor counter
73       do j=1,M             ! loop over all the other triangles
74          if(j .NE. i) then
75             Nvert = 0      ! reset number of common vertices
76             do k=1,3          ! loop over all vertices of triangle i
77                do l=1,3       ! loop over all different vertices of triangle j
78                   if(sum((triang(:,k,i)-triang(:,l,j))**2) .LE. TOL) then
79                      Nvert=Nvert+1    ! one common vertex found
80                   end if
81                end do
82             end do
83             if(Nvert .GT. 2) then
84                WRITE(*,'(A,I6,A,I6,A,I1,A)') '*** WARNING: Triangles ',i,  &
                     ' and ',j,' have ',Nvert,' vertices in common!'
85                chk_topology = .FALSE.
86                lcond = .FALSE.
87             end if
88             if(Nvert .EQ. 2) then    ! increment neighbor counter
89                Neigh=Neigh+1
90             end if
91          end if
92       end do
93       if(Neigh .NE. 3) then
94          WRITE(*,'(A,I6,A,I1,A)') '*** WARNING: Triangle ',i,' has '   &
               ,Neigh,' neighbors instead of 3.'
95          chk_topology = .FALSE.
96          lcond = .FALSE.
97       end if
98    end do

99    if(lcond) then
100      WRITE(*,'(A)') 'Topology[C] Good: All triangles have 3 neighbors.'
101   end if

      !------------------------------------------------------------------------------
      ! d) No edge of any triangle must intersect any other triangle
      !------------------------------------------------------------------------------

102   lcond = .TRUE.
      ! Main loop over all triangles
103   do i=1,M
104      a = triang(:,2,i)-triang(:,1,i)
105      b = triang(:,3,i)-triang(:,1,i)
106      g = triang(:,3,i)-triang(:,2,i)
107      do j=1,M
108         if(j .NE. i) then
109            if(intersect(triang(:,1,i), a, j) .GE. 0.0) then
110               WRITE(*,'(A,I6,A,I6)') '*** WARNING: Edge between vertices 1   &
                     and 2 of triangle ',i,' intersects with triangle ',j
111               chk_topology = .FALSE.
112               lcond = .FALSE.
113            end if
114            if(intersect(triang(:,1,i), b, j) .GE. 0.0) then
115               WRITE(*,'(A,I6,A,I6)') '*** WARNING: Edge between vertices 1   &
                     and 2 of triangle ',i,' intersects with triangle ',j
116               chk_topology = .FALSE.
117               lcond = .FALSE.
118            end if
119            if(intersect(triang(:,2,i), g, j) .GE. 0.0) then
120               WRITE(*,'(A,I6,A,I6)') '*** WARNING: Edge between vertices 1   &
                     and 2 of triangle ',i,' intersects with triangle ',j
121               chk_topology = .FALSE.
122               lcond = .FALSE.
123            end if
124         end if
125      end do
126   end do

127   if(lcond) then
128      WRITE(*,'(A)') 'Topology[D] Good: No edge intersects any other triangle.'
```

```
129   end if

130   return

131   END function chk_topology

      !------------------------------------------------------------------------------

132   function chk_orientation()

133   USE globals

134   IMPLICIT NONE

      !------------------------------------------------------------------------------
      !chk_orientation  Checks syntactical validity of a triangulation set
      !   CHK_ORIENTATION checks whether all normals point out of the domain.
      !   Needs centroids and normals to be calculated first!
      !   Assumes triangulation to be topologically valid.
      !
      !   RETURN VALUE: .TRUE. if set is syntactically valid, else .FALSE.
      !
      !   See also INIT_PART
      !
      !   todo:
      !
      !==============================================================================
      !  DIPLOMA THESIS WS01/02 ICOS                                    ETH-ZUERICH
      !------------------------------------------------------------------------------
      !
      !              PROTEIN DIFFUSION INSIDE THE ENDOPLASMIC RETICULUM
      !
      !============================ ivo f. sbalzarini ===============================
      !

      !------------------------------------------------------------------------------
      ! Declaration of external functions
      !------------------------------------------------------------------------------

      ! checks whether a point is inside the domain (needs topological validity !)
135   LOGICAL, EXTERNAL                         :: point_in_domain

      !------------------------------------------------------------------------------
      ! Declaration of input arguments and return value
      !------------------------------------------------------------------------------

      ! return value: .TRUE. if set is syntactically valid
136   LOGICAL                                   :: chk_orientation

      !------------------------------------------------------------------------------
      ! Declaration of local variables
      !------------------------------------------------------------------------------

      ! loop counters
137   INTEGER                                   :: i
      ! point in question
138   REAL(MK), DIMENSION(3)                    :: pt


139   chk_orientation = .TRUE.
      ! main loop over all triangles
140   do i=1,M
         ! follow the normal for a small distance (but larger than TOL
         ! for the difference to be detected by the comparison operator
141      pt = c(:,i)+5*TOL*normal(:,i)
142      if(point_in_domain(pt)) then
143         WRITE(*,'(A,I6,A)') '*** WARNING: Triangle ',i,' is not properly oriented.'
144         chk_orientation = .FALSE.
145      end if
146   end do

147   if(chk_orientation) then
148      WRITE(*,'(A)') 'Orientation Good: All normals point outward.'
149   end if

150   return

151   END function chk_orientation
```

C.5.4.8 Intersect a line with a triangle (intersect.f90)

`intersect` implements the algorithm given in section 3.2 to intersect a line $P + \lambda v$ with a triangle given by its index t in the current triangulation set. The function takes the two vectors $P \in \mathbb{R}^3$ and $v \in \mathbb{R}^3$ as well as the integer triangle index t as input arguments and returns the absolute value of the line parameter λ (`lambda`) at the point of intersection or -1 if no intersection point *inside* the triangle exists.

```
1     function intersect(P, v, t)

2     USE globals

3     IMPLICIT NONE

      !------------------------------------------------------------------------------
      !INTERSECT  Intersects line P+lambda*v with triangle t
      !   Returns the absolute value of lambda.
      !   If no intersection point inside triangle t exists, -1 is returned.
      !
      !   See also INIT_PART, CHK_TOPOLOGY, POINT_IN_DOMAIN
      !
      !==============================================================================
      !  DIPLOMA THESIS WS01/02 ICOS                                    ETH-ZUERICH
      !------------------------------------------------------------------------------
      !
      !              PROTEIN DIFFUSION INSIDE THE ENDOPLASMIC RETICULUM
      !
      !============================= ivo f. sbalzarini ==============================
      !

      !------------------------------------------------------------------------------
      ! Declaration of input and output variables
      !------------------------------------------------------------------------------

      ! Starting point of line
4     REAL(MK), DIMENSION(3), INTENT(IN) :: P
      ! direction vector
5     REAL(MK), DIMENSION(3), INTENT(IN) :: v
      ! triangle number
6     INTEGER, INTENT(IN)                :: t
      ! return value
7     REAL(MK)                           :: intersect

      !------------------------------------------------------------------------------
      ! Declaration of local variables
      !------------------------------------------------------------------------------

      ! line parameters
8     REAL(MK)                           :: alpha, beta, lambda
      ! edge vectors of triangle t
9     REAL(MK), DIMENSION(3)             :: a, b
      ! System matrix
10    REAL(MK), DIMENSION(3,3)           :: S
      ! determinant of system matrix and its inverse
11    REAL(MK)                           :: det, detinv

12    intersect = -1.0
      ! edge vectors
13    a = triang(:,2,t)-triang(:,1,t)
14    b = triang(:,3,t)-triang(:,1,t)
      ! determinant of system matrix
15    det = -a(3)*b(2)*v(1)+a(2)*b(3)*v(1)+a(3)*b(1)*v(2)-   &
           a(1)*b(3)*v(2)-a(2)*b(1)*v(3)+a(1)*b(2)*v(3)
16    if(abs(det) .GE. TOL) then      ! intersection point exists
         ! inverse of determinant
17       detinv=1.0/det
         ! inverse of system matrix
18       S(1,1)=(b(2)*v(3)-b(3)*v(2))*detinv
19       S(1,2)=(b(3)*v(1)-b(1)*v(3))*detinv
20       S(1,3)=(b(1)*v(2)-b(2)*v(1))*detinv
21       S(2,1)=(a(3)*v(2)-a(2)*v(3))*detinv
22       S(2,2)=(a(1)*v(3)-a(3)*v(1))*detinv
23       S(2,3)=(a(2)*v(1)-a(1)*v(2))*detinv
24       S(3,1)=(a(2)*b(3)-a(3)*b(2))*detinv
25       S(3,2)=(a(3)*b(1)-a(1)*b(3))*detinv
26       S(3,3)=(a(1)*b(2)-a(2)*b(1))*detinv
         ! solution of linear system
```

```
27       a(1:3)= P(1:3)-triang(:,1,t)
28       alpha = S(1,1)*a(1)+S(1,2)*a(2)+S(1,3)*a(3)
29       beta  = S(2,1)*a(1)+S(2,2)*a(2)+S(2,3)*a(3)
30       lambda= S(3,1)*a(1)+S(3,2)*a(2)+S(3,3)*a(3)
         ! check whether intersection point is inside triangle t
31       if((alpha+beta .LE. 1.0) .AND. (alpha .GE. 0)    &
              .AND. (beta .GE. 0)) then
32          intersect = abs(-lambda)
33       end if
34    end if

35    return

36    END function intersect
```

C.5.4.9 Create bin lists and cell lists (SortT.f90)

This subroutine sets up the bin lists and cell lists as described in section 7.2.5 and sorts the triangles into them. The data structures needed are explained in section 7.2.6.

