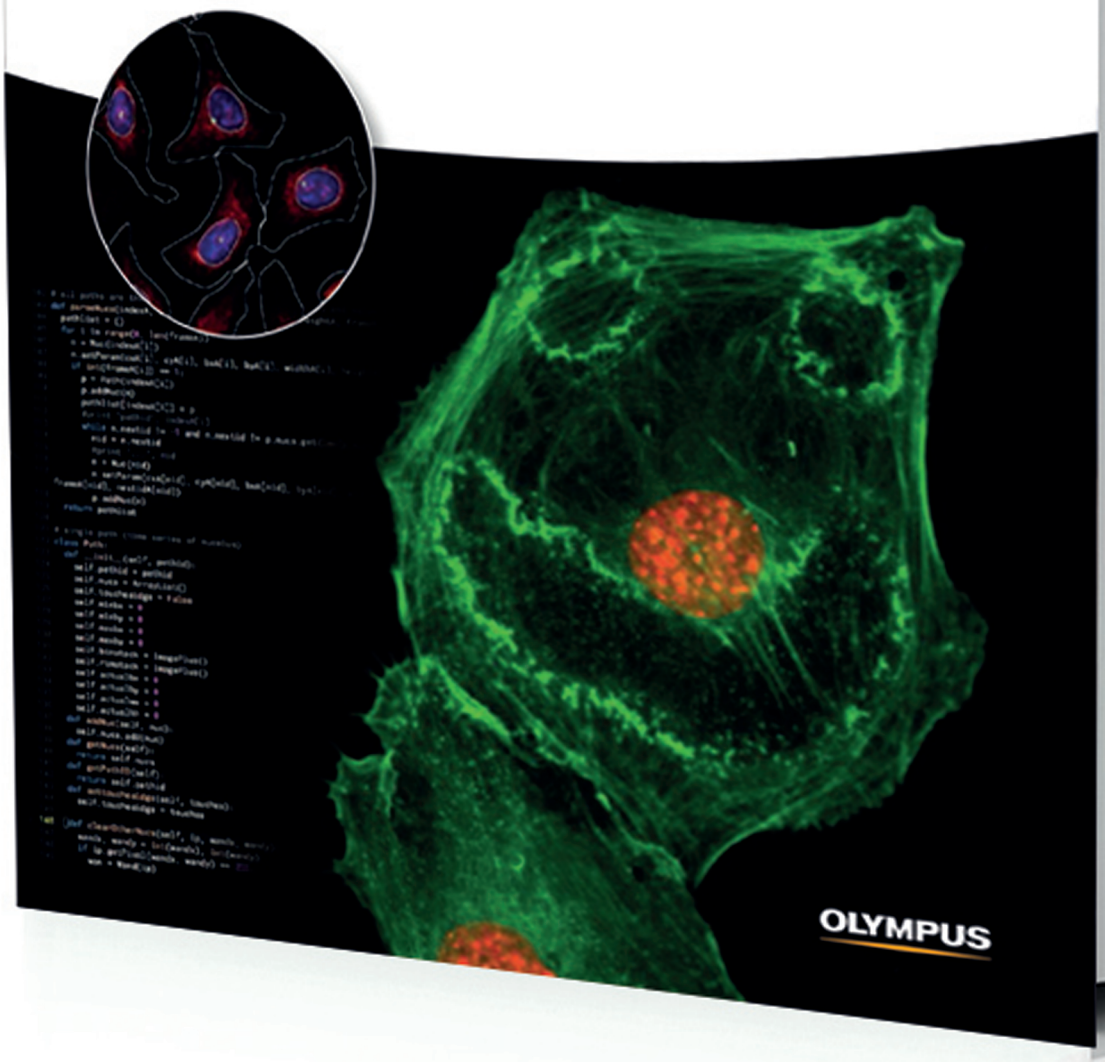
WILEY

Edited by Kota Miura

Bioimage Data Analysis



Edited by
Kota Miura

Bioimage Data Analysis

Edited by Kota Miura

Bioimage Data Analysis

WILEY-VCH
Verlag GmbH & Co. KGaA

Editor

Kota Miura
EMBL Heidelberg
69117 Heidelberg
Germany

We would like to thank Dr. Christian Tischler for the image used in the cover design.

This Publication was made possible with the generous support of Olympus.

Library of Congress Card No.: applied for

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available on the Internet at http://dnb.d-nb.de.

Print ISBN: 978-3-527-34122-1
ePDF ISBN: 978-3-527-80092-6
ePub ISBN: 978-3-527-80094-0
Mobi ISBN: 978-3-527-80093-3

Typesetting Thomson Digital, Noida, India

Printed on acid-free paper

Contents

Foreword

Modern light microscopy has become a popular tool for life science researchers, generating images of biological samples that at one point would have been limited to qualitative analysis. Technology has evolved, however, and in recent years the integration of microscopy equipment with computers has enabled advanced digital image analysis, with many powerful software packages and algorithms now available. The accuracy and objectivity achieved by extracting numerical data in this way presents a wealth of opportunities for analysis and measurement, greatly enhancing our understanding of complex biological systems.

To fully realize the variety of techniques within the field of biology known as bioimage data analysis, researchers must combine engineering skills in image processing with biological knowledge, matching a well-designed quantitative image analysis strategy to each biological question. Although many widely available textbooks cover theories and practices for image processing algorithms, practical guidance tailored to the biologist, describing how those algorithms can be applied in addressing biological questions, is largely lacking.

It is the purpose of this handbook to bridge this knowledge gap. Ideally suited to biologists – from postgraduate students to principal investigators – the book provides directions on how to analyze biological images. Experts in bioimage data analysis from across Europe begin by discussing the three most popular image processing and analysis software platforms – ImageJ, R, and Matlab. Programming using these platforms is becoming progressively more important to analyze an ever-increasing amount of image data. The next chapters describe specific topics in cell and developmental biology, which have been selected as realistic examples. These are accompanied by step-by-step instructions for treating images in a quantitative manner, and the protocols can easily be transferred and adapted for the reader's own study.

As a renowned leader in supplying researchers with the latest micro-imaging equipment, we are dedicated not only to enabling the capture of high-quality digital bioimages, but also to the analysis of vital image data. It is an honor to be asked by Kota Miura and Wiley to further support the life science research community by helping to realize this valuable resource for people whose work requires microscope image analysis. Through this handbook, we are looking forward to helping students, professors, and researchers to develop and master

their own image analysis protocols to help fuel the next generation of scientific discovery.

Head of Vertical Market *Dr. Martin Tewinkel*
Life Science Research EMEA
Scientific Solutions Division
Olympus Europa SE & Co. KG

Director of Marketing *David Rideout*
Life Science and Industrial Microscopes
Olympus Scientific Solutions Americas

Preface

In 2012, I started to design a course for bioimage analysis together with other experts, Sébastien Tosi (IRB Barcelona), Christoph Möhl (DZNE, Bonn), and Peter Bankhead (Heidelberg University). The aim of the course was to disseminate the knowledge and technique of bioimage analysis to a wider audience beyond our local colleagues. This motivation came not only from strong demands from our colleagues, but also from our own experiences. While many imaging courses had been indeed offered to the public in various places, those courses usually focused more on microscopy than image analysis. We felt that the existing courses were not enough for imaging beginners to start their analysis with their own data.

In order to design an intensive course that is highly practical for actual biological research projects, we asked other experts to suggest themes based on the actual biological problems they were analyzing. Bioimage analysis is always context-specific: a general solution is difficult to find as each project is unique in terms of the target sample, imaging conditions, and the question that is asked. However, those specific cases would provide a good template for other specific problems. Furthermore, a collection of various protocols may eventually allow us to have an overview of the landscape of current bioimage analysis strategies, which will be a valuable base to organize the knowledge we currently have.

Based on these backgrounds, "The EMBL Master Course for Bioimage Data Analysis" has been organized annually since May 2013. Peter Bankhead and Christoph Möhl left the organization team in 2013 and 2014, leaving behind many of their valuable inputs for the course design. Perrine Paul-Gilloteaux (Institut Curie, Paris), who was already an instructor in the 2013 course, joined the organization team from 2014. We always had many course applicants from all over the world, and we had to reject a majority of them because of the limitation in the number of seats. We then decided to offer the content of the course available to a much wider audience: We wrote detailed instructions, scripts, and protocols for each session of the course and compiled them as a single textbook, the one that you now have in your hand. To increase its accessibility and at the same time to keep the traditional form of publication and to maintain our original commitment, we looked for support to make an electric version of the

textbook freely available from established publishers. With valuable collaboration of Wiley and Olympus Europe, this became possible.

We are planning to publish two different types of releases of this textbook. We call them "milestone releases" and "continuous releases." The "milestone releases" are similar to "editions" in conventional publications. We will publish them with major updates of the content, both in hard copies and as an e-book. The continuous releases will be published online. Since bioimage analysis is a field rapidly evolving on a daily basis, protocols could rapidly become outdated. We need to modify the protocols and introduce newly developed algorithms to catch up with these changes. We also want to be interactive and responsive to comments and criticisms received from readers, and be open to adding new chapters with new topics. Only the online version of the textbook is capable to deal with such rapid release cycles.

As we organized the course annually and worked on the development of this textbook, a community of practice began to form among bioimage analysis experts, whom we now call "bioimage analysts". Our activities expanded to the organization of the European BioImage Analysis Symposium (EuBIAS), which is a complex of several different activities that aim to enhance communication among biologists, image analysts, and algorithm/software developers in order to boost the capability of image analysis in biological research community. These activities in EuBIAS include academic talks, courses for microscopy facility staffs, and highly interactive sessions among analysts and developers for organizing image processing algorithm implementations. If you become more interested in image analysis after reading this textbook and/or if you wish to become a professional bioimage analyst, EuBIAS would be the place where you could start getting involved in the field.

I thank Perrine Paul-Gilloteaux (Institut Curie), Sébastien Tosi (IRB Barcelona), and Julien Collombeli (IRB Barcelona) for their commitment in organizing the European BioImage Analysis Symposium with us, the scope of which now includes publishing and managing the continuous release of this textbook. I thank Cornelia Monzel (University of Heidelberg and Institut Curie) for keeping us on track to publish this textbook and also for the critical review of the introduction. I am grateful to the Course and Conference Office of EMBL Heidelberg, especially to Diah Yulianti, Carolina Garcia Sabate, Jacqueline Dreyer, and Sally Boehm who supported our bioimage analysis course for the past three years as administrators. All chapters in this textbook were originally prepared for those courses. I thank Antonio Costantino (EMBL Heidelberg), who was constantly helpful in looking for sponsors for the open access publication. I thank Gregor Cicchetti, Martin Friedrich, and Martin Graf-Utzmann (Wiley, Weinheim) for working together with us and for their enthusiasm in publishing this textbook. I thank Andreas Pfuhl (Olympus and Acquifer) and Ralph Schaefer (Olympus) for their efforts in realizing this textbook to be openly accessible. Jason Swedlow (University of Dundee) has been a passionate supporter of EuBIAS and we thank him for his continuous encouragement and support. Jean Salamero (Institut Curie, Paris) has been a great adviser and supporter of our initiative. I would like

to thank Rainer Pepperkok, Antje Kepler, and Jan Ellenberg (EMBL Heidelberg) for their valuable advice and support for the activities of image analysts community. I thank my family, Kai, Sho and Mayumi, for their patience while I was working at our home. I thank my sister, Ushio Miura, for flying from Bangkok to support us and also for correcting some of my texts. Finally, I would like to thank our fellow bioimage analysts – all the authors of this textbook, instructors, and former students of the course – for their full commitment and enthusiasm in learning, sharing, and transmitting this knowledge, and increasing the speed and quality of scientific research in the field.

Dec. 23rd, 2015 *Kota Miura*
Tokyo

1
Introduction

Kota Miura[1] *and Sébastien Tosi*[2]

[1]*European Molecular Biology Laboratory, Meyerhofstraße 1, 69117 Heidelberg, Germany*
National Institute of Basic Biology, Okazaki, 444-8585, Japan
[2]*Institute for Research in Biomedicine (IRB Barcelona), Advanced Digital Microscopy, Parc Científic de Barcelona, c/Baldiri Reixac 10, 08028 Barcelona, Spain*

Since the beginning of the twenty-first century, the use of digital imaging microscopy has spread widely among life scientists and analysis of image data became increasingly important. Already long before those technologies became available, starting in the seventeenth century, life scientists have been sketching – or imaging manually – living organisms and their structures to understand how they develop themselves and operate. Computer-based image analysis radically upgraded these traditional methods in life sciences and opened novel approaches to measure shapes, distributions, and dynamics from multidimensional images captured through high-end microscopes. From those data, life scientists are trying to decode the essence of biological systems in their spatial and temporal context. However, digital image analysis in life sciences is a new method that only became available recently in the life science community, and many scientists are still struggling to improve their image analysis skills.