```
1     subroutine SortT

2     USE globals
3     USE Util

4     IMPLICIT NONE

      !------------------------------------------------------------------------------
      !SORTT  Sorts triangles and builds lists.
      !   SortT sets up the bin lists for point_in_domain by assigning the
      !   triangles to bins of equal y and z coordinates as well as the cell
      !   lists for the nearest triangle search (if needed).
      !
      !   See also INIT_PART, POINT_IN_DOMAIN, ALLOCATELL
      !
      !==============================================================================
      !  DIPLOMA THESIS WS01/02 ICOS                                   ETH-ZUERICH
      !------------------------------------------------------------------------------
      !
      !              PROTEIN DIFFUSION INSIDE THE ENDOPLASMIC RETICULUM
      !
      !============================ ivo f. sbalzarini ===============================
      !

      !------------------------------------------------------------------------------
      ! Declaration of local variables
      !------------------------------------------------------------------------------

      ! loop counters
5     INTEGER                                 :: i, j, k, l, p, ix1, ix2
      ! minimum and maximum indices in x, y and z
6     INTEGER                                 :: minbinx, maxbinx, minbiny
7     INTEGER                                 :: maxbiny, minbinz, maxbinz
      ! x, y and z indices of current bin/cell
8     INTEGER                                 :: idxx, idxy, idxz
      ! lower and upper boundaries of bin
9     REAL(MK), DIMENSION(4)                  :: bb
      ! error trap
10    INTEGER                                 :: istat
      ! edge vectors of triangle projections, vertex 1
11    REAL(MK), DIMENSION(2)                  :: a, b, x1, y1, v1, v2
      ! three dimensional vectors for cell list
12    REAL(MK), DIMENSION(3)                  :: x13, a3, b3, n3, y13
      ! determinants
13    REAL(MK)                                :: det, det2
      ! line parameters
14    REAL(MK)                                :: alpha, beta, lambda
      ! system matrix
15    REAL(MK), DIMENSION(3,3)                :: mat
      ! lower and upper boundaries of cell and its centroid
16    REAL(MK), DIMENSION(3)                  :: bbl, bbu, bbc
      ! flag whether a certain triangle already belongs to a bin or not
17    LOGICAL                                 :: NotIn
      ! all three edge vectors of a triangle
18    REAL(MK), DIMENSION(2,3)                :: edge
```

```
      ! all edge vectors of a bin
19    REAL(MK), DIMENSION(2,2)                :: side

      !-------------------------------------------------------------------------------
      ! Initialize
      !-------------------------------------------------------------------------------

20    istat = 0
      ! initialize lists and arrays
21    CALL AllocateLL(0,0,0,0,istat)
22    if(istat .NE. 0) then
23       WRITE(*,'(A)') 'Error in AllocateLL: returning from SortT.'
24       return
25    end if
26    dby = (ymax-ymin)/real(NBINY)
27    dbz = (zmax-zmin)/real(NBINZ)

      !-------------------------------------------------------------------------------
      ! Assign triangles to linear bins for point_in_domain
      !-------------------------------------------------------------------------------

28    do i=1,M            ! loop over all triangles
29       minbiny = NBINY+1
30       maxbiny = 0
31       minbinz = NBINZ+1
32       maxbinz = 0
33       x1 = triang(2:3,1,i)
34       edge(:,1) = triang(2:3,2,i)-x1
35       edge(:,2) = triang(2:3,3,i)-x1
36       edge(:,3) = triang(2:3,3,i)-triang(2:3,2,i)
37       a = edge(:,1)
38       b = edge(:,2)
39       det = a(1)*b(2)-a(2)*b(1)
40       det = 1.0_MK/det
41       do j=1,3         ! loop over all vertices
            ! determine index of bin of this vertex
42          idxy = ceiling((triang(2,j,i)-ymin)/dby)
43          if(idxy .EQ. 0) idxy = 1             ! include boundaries at walls
44          if(idxy .EQ. NBINY+1) idxy = NBINY
45          idxz = ceiling((triang(3,j,i)-zmin)/dbz)
46          if(idxz .EQ. 0) idxz = 1             ! include boundaries at walls
47          if(idxz .EQ. NBINZ+1) idxz = NBINZ
            ! add this triangle to list of this bin if not already there
48          if(.NOT. ASSOCIATED(bin(idxy,idxz)%list)) then
49             CALL AllocateLL(1,0,idxy,idxz,istat)
50          end if
51          NotIn = .FALSE.
52          if(nbin(idxy,idxz) .EQ. 0) then
53             NotIn = .TRUE.
54          elseif(bin(idxy,idxz)%list(nbin(idxy,idxz)) .NE. i) then
55             NotIn = .TRUE.
56          end if
57          if(NotIn) then
58             p = nbin(idxy,idxz)+1
59             nbin(idxy,idxz)  = p
               ! enlarge this list if nedded
60             if(p .GT. size(bin(idxy,idxz)%list)) then
61                CALL AllocateLL(1,0,idxy,idxz,istat)
62             end if
63             bin(idxy,idxz)%list(p) = i
               ! update bounding box of cells
64             if(idxy .LT. minbiny) minbiny = idxy
65             if(idxy .GT. maxbiny) maxbiny = idxy
66             if(idxz .LT. minbinz) minbinz = idxz
67             if(idxz .GT. maxbinz) maxbinz = idxz
68          end if
69       end do
         ! loop over all bins in the bounding box of the vertices
70       do j=minbiny,maxbiny
71          bb(1) = ymin+(j-1)*dby   ! lower y
72          bb(2) = ymin+j*dby       ! upper y
73          do k=minbinz,maxbinz
               ! if no list for this bin is yet associated, create one
74             if(.NOT. ASSOCIATED(bin(j,k)%list)) then
75                CALL AllocateLL(1,0,j,k,istat)
76             end if
77             NotIn = .FALSE.
               ! if this triangle doesnt already belong to this bin, check it
78             if(nbin(j,k) .EQ. 0) then
79                NotIn = .TRUE.
```

```
80             elseif(bin(j,k)%list(nbin(j,k)) .NE. i) then
81                NotIn = .TRUE.
82             end if
83             if(NotIn) then
84                bb(3) = zmin+(k-1)*dbz   ! lower z
85                bb(4) = zmin+k*dbz       ! upper z
                  ! edge vectors of this bin (projected onto y/z plane)
                  ! only store two since the other two are identical
86                side(1,1) = bb(2)-bb(1)
87                side(2,1) = 0.0_MK
88                side(1,2) = 0.0_MK
89                side(2,2) = bb(4)-bb(3)
90                do ix1=1,2
91                   y1(1) = bb(ix1)-x1(1)
92                   do ix2=3,4
93                      y1(2) = bb(ix2)-x1(2)
94                      alpha = det*(b(2)*y1(1)-b(1)*y1(2))
95                      beta = det*(-a(2)*y1(1)+a(1)*y1(2))
96                      if((alpha .GE. 0.0_MK) .AND. (beta .GE. 0.0_MK)  &
                             .AND. (alpha+beta .LE. 1.0_MK)) then
                         ! point is inside triangle: add triangle to this bin
97                       p = nbin(j,k)+1
98                       nbin(j,k)  = p
                         ! enlarge this list if nedded
99                       if(p .GT. size(bin(j,k)%list)) then
100                         CALL AllocateLL(1,0,j,k,istat)
101                      end if
102                      bin(j,k)%list(p) = i
103                      goto 250           ! next bin
104                   end if
105                end do
106             end do
                ! if no vertex of the bin is inside the triangle, check
                ! whether any edge of the triangle intersects any edge of
                ! the bin
107             do ix1=1,3             ! loop over triangles edges
108                v1 = edge(:,ix1)
109                do ix2=1,4          ! loop over bins edges
                      ! choose adge vector mod 2 since bin is axis parallel
110                   v2 = -side(:,mod(ix2,2)+1)
                      ! system determinant (do not overwrite det !!)
111                   det2 = v1(1)*v2(2)-v1(2)*v2(2)
112                   det2 = 1.0_MK/det2
                      ! determine rhs (starting points of lines depending
                      ! on what edge we are at
113                   if(ix2 .EQ. 1) then
114                      y1 = (/bb(1), bb(3)/)
115                   elseif(ix2 .EQ. 2) then
116                      y1 = (/bb(1), bb(3)/)
117                   elseif(ix2 .EQ. 3) then
118                      y1 = (/bb(2), bb(3)/)
119                   elseif(ix2 .EQ. 4) then
120                      y1 = (/bb(1), bb(4)/)
121                   end if
122                   if(ix1 .EQ. 1 .OR. ix1 .EQ. 2) then
123                      y1 = y1 - x1
124                   else
125                      y1 = y1 - triang(2:3,2,i)
126                   end if
                      ! solve for intersection point
127                   alpha = det2*(v2(2)*y1(1)-v2(1)*y1(2))
128                   beta = det2*(-v1(2)*y1(1)+v1(1)*y1(2))
                      ! if intersected, add triangle to this bin
129                   if((alpha .GE. 0.0_MK) .AND. (beta .GE. 0.0_MK) .AND. &
                         (alpha .LE. 1.0_MK) .AND. (beta .LE. 1.0_MK)) then
                         ! point is inside triangle: add triangle to this bin
130                      p = nbin(j,k)+1
131                      nbin(j,k)  = p
                         ! enlarge this list if nedded
132                      if(p .GT. size(bin(j,k)%list)) then
133                         CALL AllocateLL(1,0,j,k,istat)
134                      end if
135                      bin(j,k)%list(p) = i
136                      goto 250           ! next bin
137                   end if
138                end do
139             end do
140   250    end if
141         end do
142      end do
```