We wrote this textbook aiming at such life scientists who have some experience in biological image analysis and who are trying to learn more to increase their own capability in extracting quantitative information from image data.

1.1
What Is Bioimage Analysis?

It might sound evident to you, but we would like to clarify our definition to avoid the confusion in the use of the term "image analysis" and also to be clear with the goal of this textbook.

Bioimage Data Analysis, First Edition. Edited by Kota Miura.

In the image processing field, "image analysis" is a way of identifying objects and patterns in images by computer. We quote a definition from a famous image processing textbook by Gonzalez and Woods [1]:

> Image analysis is a process of discovering, identifying, and understanding patterns that are relevant to the performance of an image-based task. One of the principal goals of image analysis by computer is to endow a machine with the capability to approximate, in some sense, a similar capability in human beings.

In the light of this definition, image analysis, which is also called "computer vision," aims at mimicking the way we see the world and how we identify its visible structures. Image analysis in biology does undeniably also hold this element, but more importantly, its main goal is to *measure* biological structures and phenomena in order to study and understand biological systems in a quantitative way.

To achieve this task, we in fact do not have to be bothered with similarity to the human recognition – we have more emphasis on the objectivity of quantitative measurement, rather than how that computer-based recognition becomes in agreement with human recognition. Therefore, in biology, image analysis is a process of identifying spatial distribution of biological components in images and measuring their characteristics to study their underlying mechanisms in an unbiased way. To underline this difference in the goals of image analysis in the two fields and to distinguish them from each other, we will now on refer to image analysis in biology as *bioimage analysis*.

1.2 Scope of this Textbook

The textbook starts with an overview of existing bioimage analysis tools and programming environments (Chapter 2).

The next two chapters (Chapters 3 and 4) are dedicated to the basics of programming in ImageJ Macro language and Matlab. These two software packages/programming environments are arguably the most widespread tools used for bioimage analysis. If you are already acquainted with these programming languages, you can skip this part.

Based on the programming skills acquired in the first chapters, Chapters 5–10 are devoted to typical biological problems. The reader is guided step by step to write increasingly challenging ImageJ and Matlab scripts addressing these problems. These scripts are powerful tools to automatically extract quantitative and statistical data from biological images.

Image analysis can also be performed without writing a single line of code, but programming does offer many advantages. First, repetitive tasks can be automated to decrease the user workloads. Second, written programs are the best practice to

secure the reproducibility of a given experimental protocol down to statistical results. Third, written programs can be excellent documentations of complex image analysis workflows, allowing us to inspect the details of the processing in a glance and providing a platform to further improve those methods. Finally, workflows are endowed with modularity, which enables us to construct new programs by reusing some sections of existing workflows and modifying them.

Many image processing and analysis tools are available and offered to life scientists, but instructions on how to efficiently combine functions of those tools and how to design workflows matched to a specific problem have largely been missing. We think that this is because bioimage analysis problems are so diverse that standardization of bioimage analysis is barely applicable. For this reason, each chapter focuses on a specific and clearly defined biological problem. By providing information on the general approach as well as details on solving the particular task, we hope that the reader will get valuable information to customize these image analysis workflows to one's own need, and also learn a template and good practices to write some new efficient workflows from scratch.

In this book, we minimized explanations on the details of each image processing step, that is, details of functions and algorithms. For their mathematical backgrounds and how they are implemented, many textbooks focusing on those aspects are readily available [1]. Instead, we provide more explanations on how to construct the image analysis workflows by assembling various image processing algorithms. These workflows aim at identifying biological objects and events, and extracting quantitative information on their geometry and dynamics.

In the context of bioimage analysis and microscopy we highly encourage beginners to read the open textbook "Analyzing fluorescence microscopy images with ImageJ" from Peter Bankhead [2]. We believe that this textbook is one of the best complement to ours, and that it will quickly bring you up to speed in digital image processing before dwelling in its concrete implementation. Pete does an exquisite job at clearly describing: 1) the nature of multidimensional digital images (section I), 2) how they can be processed by ImageJ (section II), and 3) their formation at the microscope and how it conditions the information they hold and the way to correctly extract and interpret this information (section III). Finally the book also contains a short introduction to ImageJ macro language (end of section II), a great appetizer to chapter 3 of our textbook.

We hope that our textbook will help you in solving your problems. Moreover, we are looking forward that you will soon discover the joy of programming bioimage analysis workflows and share that knowledge with your peers, to support your and their research in life sciences.

References

1 Gonzalez, R.C. and Woods, R.E. (1992) *Digital Image Processing*, Addison-Wesley, Reading, MA.

2 http://go.qub.ac.uk/imagej-intro.

2
Bioimage Analysis Tools

Kota Miura,[1] Sébastien Tosi,[2] Christoph Möhl,[3] Chong Zhang,[4] Perrine Paul-Gilloteaux,[5,6] Ulrike Schulze,[7] Simon F. Nørrelykke,[8] Christian Tischer,[9] and Thomas Pengo[10]