```
143   end do

      !------------------------------------------------------------------------------
      ! prune bin lists to actual size to save memory
      !------------------------------------------------------------------------------

144   do i=1,NBINY
145      do j=1,NBINZ
146         bin(i,j)%list=>reallocate(bin(i,j)%list,nbin(i,j))
147      end do
148   end do

      !------------------------------------------------------------------------------
      ! Assign triangles to cell lists for nearest neighbor search
      !------------------------------------------------------------------------------

      ! cell lists only needed for boundary condition handling
149   if(BC .GT. 0) then
150      do i=1,M            ! loop over all triangles
151         minbinx = NCELLX+1
152         maxbinx = 0
153         minbiny = NCELLY+1
154         maxbiny = 0
155         minbinz = NCELLZ+1
156         maxbinz = 0
157         x13 = triang(1:3,1,i)
158         a3 = triang(1:3,2,i)-x13
159         b3 = triang(1:3,3,i)-x13
160         n3 = normal(1:3,i)
161         det = -a3(3)*b3(2)*n3(1)+a3(2)*b3(3)*n3(1)+a3(3)*b3(1)*n3(2)-  &
                  a3(1)*b3(3)*n3(2)-a3(2)*b3(1)*n3(3)+a3(1)*b3(2)*n3(3)
162         det = 1.0_MK/det
163         mat(1,1) = -b3(3)*n3(2)+b3(2)*n3(3)
164         mat(1,2) = b3(3)*n3(1)-b3(1)*n3(3)
165         mat(1,3) = -b3(2)*n3(1)+b3(1)*n3(2)
166         mat(2,1) = a3(3)*n3(2)-a3(2)*n3(3)
167         mat(2,2) = -a3(3)*n3(1)+a3(1)*n3(3)
168         mat(2,3) = a3(2)*n3(1)-a3(1)*n3(2)
169         mat(3,1) = -a3(3)*b3(2)+a3(2)*b3(3)
170         mat(3,2) = a3(3)*b3(1)-a3(1)*b3(3)
171         mat(3,3) = -a3(2)*b3(1)+a3(1)*b3(2)
172         do j=1,3          ! loop over all vertices
               ! determine index of cell of this vertex
173            idxx = ceiling((triang(1,j,i)-xmin)/dcx)
174            if(idxx .EQ. 0) idxx = 1               ! include boundaries at walls
175            if(idxx .EQ. NCELLX+1) idxx = NCELLX
176            idxy = ceiling((triang(2,j,i)-ymin)/dcy)
177            if(idxy .EQ. 0) idxy = 1               ! include boundaries at walls
178            if(idxy .EQ. NCELLY+1) idxy = NCELLY
179            idxz = ceiling((triang(3,j,i)-zmin)/dcz)
180            if(idxz .EQ. 0) idxz = 1               ! include boundaries at walls
181            if(idxz .EQ. NCELLZ+1) idxz = NCELLZ
182            if(.NOT. ASSOCIATED(cell(idxx,idxy,idxz)%list)) then
183               CALL AllocateLL(1,idxx,idxy,idxz,istat)
184            end if
185            NotIn = .FALSE.
186            if(ncell(idxx,idxy,idxz) .EQ. 0) then
187               NotIn = .TRUE.
188            elseif(cell(idxx,idxy,idxz)%list(ncell(idxx,idxy,idxz)) .NE. i) then
189               NotIn = .TRUE.
190            end if
191            if(NotIn) then
192               p = ncell(idxx,idxy,idxz)+1
193               ncell(idxx,idxy,idxz) = p
                  ! enlarge this list if nedded
194               if(p .GT. size(cell(idxx,idxy,idxz)%list)) then
195                  CALL AllocateLL(1,idxx,idxy,idxz,istat)
196               end if
197               cell(idxx,idxy,idxz)%list(p) = i
                  ! update bounding box of cells
198               if(idxx .LT. minbinx) minbinx = idxx
199               if(idxx .GT. maxbinx) maxbinx = idxx
200               if(idxy .LT. minbiny) minbiny = idxy
201               if(idxy .GT. maxbiny) maxbiny = idxy
202               if(idxz .LT. minbinz) minbinz = idxz
203               if(idxz .GT. maxbinz) maxbinz = idxz
204            end if
205         end do
           ! loop over all cells in the bounding box of the vertices
206         do j=minbinx,maxbinx
```

```
207            bbl(1) = xmin+(j-1)*dcx    ! lower x
208            bbu(1) = xmin+j*dcx        ! upper x
209            do k=minbiny,maxbiny
210               bbl(2) = ymin+(k-1)*dcy   ! lower y
211               bbu(2) = ymin+k*dcy       ! upper y
212               do l=minbinz,maxbinz
                     ! if no list for this cell is yet associated, create one
213                  if(.NOT. ASSOCIATED(cell(j,k,l)%list)) then
214                     CALL AllocateLL(1,j,k,l,istat)
215                  end if
216                  NotIn = .FALSE.
                     ! if this triangle doesnt already belong to this cell, check it
217                  if(ncell(j,k,l) .EQ. 0) then
218                     NotIn = .TRUE.
219                  elseif(cell(j,k,l)%list(ncell(j,k,l)) .NE. i) then
220                     NotIn = .TRUE.
221                  end if
222                  if(NotIn) then
223                     bbl(3) = zmin+(l-1)*dcz   ! lower z
224                     bbu(3) = zmin+l*dcz       ! upper z
225                     bbc(1:3) = 0.5*(bbl(1:3)+bbu(1:3))  ! voxels centroid
226                     y13(1:3) = bbc(1:3)-x13(1:3)
227                     alpha = det*(mat(1,1)*y13(1)+mat(1,2)*y13(2) +   &
                              mat(1,3)*y13(3))
228                     beta = det*(mat(2,1)*y13(1)+mat(2,2)*y13(2) +    &
                              mat(2,3)*y13(3))
229                     lambda = det*(mat(3,1)*y13(1)+mat(3,2)*y13(2) +   &
                              mat(3,3)*y13(3))
                        ! if intersection point is inside triangle
230                     if((alpha .GE. 0.0_MK) .AND. (beta .GE. 0.0_MK)  &
                              .AND. (alpha+beta .LE. 1.0_MK)) then
                           ! point where normal through centroid intersects triangle
231                        y13(1:3) = bbc(1:3)+lambda*n3(1:3)
                           ! if this point is inside the cell, add triangle to cell
232                        if((y13(1) .LT. bbu(1)) .AND. (y13(2) .LT. bbu(2)) .AND. &
                              (y13(3) .LT. bbu(3)) .AND. (y13(1) .GE. bbl(1)) .AND. &
                              (y13(2) .GE. bbl(2)) .AND. (y13(3) .GE. bbl(3))) then
233                           p = ncell(j,k,l)+1
234                           ncell(j,k,l) = p
                              ! enlarge this list if nedded
235                           if(p .GT. size(cell(j,k,l)%list)) then
236                              CALL AllocateLL(1,j,k,l,istat)
237                           end if
238                           cell(j,k,l)%list(p) = i
239                        end if
240                     end if
241                  end if
242               end do
243            end do
244         end do
245      end do

         !------------------------------------------------------------------------
         ! prune cell lists to actual size to save memory
         !------------------------------------------------------------------------

246      do i=1,NCELLX
247         do j=1,NCELLY
248            do k=1,NCELLZ
249               cell(i,j,k)%list=>reallocate(cell(i,j,k)%list,ncell(i,j,k))
250            end do
251         end do
252      end do
253   end if


254   END subroutine SortT
```

C.5.4.10 Allocate dynamic list memory (AllocateLL.f90)

`AllocateLL` is a helper routine for the handling of the bin and cell lists by `SortT` and the main program. It implements the whole memory management for the dynamic list data structures as described in section 7.2.6. The input argument `flag` determines the action to be taken. If the routine is called with `flag=0`, the initial memory allocation and general set-up of the lists is done. With `flag=1`, the list associated with cell `ix`, `iy`, `iz` (or bin `iy`, `iz` if `ix` is zero) is enlarged by a

certain number of elements defined by the parameter `STEP`. If called with `flag=2`, all lists are deallocated and the memory is freed again. The variable `info` is a status flag and contains 0 if everything was successful, 1 if an error occurred.

```
1     subroutine AllocateLL(flag, ix, iy, iz, info)

2     USE globals
3     USE Util

4     IMPLICIT NONE

      !------------------------------------------------------------------------------
      !ALLOCATELL  allocates memory for triangle lists.
      !   AllocateLL allocates memory for the bin and cell lists. If called with
      !   flag=0, initial allocation is done. Calling with flag=1 results in
      !   enlargement of existing arrays. Calling with flag=2 deallocates all
      !   memory. info returns 0 if allocation was successful, 1 otherwise.
      !
      !   See also INIT_PART, POINT_IN_DOMAIN, SORTT, UTIL
      !
      !==============================================================================
      !  DIPLOMA THESIS WS01/02 ICOS                                    ETH-ZUERICH
      !------------------------------------------------------------------------------
      !
      !              PROTEIN DIFFUSION INSIDE THE ENDOPLASMIC RETICULUM
      !
      !============================== ivo f. sbalzarini ==============================
      !

      ! enlarge arrays by STEP entries everytime allocatell(1,ix,iy,iz,info) is called
5     INTEGER, PARAMETER                      :: STEP = 10

      !------------------------------------------------------------------------------
      ! Declaration of input/output variables
      !------------------------------------------------------------------------------

      ! flag: 0=initialize, 1=reallocate, 2=deallocate
6     INTEGER, INTENT(IN)                     :: flag
      ! info: 0=everything OK, 1=error occured
7     INTEGER, INTENT(OUT)                    :: info
      ! x,y,z indices of cell/bin to (re)allocate list for
8     INTEGER, INTENT(IN)                     :: ix, iy, iz

      !------------------------------------------------------------------------------
      ! Declaration of local variables
      !------------------------------------------------------------------------------

      ! loop counters
9     INTEGER                                 :: i, j, k
      ! error status
10    INTEGER                                 :: istat
      ! length of list
11    INTEGER                                 :: llen

12    info = 0

      !------------------------------------------------------------------------------
      ! flag=0: Initialize
      !------------------------------------------------------------------------------

13    if(flag .EQ. 0) then

         !---------------------------------------------------------------------------
         ! Bin lists for point_in_domain (contain all triangles inside bin)
         !---------------------------------------------------------------------------

14       istat = 0
         ! allocate matrix to hold lengths of lists
         ! first index: biny, second index: binz
15       ALLOCATE(nbin(NBINY, NBINZ), STAT=istat)
16       if (istat .NE. 0) then
17          WRITE(*,'(2A)') 'Unable to allocate memory for nbin.'
18          info = 1
19          return
20       end if
21       nbin(1:NBINY,1:NBINZ) = 0
22       istat = 0
         ! allocate array of pointers to lists
```

```
        ! first index: biny, second index: binz
23      ALLOCATE(bin(NBINY, NBINZ), STAT=istat)
24      if (istat .NE. 0) then
25         WRITE(*,'(2A)') 'Unable to allocate memory for bin lists.'
26         info = 1
27         return
28      end if
        ! nullify all pointers (associate no lists at the beginning)
29      do i=1,NBINY
30         do j=1,NBINZ
31            nullify(bin(i,j)%list)
32         end do
33      end do

        !------------------------------------------------------------------------
        ! Cell lists for nearest tri. search (contain only triangles in in-sphere)
        !------------------------------------------------------------------------

        ! cell list for nearest triangle search is only needed if boundary
        ! condition handling is enabled
34      if(BC .GT. 0) then
           ! calculate number of cells in each direction based on cut-off radius
35         NCELLX = int((xmax-xmin)/rc)
36         NCELLY = int((ymax-ymin)/rc)
37         NCELLZ = int((zmax-zmin)/rc)
38         if(NCELLX .EQ. 0) NCELLX = 1    ! make at least one cell
39         if(NCELLX .EQ. 0) NCELLX = 1
40         if(NCELLX .EQ. 0) NCELLX = 1
           ! cell sizes
41         dcx = (xmax-xmin)/real(NCELLX)
42         dcy = (ymax-ymin)/real(NCELLY)
43         dcz = (zmax-zmin)/real(NCELLZ)

44         istat = 0
           ! allocate matrix to hold list lengths
           ! first index: cellx, second: celly, third: cellz
45         ALLOCATE(ncell(NCELLX,NCELLY,NCELLZ), STAT=istat)
46         if (istat .NE. 0) then
47            WRITE(*,'(2A)') 'Unable to allocate memory for ncell.'
48            info = 1
49            return
50         end if
51         ncell(1:NCELLX,1:NCELLY,1:NCELLZ) = 0
52         istat = 0
           ! allocate array of pointers to lists
           ! first index: cellx, second: celly, third: cellz
53         ALLOCATE(cell(NCELLX,NCELLY,NCELLZ), STAT=istat)
54         if (istat .NE. 0) then
55            WRITE(*,'(2A)') 'Unable to allocate memory for cell lists.'
56            info = 1
57            return
58         end if
           ! nullify all pointers in the beginning
59         do i=1,NCELLX
60            do j=1,NCELLY
61               do k=1,NCELLZ
62                  nullify(cell(i,j,k)%list)
63               end do
64            end do
65         end do
66      end if

     !------------------------------------------------------------------------
     ! flag=1: Enlarge lists
     !------------------------------------------------------------------------

67   elseif(flag .EQ. 1) then
68      if(ix .EQ. 0) then
           ! no x index given: operation affects bin list only
69         istat = 0
70         if(ASSOCIATED(bin(iy,iz)%list)) then
              ! if list is already assigned, reallocate it
71            llen = size(bin(iy,iz)%list)
72            bin(iy,iz)%list=>reallocate(bin(iy,iz)%list,llen+STEP)
73         else
              ! if there is no list yet, allocate one
74            ALLOCATE(bin(iy,iz)%list(STEP), STAT=istat)
75            if(istat .NE. 0) then
76               WRITE(*,'(A,I5,A,I5,A)') 'Error reallocating bin list  &
                    (',iy,',',iz,')'
```

```
77                info = 1
78                return
79             end if
80          end if
81       else
            ! x index given: operation affects cell list only
82          if(BC .GT. 0) then
83             istat = 0
84             if(ASSOCIATED(cell(ix,iy,iz)%list)) then
85                llen = size(cell(ix,iy,iz)%list)
86                cell(ix,iy,iz)%list=>reallocate(cell(ix,iy,iz)%list,llen+STEP)
87             else
88                ALLOCATE(cell(ix,iy,iz)%list(STEP), STAT=istat)
89                if(istat .NE. 0) then
90                   WRITE(*,'(A,I5,A,I5,A,I5,A)') 'Error reallocating cell   &
                         list(',ix,',',iy,',',iz,')'
91                   info = 1
92                   return
93                end if
94             end if
95          end if
96       end if

      !------------------------------------------------------------------------
      ! flag=2: Deallocate all lists
      !------------------------------------------------------------------------

97    elseif(flag .EQ. 2) then
         ! bin lists: always
98       do i=1,NBINY
99          do j=1,NBINZ
100            if(ALLOCATED(bin(i,j)%list)) DEALLOCATE(bin(i,j)%list)
101         end do
102      end do
103      if(ALLOCATED(bin)) DEALLOCATE(bin)
104      if(ALLOCATED(nbin)) DEALLOCATE(nbin)

         ! cell lists: only if boundary condition handling is active
105      if(BC .GT. 0) then
106         do i=1,NCELLX
107            do j=1,NCELLY
108               do k=1,NCELLZ
109                  if(ALLOCATED(cell(i,j,k)%list)) DEALLOCATE(cell(i,j,k)%list)
110               end do
111            end do
112         end do
113         if(ALLOCATED(cell)) DEALLOCATE(cell)
114         if(ALLOCATED(ncell)) DEALLOCATE(ncell)
115      end if
116   endif

117   return

118   END subroutine AllocateLL
```

C.5.4.11 Determine if point is in domain (point_in_domain.f90)

The next function determines whether the point given by its input argument $p \in \mathbb{R}^3$ is inside or outside the closed surface described by the triangulated set. If the point is inside, the value 1 is returned, 0 if it is outside. If boundary condition handling is switched on (i.e. the parameter `boundarycondition` in the input file is > 0), a value of 2 is returned if the point is inside the domain *and* closer than `cutoff` to its boundary. These are the points that need to be mirrored to the outside. In order to be able to do so, the function in this case also returns the index of the closest triangle in `closest` and the orthogonal distance of the point from this triangle in `lammin`. Bin and cell lists are used to accelerate the regular triangle handling and the search for the closest triangle, respectively.

If the parameter `debug` is set to `.TRUE.`, a set of additional files is created every time the function is invoked. The file `bin.deb` contains the edges of the current bin to be visualized with `gnuplot`. The file `tlist.deb` contains the vertices of all the triangles that belong to this bin, `triang.deb` contains the vertices of the

triangles that actually have been intersected by the line $P + \lambda[1, 0, 0]$, `points.deb` contains the coordinates of all the intersection points and `coeff.deb` contains the distances from point P to all the intersection points with the triangles. All files can be visualized using `gnuplot` and the `sp` command. Figure 7.3 has for example been created this way.

```
1     function point_in_domain(p, lammin, closest)

2     USE globals

3     IMPLICIT NONE

      !------------------------------------------------------------------------------
      !POINT_IN_DOMAIN  checks whether a point is inside the domain or not.
      !   if the point given by p(1:3) is inside the closed surface
      !   defined by triang(...), the function returns 1, otherwise
      !   0. If the point is inside AND closer to the boundary than rc,
      !   the value 2 is returned. The return valiable lammin contains the
      !   orthogonal distance to the closest triangle and closest is its index.
      !   The latter two are only calculated if BC is not 0.
      !
      !   See also INIT_PART, SORTT
      !
      !==============================================================================
      !  DIPLOMA THESIS WS01/02 ICOS                                   ETH-ZUERICH
      !------------------------------------------------------------------------------
      !
      !              PROTEIN DIFFUSION INSIDE THE ENDOPLASMIC RETICULUM
      !
      !============================ ivo f. sbalzarini ===============================
      !