[1]European Molecular Biology Laboratory, Meyerhofstraße 1, 69117 Heidelberg, Germany National Institute of Basic Biology, Okazaki, 444-8585, Japan
[2]Institute for Research in Biomedicine (IRB Barcelona), Advanced Digital Microscopy, Parc Científic de Barcelona, c/Baldiri Reixac 10, 08028 Barcelona, Spain
[3]German Center of Neurodegenerative Diseases (DZNE), Image and Data Analysis Facility (IDAF), Core Facilities, Holbeinstraße 13–15, 53175 Bonn, Germany
[4]Universitat Pompeu Fabra, Carrer Tànger 122–140, Barcelona 08018, Spain
[5]Institut Curie, Centre de Recherche, Paris 75248, France
[6]Cell and Tissue Imaging Facility, PICT-IBiSA, CNRS, UMR 144, Paris 75248, France
[7]The Francis Crick Institute, Mill Hill Laboratory, The Ridgeway, Mill Hill, London NW7 1AA, UK
[8]ETH Zurich, Scientific Center for Optical and Electron Microscopy (ScopeM), Image and Data Analysis (IDA) Unit, HPI F15, Wolfgang-Pauli-Strasse 14, 8093 Zurich, Switzerland
[9]EMBL Heidelberg, Advanced Light Microscopy Facility, Meyerhofstraße 1, 69117 Heidelberg, Germany
[10]University of Minnesota, University of Minnesota Informatics Institute, Cancer and Cardiovascular Research Building, 2231 6th St SE, Minneapolis, MN 55445, USA

2.1
Overview of Bioimage Analysis Tools

In this chapter, we provide an overview of tools useful for bioimage analysis. The list is far from being exhaustive, but most of these are tools we use, or have used, in actual research projects. A few of the listed tools have been tested, but not used, by the authors; they are listed here for their potential and specific features that are not commonly found in other tools.

We grouped the tools into four different categories. Two major categories are the *graphical user interface* (GUI) and the *command line interface* (CLI). The computation engines of these tools are usually very similar (algorithms, libraries), but GUIs allow you to operate on images using menu selection, wizards, and mouse clicks. They are usually easier to get a grip on and also, as images are often displayed during processing, parameter tweaking and algorithm updates can be immediately confirmed by eye, in an interactive way. This allows the user

Bioimage Data Analysis, First Edition. Edited by Kota Miura.

to manually select regions of interest and easily try out various algorithms to visually track their effects.

In CLI software, the user inputs text commands to process and analyze images. This interface needs some knowledge on the Unix-style command line language, file system, and names of image processing algorithms, to send commands via the terminal.

Since images are not necessarily displayed, the CLI-based image processing and analysis is less intuitive than that based on GUI and it adds a hurdle for the beginner. However, the advantage is that once the user has learned this interface, extending it to a script is quite natural and seamless. The automation of processing and analysis can easily be achieved.

The third category of image analysis and exploration tools consists of *image databases*. As the number and size of image data increase, the only way to effectively handle the data is through the use of database management systems. They typically allow us to organize the data, metadata, and analysis results in projects, and provide remote viewing capability of the images. They sometimes also support user annotations and allow us to launch batch analysis scripts on user-defined regions of interest. We will introduce several such systems.

The fourth category of tools consists of *libraries*. These tools are mainly interfaced for use by programming or scripting. Both GUI and CLI tools often use these libraries in their back end. As the direct use of image processing libraries is uncommon for the average biologist, we exclude them from this overview.

A summary of all tools appearing this chapter is provided in Appendix 2.A.

2.2 GUI Tools: Generic Platforms

2.2.1 ImageJ

ImageJ is one of the most widely used software for bioimage analysis [1]. It is a public domain software, completely open source and free to download (Table 2.1). It runs on the Java virtual machine, which allows the software to be largely free from dependence on operating systems. Usage is open to broad type of users, from biologists with marginal knowledge on image processing to

Table 2.1 Links to ImageJ packages.

Package	URL
ImageJ	http://imagej.nih.gov/ij/
Fiji	fiji.sc
ImageJ2	imagej.net/ImageJ2
Bio7	bio7.org

the experts of image processing algorithm development. It has an intuitive graphical user interface that allows easy access for beginners, but at the same time it allows serious development of plug-ins to add new functions. Many plug-ins are developed worldwide and most of them are freely offered. The community of ImageJ users and developers is huge: it has a highly active mailing list with nearly 1900 subscribers. The ImageJ website has had nearly 7000 visitors per day, and the full text search for "ImageJ" in Europe PubMed Central returned more than 55 000 articles (May 2015) and during 2014 there were 9280 articles using ImageJ. The last ImageJ conference in Luxembourg was attended by 119 people. The first release of ImageJ was in 1997, and as of May 2015, it is still under active development by Wayne Rasband. The flexible interface of ImageJ invoked a huge number of plug-in development projects, resulting in numerous implementations of image processing algorithms and convenient utilities. This availability of a variety of functions contributed largely to the spread of ImageJ all over the world. A problem associated with this wide impact was that dependencies between those plug-ins became complicated, and often hard to untangle. To resolve this, a distribution of ImageJ bundled with many plug-ins called Fiji was initiated in 2007 and has been actively maintained by a large group of developers [2]. In normal ImageJ there are about 500 commands, and in this distribution called Fiji, the number exceeds 900. The central feature of Fiji is its automatic plug-in updating function.

Bio7 is a unique distribution that focuses on merging ImageJ and R (see description in Section 2.7.4) in a single interface [3]. The passing of image analysis output to statistical analysis by R could be smoothly achieved.

Since 2009, ImageJ2 project has started to inherit the best part of ImageJ while upgrading the core architecture of the software. It is a complete rewrite of ImageJ, but its compatibility to ImageJ environment such as plug-ins and ImageJ Macro is maintained. ImageJ2 is already a part of Fiji, but by default Fiji runs on ImageJ. Switching to ImageJ2 back end is possible by changing the configuration.

2.2.2 Icy

Icy (icy.bioimageanalysis.org/) is a software platform for bioimage analysis with a strong emphasis on collaborative efforts [4]. Plug-ins for Icy are managed through its highly interactive website aimed at collecting inputs and feedbacks from both developers and biologists. Several different approaches are offered for constructing image analysis workflows. For biologists who are not accustomed to programming, a visual programming interface is provided that allows the user to graphically design image analysis workflows, just like LabVIEW and KNIME. Resulting visual programs are associated with the capacity to automatically check its plug-in dependencies and install any missing releases. This ensures the perfect reproduction of image analysis workflows documented in published papers. At the same time, JavaScript and Python are also integrated for scripting workflows in a conventional way. The learning curve for the beginner is

facilitated by rich examples and an intuitive keyword search. It integrates various Java computing libraries to ease the development of custom plug-ins. For example, ImageJ is included as a back end and ImageJ plug-ins can be launched from other plug-ins or from an integrated version of ImageJ. Other integrated libraries include ImgLib2, VTK, and OpenCV.