      !------------------------------------------------------------------------------
      ! Declaration of external functions
      !------------------------------------------------------------------------------

      ! intersects a triangle with a given line
4     REAL(MK), EXTERNAL                  :: intersect

      !------------------------------------------------------------------------------
      ! Declaration of input and output variables
      !------------------------------------------------------------------------------

      ! point in question
5     REAL(MK), DIMENSION(3), INTENT(IN)  :: p
      ! return value
6     INTEGER                             :: point_in_domain
      ! distance to closest triangle
7     REAL(MK), INTENT(OUT)               :: lammin
      ! index of closest triangle
8     INTEGER, INTENT(OUT)                :: closest

      !------------------------------------------------------------------------------
      ! Declaration of local variables
      !------------------------------------------------------------------------------

      ! loop counter
9     INTEGER                             :: i, j, k, l, tri
      ! angles in spherical coordinates
10    REAL(MK)                            :: phi, theta
      ! edge vectors and direction
11    REAL(MK), DIMENSION(3)              :: a, b
      ! system matrix
12    REAL(MK), DIMENSION(3,3)            :: mat
      ! number of intersection points
13    INTEGER                             :: nip
      ! line parameter lambda
14    REAL(MK)                            :: lam
      ! x, y and z indices of current bin/cell
15    INTEGER                             :: idxx, idxy, idxz
      ! produce debuging output ?
16    LOGICAL                             :: debug

      !------------------------------------------------------------------------------
      ! Follow x direction and intersect with all triangles
      !------------------------------------------------------------------------------
```

```
17    debug = .false.
      !if(p(1) .gt. 400.0_MK .and. p(1) .lt. 402.0_MK .and. &
      !   p(2) .gt. 386.0_MK .and. p(2) .lt. 388.0_MK .and. &
      !   p(3) .gt. 5.0_MK .and. p(3) .lt. 5.5_MK) then
      !   debug = .true.
      !   print*,'point is: ',p
      !end if

18    nip = 0
19    mat(1,1) = 0.0
20    mat(2,1) = 0.0
21    idxy = ceiling((p(2)-ymin)/dby)
22    if(idxy .EQ. 0) idxy = 1             ! account for boundary points
23    if(idxy .EQ. NBINY+1) idxy = NBINY
24    idxz = ceiling((p(3)-zmin)/dbz)
25    if(idxz .EQ. 0) idxz = 1             ! account for boundary points
26    if(idxz .EQ. NBINZ+1) idxz = NBINZ
27    if(debug) then
28       print*,'bin indices: ',idxy,idxz
29       print*,'nbin=',nbin(idxy,idxz)
30       open(70,file='bin.deb',status='replace')
31       WRITE(70,*) xmin, ymin+(idxy-1)*dby, zmin+(idxz-1)*dbz
32       WRITE(70,*) xmin, ymin+(idxy)*dby, zmin+(idxz-1)*dbz
33       WRITE(70,*) xmin, ymin+(idxy)*dby, zmin+(idxz)*dbz
34       WRITE(70,*) xmin, ymin+(idxy-1)*dby, zmin+(idxz)*dbz
35       WRITE(70,*) xmin, ymin+(idxy-1)*dby, zmin+(idxz-1)*dbz
36       WRITE(70,'(A)')
37       WRITE(70,'(A)')
38       WRITE(70,*) xmax, ymin+(idxy-1)*dby, zmin+(idxz-1)*dbz
39       WRITE(70,*) xmax, ymin+(idxy)*dby, zmin+(idxz-1)*dbz
40       WRITE(70,*) xmax, ymin+(idxy)*dby, zmin+(idxz)*dbz
41       WRITE(70,*) xmax, ymin+(idxy-1)*dby, zmin+(idxz)*dbz
42       WRITE(70,*) xmax, ymin+(idxy-1)*dby, zmin+(idxz-1)*dbz
43       WRITE(70,'(A)')
44       WRITE(70,'(A)')
45       WRITE(70,*) xmin, ymin+(idxy-1)*dby, zmin+(idxz-1)*dbz
46       WRITE(70,*) xmax, ymin+(idxy-1)*dby, zmin+(idxz-1)*dbz
47       WRITE(70,'(A)')
48       WRITE(70,'(A)')
49       WRITE(70,*) xmin, ymin+(idxy)*dby, zmin+(idxz-1)*dbz
50       WRITE(70,*) xmax, ymin+(idxy)*dby, zmin+(idxz-1)*dbz
51       WRITE(70,'(A)')
52       WRITE(70,'(A)')
53       WRITE(70,*) xmin, ymin+(idxy)*dby, zmin+(idxz)*dbz
54       WRITE(70,*) xmax, ymin+(idxy)*dby, zmin+(idxz)*dbz
55       WRITE(70,'(A)')
56       WRITE(70,'(A)')
57       WRITE(70,*) xmin, ymin+(idxy-1)*dby, zmin+(idxz)*dbz
58       WRITE(70,*) xmax, ymin+(idxy-1)*dby, zmin+(idxz)*dbz
59       CLOSE(70)
60    end if

      ! intersect with all triangles in the same bin
61    do j=1,nbin(idxy,idxz)
         ! get index of next triangle
62       i = bin(idxy,idxz)%list(j)
63       if(debug) then
64          open(70,file='tlist.deb',position='append')
65          WRITE(70,*) triang(:,1,i)
66          WRITE(70,*) triang(:,2,i)
67          WRITE(70,*) triang(:,3,i)
68          WRITE(70,*) triang(:,1,i)
69          WRITE(70,'(A)')
70          WRITE(70,'(A)')
71          CLOSE(70)
72       end if
         ! edge vectors
73       a = triang(:,2,i)-triang(:,1,i)
74       b = triang(:,3,i)-triang(:,1,i)
         ! determinant of system matrix
75       phi = -a(3)*b(2)+a(2)*b(3)          ! for v=(1,0,0)
76       if (abs(phi) .GE. TOL) then         ! if intersection point exists
77          phi = 1.0/phi
            ! inverse of system matrix
78          mat(1,2) = phi*b(3)
79          mat(1,3) = -phi*b(2)
80          mat(2,2) = -phi*a(3)
81          mat(2,3) = phi*a(2)
82          mat(3,1) = phi*(-a(3)*b(2)+a(2)*b(3))
83          mat(3,2) = phi*(a(3)*b(1)-a(1)*b(3))
```

```
84          mat(3,3) = phi*(-a(2)*b(1)+a(1)*b(2))
            ! right hand side
85          a = p-triang(:,1,i)
            ! solution of 3x3 system
86          b = (mat(:,1)*a(1))+(mat(:,2)*a(2))+(mat(:,3)*a(3))
            ! intersect inside triangle ?
87          if ((b(1)+b(2) .LE. 1.0_MK) .AND. (b(1) .GE. 0.0_MK)   &
                 .AND. (b(2) .GE. TOL) .AND. (b(3) .GE. TOL)) then
88             if(debug) then
89                print*,'intersected triangle ',i
90                OPEN(70,file='triang.deb',position='append')
91                WRITE(70,*) triang(:,1,i)
92                WRITE(70,*) triang(:,2,i)
93                WRITE(70,*) triang(:,3,i)
94                WRITE(70,*) triang(:,1,i)
95                WRITE(70,'(A)')
96                WRITE(70,'(A)')
97                CLOSE(70)
98                open(70,file='points.deb',position='append')
99                write(70,*) triang(:,1,i)+b(1)*(triang(:,2,i)-triang(:,1,i)) &
                       +b(2)*(triang(:,3,i)-triang(:,1,i))
100               close(70)
101               open(70,file='coeff.deb',position='append')
102               write(70,*) b
103               close(70)
104            end if
105            nip=nip+1
106         end if
107      end if
108   end do

      !------------------------------------------------------------------------------
      ! Odd or even number of intersections ?
      !------------------------------------------------------------------------------

109   if(mod(nip,2) .EQ. 0.0) then
110      point_in_domain = 0
111   else
112      point_in_domain = 1
113   end if
114   if(debug) print*,'point_in_domain=',point_in_domain

      !------------------------------------------------------------------------------
      ! For the points inside: check if they are close to the boundary
      ! and find the nearest triangle as well as the distance to it
      !------------------------------------------------------------------------------

115   if(BC .GT. 0) then             ! only do this if needed for the B.C.
116      if(point_in_domain .EQ. 1) then
117         lammin = HUGE(lam)
118         closest = 0
            ! determine indices of cell we are in
119         idxx = ceiling((p(1)-xmin)/dcx)
120         if(idxx .EQ. 0) idxx = 1             ! account for boundary points
121         if(idxx .EQ. NCELLX+1) idxx = NCELLX
122         idxy = ceiling((p(2)-ymin)/dcy)
123         if(idxy .EQ. 0) idxy = 1             ! account for boundary points
124         if(idxy .EQ. NCELLY+1) idxy = NCELLY
125         idxz = ceiling((p(3)-zmin)/dcz)
126         if(idxz .EQ. 0) idxz = 1             ! account for boundary points
127         if(idxz .EQ. NCELLZ+1) idxz = NCELLZ
            ! loop over all triangles in this cell and its nearest neighbors
128         do i=idxx-1,idxx+1
129            do j=idxy-1,idxy+1
130               do k=idxz-1,idxz+1
                     ! if this is a valid cell ...
131                  if((i .GT. 0 .AND. j .GT. 0 .AND. k .GT. 0) .AND.  &
                        (i .LE. NCELLX .AND. j .LE. NCELLY .AND. k .LE. NCELLZ)) then
                        ! ... loop over all triangles in this cell
132                     do l=1,ncell(i,j,k)
                           ! get nextr triangle from the cell list
133                        tri = cell(i,j,k)%list(l)
                           ! intersect normal through point with triangle
134                        lam = intersect(p, normal(1:3,tri), tri)
135                        if(lam .GE. 0) then   ! if intersection pt. exists
136                           if(lam .LE. lammin) then
                                 ! if this is closer than anything before
137                              lammin = lam
138                              closest = tri ! => update data
139                           end if
```

```
140                        end if
141                     end do
                        ! if point is closer to surface than rc, set return
                        ! value to 2. NOTE: normal(:,:) are unit vectors
142                     if(lammin .LE. rc) point_in_domain = 2
143                  end if
144               end do
145            end do
146         end do
147      end if
148   end if

149   return

150   end function point_in_domain
```

C.5.4.12 Create voxel representation (Voxelize.f90)

This routine converts the triangulated surface to a voxel representation according to algorithm 5.1. The geometry is covered by voxels of extension `vx`×`vy`×`vz`. `resx`, `resy` and `resz` are output variables containing the actual number of voxels that have been generated in each spatial direction. `info` finally is a status flag which contains 0 on success, 1 otherwise. The binary voxel representation where all the voxels that are intersected by the surface triangulation are set to 1, all others to zero is contained in the global array `INTEGER voxel(resx,resy,resz)` (see section C.5.4.1) after the call to this routine.