2.3 GUI Tools: Workflow Based

2.3.1 CellProfiler + CellProfiler Analyst

CellProfiler (www.cellprofiler.org/) is designed to enable biologists without training in computer vision or programming to quantitatively measure cell or whole-organism phenotypes from thousands of images automatically [5,6]. The researcher creates an analysis workflow called "pipeline" from "modules" that first find cells and cell compartments and then measure features of those cells to extract numerical data that characterize the biological objects and phenomena. Pipeline construction is structured so that the most general and successful methods and strategies are the ones that are automatically suggested, but the user can override these defaults and pull from many of the basic algorithms and techniques of image analysis to customize solution to problems.

Because it was initially developed for high-content screening image-based assay, it has been constructed to easily batch process thousands of images. User-defined pipelines can be saved and reused afterward.

CellProfiler pipeline is a simple text file and it could be executed from command line using CellProfiler.py script. CellProfiler can also be used as a library directly from Python script by importing modules. CellProfiler can be extended through plug-ins written in Python or for ImageJ.

CellProfiler Analyst is a companion software for CellProfiler, which can also be used by its own to analyze large data set of features extracted from images. It allows interactive exploration and analysis of measured data, including a supervised machine learning system to recognize subtle phenotypes.

2.3.2 ilastik

ilastik (ilastik.org/) is a framework, GUI, and suite of workflows that facilitate automated segmentation, classification, tracking, and counting in 2D and 3D multispectral images and videos [7]. These workflows are cast as interactive machine learning problems, which require no user experience in image processing but rather provide example annotations. Taking the pixel classification workflow as an example, ilastik has a convenient mouse interface for labeling an arbitrary number of classes in the images. These labels, along with a set of image

features, are then used for a machine learning-based method to classify image regions in different classes. In the interactive training mode, ilastik provides real-time feedback of the current classifier predictions and thus allows for targeted training and overall reduced labeling time. Once the classifier has been trained on a representative subset of the data, it can be used to automatically batch process a very large number of data sets. Other workflows follow similar usage steps. The plug-in functionality allows advanced users to add their own problem-specific features, apart from the provided set of features based on color, edges, and textures in the image. ilastik projects can be further imported to other tools such as CellProfiler and KNIME, for specific post-processing or analysis tasks.

2.3.3 Definiens Developer XD

Definiens Developer XD (developer.definiens.com/) is a commercial image segmentation and classification tool. The user designs a signal processing workflow by combining built-in filtering, thresholding, and object classification modules. Object detection is typically done on hierarchical object levels, for example, cell level for cell objects and organelle level for nucleus and ER objects inside a cell object. For each object, a huge set of features (shape-based, intensity-based, relations to neighboring objects, etc.) is available and can be used for object classification or merging with neighboring objects. The classical Definiens workflow is the so-called bottom-up approach: in the first step, the image is segmented in numerous small objects, resulting in a heavy oversegmentation of the target objects. Objects are then fused step by step on the basis of features such as the relative border to neighboring object or an elliptic fit of resulting (fused) object. Objects can be assigned to different classes (such as nucleus or cancer cell) based on their features.

2.4 GUI Tools: 4D + *t* Data Exploration and Analysis

2.4.1 Amira

Amira (now distributed and maintained by FEI, http://www.fei.com/software/amira-3d-for-life-sciences/) is a 3D visualization software, allowing also to do 3D processing and quantification (using Visilog and ITK libraries, as well as specific filters). Amira takes full advantage of the hardware acceleration and some filters could be run on NVidia graphical cores. The learning curve of Amira is eased by the presence of tutorials and example, but its object-oriented philosophy makes it sometimes difficult to apprehend for beginners, even though the functions are quite similar to professional GUI animation software. Amira can be controlled by the TCL scripting language, and is interfaced with Matlab. Additional modules can be created using C++. One of the main strength of

Amira is the number and quality of registration and optimization algorithm for automatic data fusion. It will be merged with Avizo (more physics/industry oriented) in the near future.

2.4.2 Arivis Vision4D

Arivis Vision4D (http://vision.arivis.com/en/arivis-Vision4D) is commercial software developed in Rostock, Germany. The main strength of Vision4D is its ability to handle really large data sets, for example, SPIM and EM data in the terabyte range. This is achieved through the use of a proprietary file format: When first opening and importing image data, the data are automatically copied into a sophisticated format that allows rapid Google Maps-like zooming and panning. It is possible to interactively visualize, process, segment, and analyze multidimensional data (3D + time + channels + scopes). The analysis is pipeline-based and several modules are preinstalled, for example, colocalization, FRET, and tracking. For this, filters/image processing steps from ITC and VIGRA are available. Import (even acquisition from TWAIN devices) of a wide, and growing, variety of image formats is supported and export in tiff, jpg, and so on is possible. Arivis WebView makes it possible to set up a web browser interface for collaborative work, for example, manual annotation of a large data set, allowing multiple users to simultaneously work on the same data. Arivis is a young company and its products are under rapid development at the moment of writing (2015).

2.4.3 Imaris

Bitplane Imaris (http://www.bitplane.com/imaris/imaris) is a commercial software for visualization and analysis of 3D data sets from fluorescence microscopy. The fast rendering engine together with the clear user interface makes it easy to reconstruct and interactively explore 3D data.

Image analysis tools are organized in workflows; that is, the user is guided through a wizard if he/she wants to detect and analyze objects in the image. One example is the “Find Spots” workflow, a method to segment spot-shaped objects and track them over time. Within the “Find Spots” wizard, the user is asked to define detection parameters step by step. Instant visual feedback allows us to fine-tune parameters as, for example, intensity thresholds. Besides others, workflows for segmenting cells as well as a neuron reconstruction tool are available. After completing the wizard, detected objects are rendered in 3D and can be overlaid with the original data. Statistics of detected objects can be exported to CSV files or visualized inside Imaris with the Vantage module. Due to the workflow-oriented approach, the software is very easy to use, but not very flexible. To overcome these restrictions, Imaris can be customized with plug-ins (so-called Xtensions) via the ImarisXT API. With the interface, image and object data can be accessed with scripts written in Matlab or Python.

Besides its image analysis capabilities, the software provides a very easy-to-use tool for rendering animations. The user defines some keyframe views by rotating and zooming in the data set and the software computes smooth camera movements from one keyframe to the next. The result can be exported to standard movie formats (e.g., avi format).