```
1     subroutine Voxelize(vx,vy,vz,resx,resy,resz,info)

2     USE globals

3     IMPLICIT NONE

      !------------------------------------------------------------------------------
      !Voxelize  converts the surface to a 3D voxel representation.
      !   VOXELIZE finds a voxel representation of the surface
      !   using voxels of size vx*vy*vz. The representation is contained in
      !   the global binary 3D array voxel (int). resx, resy, resz return the
      !   actual number of voxels used in it. info returns 0 on success, else 1
      !
      !   See also INIT_PART, BCDIM
      !
      !==============================================================================
      !  DIPLOMA THESIS WS01/02 ICOS                                    ETH-ZUERICH
      !------------------------------------------------------------------------------
      !
      !              PROTEIN DIFFUSION INSIDE THE ENDOPLASMIC RETICULUM
      !
      !============================== ivo f. sbalzarini ==============================
      !

      !------------------------------------------------------------------------------
      ! Declaration of input/output variables
      !------------------------------------------------------------------------------

      ! desired voxel size (in x, y, z)
4     REAL(MK), INTENT(INOUT)                  :: vx, vy, vz
      ! number of voxels in x,y,z
5     INTEGER, INTENT(OUT)                     :: resx, resy, resz
      ! error varaible
6     INTEGER, INTENT(OUT)                     :: info

      !------------------------------------------------------------------------------
      ! Declaration of local variables
      !------------------------------------------------------------------------------

      ! loop counters
7     INTEGER                                  :: i, j, k, l, ix1, ix2, ix3
      ! minimum/maximum voxel indices in x,y,z of a given triangle
8     INTEGER                                  :: ixmin, ixmax, iymin, iymax
9     INTEGER                                  :: izmin, izmax
      ! indices of voxel
10    INTEGER                                  :: ixx, ixy, ixz
```

```
      ! vertex one of triangle, right hand side of eqs
11    REAL(MK), DIMENSION(3)                  :: x1, y1
      ! triangle edge vectors and normal
12    REAL(MK), DIMENSION(3)                  :: a, b, no
      ! lower and upper boundaries of voxel and its centroid
13    REAL(MK), DIMENSION(3)                  :: bbl, bbu, bbc
      ! error trap
14    INTEGER                                 :: istat
      ! system matrix
15    REAL(MK), DIMENSION(3,3)                :: mat
      ! determinant
16    REAL(MK)                                :: det
      ! system solution
17    REAL(MK)                                :: alpha, beta, lambda

      !-----------------------------------------------------------------------------
      ! Initialize
      !-----------------------------------------------------------------------------

18    info = 0
19    resx = ceiling((xmax-xmin)/vx)
20    resy = ceiling((ymax-ymin)/vy)
21    resz = ceiling((zmax-zmin)/vz)
22    if(resx .EQ. 0) resx = 1
23    if(resy .EQ. 0) resy = 1
24    if(resz .EQ. 0) resz = 1
25    istat = 0
      ! initialize array
26    if(ALLOCATED(voxel)) DEALLOCATE(voxel)
27    ALLOCATE(voxel(resx,resy,resz), STAT=istat)
28    if(istat .NE. 0) then
29       WRITE(*,'(A)') 'Error in Voxelize: cannot allocate voxel array'
30       info = 1
31       return
32    end if
33    voxel(1:resx,1:resy,1:resz) = 0
34    vx = (xmax-xmin)/real(resx)
35    vy = (ymax-ymin)/real(resy)
36    vz = (zmax-zmin)/real(resz)

      !-----------------------------------------------------------------------------
      ! Discretize geometry onto voxels
      !-----------------------------------------------------------------------------

37    do i=1,M            ! loop over all triangles
38       ixmin = resx+1   ! reset bounding box indices
39       ixmax = 0
40       iymin = resy+1
41       iymax = 0
42       izmin = resz+1
43       izmax = 0
44       x1 = triang(1:3,1,i)      ! vertex one of this triangle
45       a = triang(1:3,2,i)-x1    ! edge vectors
46       b = triang(1:3,3,i)-x1
47       no = normal(1:3,i)        ! triangles normal
48       det = -a(3)*b(2)*no(1)+a(2)*b(3)*no(1)+a(3)*b(1)*no(2)-  &
              a(1)*b(3)*no(2)-a(2)*b(1)*no(3)+a(1)*b(2)*no(3)
49       det = 1.0_MK/det
50       mat(1,1) = -b(3)*no(2)+b(2)*no(3)         ! system matrix (see notes)
51       mat(1,2) = b(3)*no(1)-b(1)*no(3)
52       mat(1,3) = -b(2)*no(1)+b(1)*no(2)
53       mat(2,1) = a(3)*no(2)-a(2)*no(3)
54       mat(2,2) = -a(3)*no(1)+a(1)*no(3)
55       mat(2,3) = a(2)*no(1)-a(1)*no(2)
56       mat(3,1) = -a(3)*b(2)+a(2)*b(3)
57       mat(3,2) = a(3)*b(1)-a(1)*b(3)
58       mat(3,3) = -a(2)*b(1)+a(1)*b(2)
59       do j=1,3         ! loop over all vertices
            ! determine index of voxel of this vertex
60          ixx = ceiling((triang(1,j,i)-xmin)/vx)
61          if(ixx .EQ. 0) ixx = 1               ! include boundaries at walls
62          if(ixx .EQ. resx+1) ixx = resx
63          ixy = ceiling((triang(2,j,i)-ymin)/vy)
64          if(ixy .EQ. 0) ixy = 1               ! include boundaries at walls
65          if(ixy .EQ. resy+1) ixy = resy
66          ixz = ceiling((triang(3,j,i)-zmin)/vz)
67          if(ixz .EQ. 0) ixz = 1               ! include boundaries at walls
68          if(ixz .EQ. resz+1) ixz = resz
            ! set this voxel to one
69          voxel(ixx,ixy,ixz) = 1
```

```
           ! update bounding box of voxels
70         if(ixx .LT. ixmin) ixmin = ixx
71         if(ixx .GT. ixmax) ixmax = ixx
72         if(ixy .LT. iymin) iymin = ixy
73         if(ixy .GT. iymax) iymax = ixy
74         if(ixz .LT. izmin) izmin = ixz
75         if(ixz .GT. izmax) izmax = ixz
76       end do
         ! loop over all voxels in the bounding box of the vertices
77       do j=ixmin,ixmax
78          bbl(1) = xmin+(j-1)*vx    ! lower x
79          bbu(1) = xmin+j*vx        ! upper x
80          do k=iymin,iymax
81             bbl(2) = ymin+(k-1)*vy   ! lower y
82             bbu(2) = ymin+k*vy       ! upper y
83             do l=izmin,izmax
84                bbl(3) = zmin+(l-1)*vz   ! lower z
85                bbu(3) = zmin+l*vz       ! upper z
86                bbc(1:3) = 0.5*(bbl(1:3)+bbu(1:3))  ! voxels centroid
87                y1(1:3) = bbc(1:3)-x1(1:3)
88                alpha = det*(mat(1,1)*y1(1)+mat(1,2)*y1(2) +   &
                       mat(1,3)*y1(3))
89                beta = det*(mat(2,1)*y1(1)+mat(2,2)*y1(2) +    &
                       mat(2,3)*y1(3))
90                lambda = det*(mat(3,1)*y1(1)+mat(3,2)*y1(2) +   &
                       mat(3,3)*y1(3))
91                y1(1:3) = bbc(1:3)+lambda*no(1:3)
                  ! if centroid is over triangle and intersection point
                  ! with triangle is inside voxel, set it to one
92                if ((alpha .GE. 0.0_MK) .AND. (beta .GE. 0.0_MK)  &
                       .AND. (alpha+beta .LE. 1.0)) then
93                   if(y1(1) .GE. bbl(1) .AND. y1(1) .LE. bbu(1) .AND.   &
                         y1(2) .GE. bbl(2) .AND. y1(2) .LE. bbu(2) .AND.   &
                         y1(3) .GE. bbl(3) .AND. y1(3) .LE. bbu(3)) voxel(j,k,l) = 1
94                end if
95             end do
96          end do
97       end do
98    end do

99    END subroutine Voxelize
```

C.5.4.13 Determine box counting dimension (BCdim.f90)

`BCdim` calculates the box counting dimension of the geometry according to algorithm 5.2. The geometry is first converted to a voxel representation of initial voxel size `vx0`×`vy0`×`vz0` using the previous routine. The input parameter `thresh` contains the threshold value for the binarization step as described in section 5.2. `info` is a status flag that contains 0 if the function returns without errors and 1 otherwise. The return value of the function `BCdim` is the least squares estimation of the box counting dimension.