2.4.4 Volocity

Volocity (http://cellularimaging.perkinelmer.com/downloads/) is a commercial software package that was created by PerkinElmer. It consists of a free-of-charge core packet, called "Volocity LE," and several extensions with additional functionaries, which must be purchased: "Volocity LE" enables the user to import image sequences and perform basic image processing. The "Volocity Visualization" extension offers 3D rendering and interactive exploration of multichannel data sets. "Volocity Classification" allows for measuring and tracking of microscopy data in three dimensions over time. The "Volocity Restoration" package offers rapid and easy deconvolution, using measured or estimated point spread functions.

The advantages of Volocity lie in its ease of use. Volocity is designed to enable scientists without expertise in programming to visualize and analyze 3D-based data sets. A graphical user interface is used to process images in a "what you see is what you get" manner. Complex fluorescence *z*-stack images from wide-field and confocal microscopes are easily processed and analyzed.

While it is a powerful tool to read and prepare 3D fluorescence data for presentation and to process images, it does not allow the user to extend its functions by self-written macros. Therefore, this software is a tool for application only.

2.4.5 Vaa3D

Vaa3D (www.vaa3d.org) is a handy, fast, and versatile 3D/4D/5D image visualization and analysis open-source software for bioimages and surface objects [8]. It is particularly oriented toward filament-like structure tracing and provides some unique segmentation and analysis functions. It supports a very simple and powerful plug-in interface to extend its base functions.

2.5 GUI Tools: Image Restoration and Analysis

2.5.1 AutoQuant X

AutoQuant X (Media Cybernetics, www.mediacy.com/index.aspx?page=autoquant) is a commercial software suite mainly used for the deconvolution of fluorescence

microscopy images (wide-field, confocal, two-photon, and spinning disk) and their correction (aberration, registration, and bleaching). The software can also deal with image quantification (colocalization, ratiometry, and FRET) or data analysis and 3D visualization. In simple terms, deconvolution is an image restoration operation by which out-of-focus light is removed from a three-dimensional stack of images. The goal is to reconstruct the original emission pattern from the blurred and degraded version acquired with the microscope. The most advanced deconvolution algorithms provided (blind and nonblind maximum likelihood) are very robust, among the most efficient available, and their implementation is computationally highly optimized. Some more classical algorithms (no/nearest neighbors, inverse, and Wiener filters) as well as differential interference contrast (DIC) image restoration and 2D image deconvolution are also implemented.

2.5.2 SVI Huygens

SVI Huygens (svi.nl/HomePage) is a commercial software suite for the deconvolution of microscopy images. In simple terms, deconvolution is an image restoration operation by which out-of-focus light is attenuated from a 3D stack of images. The goal is to reconstruct the original emission pattern from the blurred and degraded version acquired with the microscope. The Huygens suite in its current version (version 15.05 at the time of writing) can be used with images acquired in the following modalities: wide-field, confocal, Nipkow confocal (spinning disk), STED, and SPIM. The software suite also includes optional visualization and analysis algorithms, including a surface renderer, a "simulated fluorescence process renderer," object tracking, and colocalization, among others.

2.6 GUI Tools: Specialized Software

2.6.1 CellCognition

CellCognition (www.cellcognition.org/) is a computational framework dedicated to the automatic analysis of live cell imaging data in the context of high-content screening, initially specialized in the detection of mitotic events and to follow their time course [9]. Its GUI is called CeCogAnalyzer, which allows biologists to parameterize analysis workflow. It contains machine learning-based algorithms for segmentation of cells and cellular compartments based on various fluorescent markers, features to describe cellular morphology by both texture and shape, tools for visualizing and annotating the phenotypes, classification, tracking, and error correction. CellCognition can be used by novices in the field of image analysis and is applicable to hundreds of thousands of images in computer clusters with minimal effort.

2.6.2 NeuronStudio

NeuronStudio (http://research.mssm.edu/cnic/tools-ns.html) is an open-source software designed to allow reconstruction of neuronal structures from confocal and multiphoton images [10,11]. It is a self-contained software package that is free and easy to use. NeuronStudio provides tools for manual, semi-manual, and automatic tracing of the dendritic arbor as well as manual and automatic detection and classification of dendritic spines. In addition, advanced 2D and 3D visualization techniques facilitate the verification of the reconstruction, as well as allowing accurate manual editing. The output of the program is compatible with standard compartment modeling and morphometric software applications. The software is very optimized, but unfortunately still in beta release and the project is not active since 2009.

2.6.3 TMARKER

TMARKER (http://www.nexus.ethz.ch/equipment_tools/software/tmarker.html, https://github.com/ETH-NEXUS/TMARKER) is an open-source application for the detection and classification of nuclei in immunohistochemical (IHC) tissue microarrays (TMAs). The software is written in Java, is free, is easy to install, has a straightforward user interface, and comes with easy-to-follow tutorials and test data. TMARKER was developed for nuclei counting and nuclear IHC staining estimation of human cancer tissue and can be expected to perform best on similar data.

The main usage is the automated detection of nuclei in color images via color deconvolution and/or supervised via graph cuts or superpixels. After detection, nuclei can be classified via built-in supervised machine learning models using random forests, Bayesian networks, or support vector machine (SVM) algorithms. Some statistical diagnostics, such as *F*-score and precision/recall plots, can be performed directly in TMARKER. While technically advanced, the input from the user is usually limited to estimating the nucleus size and a few other parameters and then clicking on nuclei that exemplify the different classes of interest.

After segmentation and classification, the results can be exported as csv files, or as html to form a small report complete with results and images. Segmentation settings and results can also be saved as xml and tma (TMARKER) files and later loaded for further processing in TMARKER.

2.7 CLI Tools

With many of the tools discussed above, images are processed interactively while viewing the effect of processing on each image. In command line-based tools

listed below, images are typically imported without opening the image on desktop: they are loaded as an object in the memory and processing is done without visually checking changes to the image.