```
1     function BCdim(vx0, vy0, vz0, thresh, info)

2     USE globals

3     IMPLICIT NONE

      !------------------------------------------------------------------------------
      !BCdim  Calculates the box counting dimension of a triangulated shape.
      !   BCDIM first translates the triangulated surface to a voxel
      !   representation with voxels of size vx0*vy0*vz0 and then performs successive
      !   resolution lowering. Voxels with a value lower than thresh are removed
      !   after the low-pass filter. The box counting dimension is estimated as a
      !   linear least-squares fit trough all the measurement points.
      !   info returns 0 on success, else 1.
      !
      !   See also INIT_PART, VOXELIZE
      !
      !==============================================================================
      !  DIPLOMA THESIS WS01/02 ICOS                                  ETH-ZUERICH
      !------------------------------------------------------------------------------
      !
      !              PROTEIN DIFFUSION INSIDE THE ENDOPLASMIC RETICULUM
```

```
      !
      !============================ ivo f. sbalzarini ==============================
      !

      ! Factor by which the box size is increased every reduction step
4     REAL(MK), PARAMETER                     :: fact = 2.0_MK

      !------------------------------------------------------------------------------
      ! Declaration of input/output variables
      !------------------------------------------------------------------------------

      ! box counting dimension
5     REAL(MK)                                :: BCdim
      ! initial voxel size
6     REAL(MK), INTENT(IN)                    :: vx0, vy0, vz0
      ! threshold for voxel values
7     REAL(MK), INTENT(IN)                    :: thresh
      ! error varaible
8     INTEGER, INTENT(OUT)                    :: info

      !------------------------------------------------------------------------------
      ! Declaration of local variables
      !------------------------------------------------------------------------------

      ! loop counters
9     INTEGER                                 :: redstep, i, j, k, isx, isy, isz
      ! index of current voxel
10    INTEGER                                 :: idx, idy, idz
      ! indices of voxel in reduced data set
11    INTEGER                                 :: ired, jred, kred
      ! number of voxels in x,y,z
12    INTEGER                                 :: nvx, nvy, nvz, npix, nones
      ! new number of voxels after reduction step
13    INTEGER                                 :: nvxred, nvyred, nvzred
      ! number of reduction steps to perform
14    INTEGER                                 :: nred
      ! error trap
15    INTEGER                                 :: istat
      ! Array to hold the voxel counts after each reduction step and its log
16    INTEGER, DIMENSION(:), ALLOCATABLE      :: Nvox
17    REAL(MK), DIMENSION(:), ALLOCATABLE     :: logN
      ! array to hold the box size of each step
18    REAL(MK), DIMENSION(:), ALLOCATABLE     :: s
      ! accumulated voxel values for low-pass average and their total number
19    REAL(MK)                                :: val
20    INTEGER                                 :: nval
      ! temporary voxel array for resolution reduction
21    INTEGER, DIMENSION(:,:,:), ALLOCATABLE  :: voxtmp
      ! sum of all reduction steps and sum of their squares
22    REAL(MK)                                :: sumi, sumi2
      ! determinant and its inverse
23    REAL(MK)                                :: det, detinv
      ! slope and y-axis intersect of linear fit
24    REAL(MK)                                :: a, b
25    LOGICAL, DIMENSION(:,:,:), ALLOCATABLE  :: flag
      ! voxel size at current reduction step
26    REAL(MK)                                :: vx, vy, vz

      !------------------------------------------------------------------------------
      ! Initialize and allocate memory
      !------------------------------------------------------------------------------

27    if(thresh .GT. 1.0_MK .OR. thresh .LT. 0.0_MK) then
28       WRITE(*,'(A)') 'Threshold for BCdim must be between 0 and 1'
29       info = 1
30       return
31    end if

32    info = 0
33    BCdim = 0.0_MK
34    vx = vx0
35    vy = vy0
36    vz = vz0
      ! convert surface to voxel representation
37    CALL Voxelize(vx0,vy0,vz0,nvx,nvy,nvz,info)
38    if(info .NE. 0) then
39       WRITE(*,'(A)') 'Error in Voxelize called from BCdim.'
40       return
41    end if
      ! reduce until the smallest dimension contains less than 16 voxels
```

```
42    nred = ceiling(log(real(minval((/nvx,nvy,nvz/)))/16.0_MK)/log(fact))

43    istat = 0
44    ALLOCATE(Nvox(nred+1), s(nred+1), logN(nred+1), STAT=istat)
45    if(istat .NE. 0) then
46       WRITE(*,'(A)') 'Error allocating memory for Nvox in BCdim'
47       info = 1
48       return
49    end if

      !------------------------------------------------------------------------
      ! Box counting algorithm by successive reduction of resolution
      !------------------------------------------------------------------------

50    do redstep=1,nred
51       ALLOCATE(flag(nvx,nvy,nvz))
52       flag = .FALSE.
         ! outline voxel set: only keep outer layer of voxels
53       do i=1,nvx                     ! loop over all voxels
54          do j=1,nvy
55             do k=1,nvz
56                if(voxel(i,j,k) .NE. 0) then    ! if a voxel is 1, count
                                                  ! the number of its
57                   npix = 0                     ! neighbors that are 1 too
58                   nones = 0
59                   do isx=-1,1
60                      idx = i+isx
61                      do isy=-1,1
62                         idy = j+isy
63                         do isz=-1,1
64                            idz = k+isz
65                            if((idx .GT. 0 .AND. idx .LE. nvx) .AND.   &
                              (idy .GT. 0 .AND. idy .LE. nvy) .AND.   &
                              (idz .GT. 0 .AND. idz .LE. nvz)) then
                                 ! if inside array: check it
66                               npix = npix + 1
67                               if(voxel(idx,idy,idz) .EQ. 1) nones=nones+1
68                            end if
69                         end do
70                      end do
71                   end do
                     ! if all its neighbors are one, mark it for deletion
                     ! since it is not a border voxel
72                   if(nones .EQ. npix) flag(i,j,k) = .TRUE.
73                end if
74             end do
75          end do
76       end do
         ! delete all interior voxels
77       where(flag) voxel=0
78       DEALLOCATE(flag)

79       Nvox(redstep) = sum(voxel)
80       s(redstep) = vx
81       vx = fact*vx
82       vy = fact*vy
83       vz = fact*vz

84       WRITE(*,'(A,I2,A,I2,A,I7)') 'BCdim: reduction step ',redstep,' of  &
            ',nred,'.   voxels: ',Nvox(redstep)
         ! new voxel numbers after reduction step
85       nvxred = ceiling((1.0_MK/fact)*real(nvx))
86       nvyred = ceiling((1.0_MK/fact)*real(nvy))
87       nvzred = ceiling((1.0_MK/fact)*real(nvz))
         ! memory for reduced voxel array
88       ALLOCATE(voxtmp(nvxred,nvyred,nvzred), STAT=istat)
89       if(istat .NE. 0) then
90          WRITE(*,'(A)') 'Error allocating temporary voxel array in BCdim'
91          info = 1
92          return
93       end if
         ! low-pass filtering: repalce each pixel by the average of its neighbors
94       ired = 0
95       jred = 0
96       kred = 0
         ! loop over every second voxel
97       do i=1,nvx,2
98          do j=1,nvy,2
99             do k=1,nvz,2
                  ! reset mean values
```

```
100               val = 0.0_MK
101               nval = 0
                  ! loop over its neighbors
102               do isx=-1,1
103                  idx = i+isx
104                  do isy=-1,1
105                     idy = j+isy
106                     do isz=-1,1
107                        idz = k+isz
108                        if((idx .GT. 0 .AND. idx .LE. nvx) .AND.   &
                              (idy .GT. 0 .AND. idy .LE. nvy) .AND.   &
                              (idz .GT. 0 .AND. idz .LE. nvz)) then
                              ! if inside array: update sliding mean
109                           val = val+real(voxel(idx,idy,idz))
110                           nval = nval + 1
111                        end if
112                     end do
113                  end do
114               end do
                  ! mean of surrounding voxel values
115               if(nval .GT. 0) val = val/real(nval)
                  ! set corresponding voxel in reduced data set to one if mean
                  ! is larger than threshold, else zero
116               ired = ceiling((1.0_MK/fact)*i)
117               jred = ceiling((1.0_MK/fact)*j)
118               kred = ceiling((1.0_MK/fact)*k)
119               voxtmp(ired,jred,kred) = 1
120               if(val .LT. thresh) voxtmp(ired,jred,kred) = 0
121            end do
122         end do
123      end do
          ! resize voxel array to reduced sizes
124      DEALLOCATE(voxel, STAT=istat)
125      if(istat .NE. 0) then
126         WRITE(*,'(A)') 'Error deallocating voxel array in BCdim'
127         info = 1
128         return
129      end if
130      ALLOCATE(voxel(nvxred,nvyred,nvzred), STAT=istat)
131      if(istat .NE. 0) then
132         WRITE(*,'(A)') 'Error reallocating voxel array in BCdim'
133         info = 1
134         return
135      end if
          ! assign new voxel values
136      voxel = voxtmp
137      nvx = nvxred
138      nvy = nvyred
139      nvz = nvzred
          ! deallocate temporary voxel array
140      DEALLOCATE(voxtmp, STAT=istat)
141      if(istat .NE. 0) then
142         WRITE(*,'(A)') 'Error deallocating temporary voxel array in BCdim'
143         info = 1
144         return
145      end if
146   end do
      ! also get last data point
147   Nvox(nred+1) = sum(voxel)
148   s(nred+1) = vx

      !------------------------------------------------------------------------
      ! Logarithms of results and output to file
      !------------------------------------------------------------------------

149   do i=1,nred+1
150      s(i) = log(1.0_MK/s(i))
151      logN(i) = log(real(Nvox(i)))
152   end do