2.7.1 Matlab

Matlab (Matrix Laboratory, http://www.mathworks.com/products/matlab/) is a commercial integrated development environment (IDE) and a programming language oriented toward matrix manipulation and linear algebra launched by MathWorks in 1984. It is essentially a scripting language used to combine optimized high-level functions. To complement the extensive set of functions provided with the IDE, a large number of commercial and open toolboxes are available. The functions can be called from the interactive console or assembled as scripts. The most useful toolboxes for bioimage analysis are most certainly the image processing and the statistical toolboxes. Images are formally manipulated as matrices or higher dimension objects. Most common image processing and data analysis operations are implemented. Data visualization and simple GUI design are also extensively covered. Matlab is typically used for workflow prototyping, exchange, and publication. A large community of users is openly sharing their scripts. Matlab can also be used to release end products as the scripts can be compiled to executable files and ran outside the environment, or even encrypted. A free alternative to Matlab is Octave (http://www.gnu.org/software/octave/index.html), offering less toolboxes.

2.7.2 KNIME

KNIME (https://tech.knime.org/community/image-processing), the Konstanz Information Miner, is an open-source platform that contains modules for data integration, transformation, analysis, visualization, reporting, and integration [12]. KNIME's friendly graphical interface allows users to visually create workflows/pipelines as assembly of nodes. This enables the flexibility of independently executing one, some, or all nodes' tasks, and inspecting their results. KNIME is implemented in Java with extension mechanism. It also allows for wrappers calling other code written in Python, Perl. KNIME offers possibility to integrate other open-source projects such as R, LIBSVM, JFreeChart, ImageJ, CellProfiler, and the Chemistry Development Kit. Recently, ilastik has also provided projects that can be integrated with KNIME and CellProfiler developers are now collaborating together with KNIME team to allow KNIME to run CellProfiler pipeline. In terms of processing large data, it is only limited by the available hard disk space, as opposed to most other open-source tools that are limited to the available RAM.

The KNIME Image Processing plug-in is a large repository based on ImgLib2, a generic multidimensional image processing library also used in Fiji and Icy. In

particular, it provides >120 image formats, algorithms for preprocessing, filtering segmentation, feature extraction, tracking, and classification.

2.7.3 Python

Python (www.python.org/) is not a software package, but is a scripting language. The merit of Python is that there are numerous libraries for image processing and analysis. In terms of scripting, Python is more powerful than Matlab due to its bridging capability to many computer languages such as C, C++, and Java. Considering that the trend of image processing and analysis is getting more and more toward cross-language library usage, Python is a good choice to learn for high-end processing and analysis.

2.7.4 R/Image Processing in R

R is a software environment for statistical computing and graphics (Table 2.2). Numerous add-ons, called "packages," written by researchers are freely available via the Comprehensive R Archive Network (CRAN). R is widely used in combination with other image processing software packages for analyzing numerical data extracted from image data and to plot those results. RStudio is an IDE for R. It combines many useful features such as markdown-based documentation system called "R Markdown" and the version control using Git. Besides its usefulness for statistical computing, image processing and analysis is also possible in R using "EBImage" toolbox, a part of Bioconductor project.

2.7.5 LabVIEW

LabVIEW (Laboratory Virtual Instrument Engineering Workbench, http://www.ni.com/labview/) is a commercial visual programming language launched by National Instruments in 1986. The graphical language is named "G". This graphical language naturally allows task parallelism and synchronization. It is primarily aimed at instrument control, and many devices (cameras, stages, lasers, etc.) come with LabVIEW drivers and software development kits. Like Matlab, LabVIEW became popular in many fields of engineering and its functions are organized in toolboxes. LabVIEW Vision is an extensive image processing and image analysis toolbox that

Table 2.2 Links to R and related packages.

Package	URL
R	www.r-project.org/
RStudio	http://www.rstudio.com/
EBImage	http://www.bioconductor.org/packages/release/bioc/html/EBImage.html

teams up well to drive hardware equipped with video camera and relying on computer vision for its operation (e.g., quality control, microscopes, etc.). It can, of course, also be used for independent, offline, image processing and analysis.

2.7.6 IGOR Pro

IGOR Pro (https://www.wavemetrics.com/products/igorpro/igorpro.htm) is a commercially available data analysis software developed by WaveMetrics, Inc. When it was introduced in 1989, it aimed at time series analysis. Since then, it has largely evolved and, nowadays, covers other applications such as curve fitting and image processing. A library for image processing is readily available, and an add-on for live image capturing is also available. Being mostly used by technical professionals, it is highly suitable for experimentation with scientific and engineering data and allows for the production of publication-quality graphs and page layouts. It comes with its own programming language called "IgorPro Procedure" and compiler allowing to extend the built-in functions or writing independent add-on in C or C++ ("XOP Tool Kit"). While this requires some analytical and programming skills, many users value the mixture of provided functions and freedom to extend them for a most suitable acquisition and analysis of data.

2.8 Image Databases

2.8.1 OMERO

OMERO (http://www.openmicroscopy.org/site/products/omero), developed by the Open Microscopy Environment consortium, is a software suite for the management of biological microscopy data. It is composed of a back-end server suite with the database and appropriate management software, and a set of front-end interfaces, which communicate at different levels with the server, such as desktop clients, web applications, and software libraries (to access the data remotely from Java, Matlab, Python, and other development platforms). Since version 5, the images are kept in their original format, which allows the user to have access to the original file, as well as to interact with the data through the front-end applications. It supports regions of interest, data tables, and the remote execution of analysis routines.

2.8.2 BisQue

BisQue (Bio-Image Semantic Query User Environment, bioimage.ucsb.edu/bisque) is a scientific image management tool organized around a server–client architecture. Its purpose is to store, visualize, organize, and analyze images in the cloud.

The images can be imported to the database (100+ biological formats supported) with relevant metadata and integrated to data sets. It is then possible to add hand-drawn annotations (possibly in collaboration) and to launch custom-made ImageJ, Matlab, or Python scripts to analyze the images. Search and comparison of data sets by image data and content are supported. The queries support high-level semantic articulations.

2.8.3 openBIS

The Open Source Biology Information System (openBIS, http://www.cisd.ethz.ch/software/openBIS) was developed by the Scientific IT Services (SIS) group at ETH Zurich, Switzerland. Its purpose is the management, annotation, and sharing of data. The openBIS software framework is extensible and has been customized for high-content screening, proteomics, metabolomics, and deep sequencing. Typically, openBIS is installed on a Linux server by a system administrator and accessed through a web interface by the users. openBIS is a free open-source software: powerful, well documented, and quite flexible. However, to take full advantage of this flexibility you do need to know a fair level of Python, shell scripting, and command line tools. Data can be stored in a distributed manner on multiple separate network-associated servers and the level of access via the web interface is easy to control, for example, which data a given user is allowed to read and/or edit. It is also possible to use openBIS as the platform when setting up a publication server where data are shared with the general public.