153   OPEN(60, FILE='BC.gnu', STATUS='REPLACE', ACTION='WRITE')
154   do i=1,nred+1
155      WRITE(60,*) s(i), logN(i)
156   end do
157   CLOSE(60)

      !------------------------------------------------------------------------
      ! Least squares fit to estimate the dimension (slope)
      !------------------------------------------------------------------------
```

```
158   sumi = sum(s)
159   sumi2 = sum(s(:)**2)
160   det = sumi2*(nred+1.0_MK)-sumi*sumi
161   detinv = 1.0_MK/det
162   a = 0.0_MK
163   b = 0.0_MK
164   do i=0,nred
165      a = a + ((nred+1.0_MK)*s(i+1)-sumi)*logN(i+1)
166      b = b + (sumi2-s(i+1)*sumi)*logN(i+1)
167   end do
168   a = detinv*a
169   b = detinv*b

170   WRITE(*,'(A,F8.5,A,F8.5)') 'Box counting results:  a = ',a,',  b = ',b
      ! the slope is the box counting dimension
171   BCdim = a

      !-------------------------------------------------------------------------------
      ! Write gnuplot macro (invoke using: gnuplot plotit.mac)
      !-------------------------------------------------------------------------------

172   OPEN(60, FILE='plotit.mac', STATUS='REPLACE', ACTION='WRITE')
173   WRITE(60,*) 'set key top left'
174   WRITE(60,'(A,F9.6,A,F9.6,A)') 'p ''BC.gnu'' w p, ',a,'*x+',b,' w l'
175   WRITE(60,*) 'pause -1'
176   CLOSE(60)

      !-------------------------------------------------------------------------------
      ! free memory and terminate
      !-------------------------------------------------------------------------------

177   DEALLOCATE(Nvox)
178   DEALLOCATE(s)
179   DEALLOCATE(logN)
180   if(ALLOCATED(voxel)) DEALLOCATE(voxel)

181   return

182   END function BCdim
```

C.5.4.14 Convert text to upper case (UpperCase.f90)

`UpperCase.f90` takes the two input arguments, `string` and `ilen`, and converts the character string `string` of length `ilen` to all upper case letters. The result is again contained in the variable `string`.

```
1     subroutine UpperCase(string,ilen)

2     IMPLICIT NONE

      !-------------------------------------------------------------------------------
      !UpperCase  Converts a string to all upper case characters.
      !   UPPERCASE takes the string "string" of length ilen and converts
      !   every character of it to upper case.
      !
      !   See also ReadParams
      !
      !   todo:
      !
      !===============================================================================
      !  DIPLOMA THESIS WS01/02 ICOS                                    ETH-ZUERICH
      !-------------------------------------------------------------------------------
      !
      !              PROTEIN DIFFUSION INSIDE THE ENDOPLASMIC RETICULUM
      !
      !============================== ivo f. sbalzarini ==============================
      !

      !-------------------------------------------------------------------------------
      ! Input/Output arguments
      !-------------------------------------------------------------------------------

      ! the string to be converted
3     CHARACTER(LEN=*), INTENT(INOUT)            :: string
      ! length of the string
```

```
4     INTEGER, INTENT(IN)                       :: ilen

      !------------------------------------------------------------------------------
      ! Declaration of local variables
      !------------------------------------------------------------------------------

      ! loop counter
5     INTEGER                                   :: i, j
      ! alphabet boundaries and shift lower->upper
6     INTEGER                                   :: i1,i2,i3,iadd

      !------------------------------------------------------------------------------
      ! uppercase
      !------------------------------------------------------------------------------

      ! determine alphabet boundaries
7     i1   = IACHAR('a') - 1
8     i2   = IACHAR('z') + 1
9     i3   = IACHAR('A')
      ! shift to upper case
10    iadd = i3 - i1 - 1
      ! shift all lower case characters to upper case
11    do i=1,ilen
12       j = IACHAR(string(i:i))
13       if (j.GT.i1.AND.j.LT.i2) then
14          string(i:i) = CHAR(j+iadd)
15       end if
16    end do

      !------------------------------------------------------------------------------
      ! return
      !------------------------------------------------------------------------------

17    9999  CONTINUE
18    return

19    END subroutine UpperCase
```

C.5.4.15 Dynamic list structure handling (Util.f90)

`Util.f90` contains an overloaded set of dynamic list reallocation functions as described in section 7.2.6 and [Numerical Recipes in Fortran 90 (1996)]. Currently, the function is overloaded for real vectors (`reallocate_rv`), real matrices (`reallocate_rm`), integer vectors (`reallocate_iv`), integer matrices (`reallocate_im`) and character vectors (`reallocate_hv`). However, only the one for integer vectors is used in this work.

```
1     module Util

2     USE globals

3     IMPLICIT NONE

      !------------------------------------------------------------------------------
      !Util  contains common utility functions.
      !   UTIL currently only implements an overloaded dynamic array
      !   reallocation function
      !
      !
      !   See also INIT_PART, SORTT
      !
      !   todo:
      !
      !==============================================================================
      !  DIPLOMA THESIS WS01/02 ICOS                                   ETH-ZUERICH
      !------------------------------------------------------------------------------
      !
      !              PROTEIN DIFFUSION INSIDE THE ENDOPLASMIC RETICULUM
      !
      !============================ ivo f. sbalzarini ===============================
      !

      !------------------------------------------------------------------------------
      ! Interface declaration of contained functions
```

```
      !------------------------------------------------------------------------------

4     INTERFACE reallocate
5        MODULE PROCEDURE reallocate_rv,reallocate_rm,reallocate_iv,  &
              reallocate_im,reallocate_hv
6     END INTERFACE

7     CONTAINS

      !------------------------------------------------------------------------------
      ! Reallocate a pointer to a new size preserving its previous contents
      !------------------------------------------------------------------------------

      ! for real vectors
8     function reallocate_rv(p,n)
9       REAL(MK), DIMENSION(:), POINTER             :: p, reallocate_rv
10      INTEGER(I4B), INTENT(IN)                    :: n
11      INTEGER(I4B)                                :: nold, ierr

12      ALLOCATE(reallocate_rv(n), STAT=ierr)
13      if(ierr .NE. 0) then
14         WRITE(*,'(A)') 'Error allocating memory in reallocate_rv'
15         return
16      end if
17      if(.NOT. ASSOCIATED(p)) return
18      nold = size(p)
19      reallocate_rv(1:min(nold,n))=p(1:min(nold,n))
20      DEALLOCATE(p)
21    END function reallocate_rv

      ! for integer vectors
22    function reallocate_iv(p,n)
23      INTEGER(I4B), DIMENSION(:), POINTER         :: p, reallocate_iv
24      INTEGER(I4B), INTENT(IN)                    :: n
25      INTEGER(I4B)                                :: nold, ierr

26      ALLOCATE(reallocate_iv(n), STAT=ierr)
27      if(ierr .NE. 0) then
28         WRITE(*,'(A)') 'Error allocating memory in reallocate_iv'
29         return
30      end if
31      if(.NOT. ASSOCIATED(p)) return
32      nold = size(p)
33      reallocate_iv(1:min(nold,n))=p(1:min(nold,n))
34      DEALLOCATE(p)
35    END function reallocate_iv

      ! for character vectors
36    function reallocate_hv(p,n)
37      CHARACTER(1), DIMENSION(:), POINTER         :: p, reallocate_hv
38      INTEGER(I4B), INTENT(IN)                    :: n
39      INTEGER(I4B)                                :: nold, ierr

40      ALLOCATE(reallocate_hv(n), STAT=ierr)
41      if(ierr .NE. 0) then
42         WRITE(*,'(A)') 'Error allocating memory in reallocate_hv'
43         return
44      end if
45      if(.NOT. ASSOCIATED(p)) return
46      nold = size(p)
47      reallocate_hv(1:min(nold,n))=p(1:min(nold,n))
48      DEALLOCATE(p)
49    END function reallocate_hv

      ! for real matrices
50    function reallocate_rm(p,n,m)
51      REAL(MK), DIMENSION(:,:), POINTER           :: p, reallocate_rm
52      INTEGER(I4B), INTENT(IN)                    :: n, m
53      INTEGER(I4B)                                :: nold, mold, ierr

54      ALLOCATE(reallocate_rm(n,m), STAT=ierr)
55      if(ierr .NE. 0) then
56         WRITE(*,'(A)') 'Error allocating memory in reallocate_rm'
57         return
58      end if
59      if(.NOT. ASSOCIATED(p)) return
60      nold = size(p,1)
61      mold = size(p,2)
62      reallocate_rm(1:min(nold,n),1:min(mold,m))=p(1:min(nold,n),1:min(mold,m))
63      DEALLOCATE(p)
```

```
64    END function reallocate_rm

      ! for integer matrices
65    function reallocate_im(p,n,m)
66      INTEGER(I4B), DIMENSION(:,:), POINTER       :: p, reallocate_im
67      INTEGER(I4B), INTENT(IN)                    :: n, m
68      INTEGER(I4B)                                :: nold, mold, ierr

69      ALLOCATE(reallocate_im(n,m), STAT=ierr)
70      if(ierr .NE. 0) then
71         WRITE(*,'(A)') 'Error allocating memory in reallocate_im'
72         return
73      end if
74      if(.NOT. ASSOCIATED(p)) return
75      nold = size(p,1)
76      mold = size(p,2)
77      reallocate_im(1:min(nold,n),1:min(mold,m))=p(1:min(nold,n),1:min(mold,m))
78      DEALLOCATE(p)
79    END function reallocate_im

80    END module Util
```