2.8.4 Avadis iMANAGE

Avadis iMANAGE (http://www.strandls.com/solutions/strand-imanage), developed in collaboration between Strand Life Sciences and Institut Curie imaging facility and IT department, is a ready-made commercial software suite for the management of biological microscopy data in particular on facilities. It provides shared, secure, uniform, and open access to image life cycle data and algorithms, through an easy access to cluster computing from the database web client. It has been thought to be highly scalable, and to deal with storage quota specifications and invoicing. Dynamic folder hierarchies can be created based on image and user metadata can be created on the fly. All data can be accessed and parsed by blocks (only relevant bits for a process can be chosen to be accessed). There is no conversion of data, neither duplication. A rich application programming interface in Java and SOAP allows easy extensions.

Appendix 2.A

Interfaces, execution environments, ecosystem, and licenses are summarized in Table 2.3.

Table 2.3 Summary of image analysis tools.

Name	GUI	CLI	Scripting	OS	Scripting Language	License
ImageJ	O	O	O	All	ImageJ Macro, Javascript, Jython, JRuby, BeanShell, Groovy, Clojure	Public domain
Icy	O	O	O	All	ImageJ Macro, Javascript, Jython, Protocol (Graphical Programming)	GPL v3
CellProfiler/ Analyst	O	O	O	All	Python, Workflow	GPL v2
ilastik	O	O	O	All	Python	GPL
Definiens Developer XD	O	O	O	All	Definiens Rulesets (Graphical Programming)	Commercial
Amira	O	O	O	All	TCL, Python	Commercial
Arivis Vision4D	O	X	O	Win	Python	Commercial
Imaris	O	X	X	Win, OSX	Matlab	Commercial
Volocity	O	X	X	Win, OSX	—	Commercial
Vaa3D	O	O	X	All	—	MIT
AutoQuant X	O	X	X	Win	—	Commercial
Huygens	O	O	O	All	TCL	Commercial
CellCognition	O	O	O	All	Python	LGPL
NeuronStudio	O	X	X	Win	—	Original
Matlab	X	O	O	All	Matlab	Commercial
KNIME	O	O	O	All	Workflow (Graphical Programming), Perl, Python, ImageJ Macro, Groovy (Java), R, Matlab	GPL
Python	X	O	O	All	Python	Python Software Foundation License
R	X	O	O	All	R	GPLv2/v3
LabVIEW	X	O	O	All	LabVIEW VI (Graphical Programming)	Commercial
IGOR Pro	X	O	O	Win, OSX	Igor Procedure	Commercial
OMERO	O	O	O	All	Python	GNU public "copyleft" license
BisQue	O	X	X	All	—	BSD (modified)
openBIS	O	O	O	All (client)/ Linux (server)	Java, Jython	Apache Software License 2.0

(*continued*)

Table 2.3 (*Continued*)

Name	GUI	CLI	Scripting	OS	Scripting Language	License
Avadis iMANAGE	O	X	O	All	SOAP, Matlab, ImageJ macro, ICY Protocol	Commercial
TMARKER	O	X	X	All	—	GPL v2

References

1 Schneider, C.A., Rasband, W.S., and Eliceiri, K.W. (2012) NIH Image to ImageJ: 25 years of image analysis. *Nat. Methods*, **9** (7), 671–675.

2 Schindelin, J., Arganda-Carreras, I., Frise, E., Kaynig, V., Longair, M., Pietzsch, T., Preibisch, S., Rueden, C., Saalfeld, S., Schmid, B., Tinevez, J.Y., White, D.J., Hartenstein, V., Eliceiri, K., Tomancak, P., and Cardona, A. (2012) Fiji: an open-source platform for biological-image analysis. *Nat. Methods*, **9** (7), 676–682.

3 Austenfeld, M. and Beyschlag, W. (2012) A graphical user interface for R in a rich client platform for ecological modeling. *J. Stat. Software*, **49** (4), 1–19.

4 de Chaumont, F., Dallongeville, S., Chenouard, N., Hervé, N., Pop, S., Provoost, T., Meas-Yedid, V., Pankajakshan, P., Lecomte, T., Le Montagner, Y., Lagache, T., Dufour, A., and Olivo-Marin, J.C. (2012) Icy: an open bioimage informatics platform for extended reproducible research. *Nat. Methods*, **9** (7), 690–696.

5 Carpenter, A.E., Jones, T.R., Lamprecht, M.R., Clarke, C., Kang, I.H., Friman, O., Guertin, D.A., Chang, J.H., Lindquist, R.A., Moffat, J., Golland, P., and Sabatini, D.M. (2006.) CellProfiler: image analysis software for identifying and quantifying cell phenotypes. *Genome Biol.*, **7** (10), R100.

6 Kamentsky, L., Jones, T.R., Fraser, A., Bray, M.A., Logan, D.J., Madden, K.L., Ljosa, V., Rueden, C., Eliceiri, K.W., and Carpenter, A.E. (2011) Improved structure, function and compatibility for CellProfiler: modular high-throughput image analysis software. *Bioinformatics*, **27** (8), 1179–1180

7 Sommer, C., Straehle, C., Kothe, U., and Hamprecht, F.A. (2011) ilastik: interactive learning and segmentation toolkit. Proceedings of the International Symposium on Biomedical Imaging, pp. 230–233.

8 Peng, H., Ruan, Z., Long, F., Simpson, J.H., and Myers, E.W. (2010) V3D enables real-time 3D visualization and quantitative analysis of large-scale biological image data sets. *Nat. Biotechnol.*, **28** (4), 348–353.

9 Held, M., Schmitz, M.H.A., Fischer, B., Walter, T., Neumann, B., Olma, M.H., Peter, M., Ellenberg, J., and Gerlich, D.W. (2010) CellCognition: time-resolved phenotype annotation in high-throughput live cell imaging. *Nat. Methods*, **7** (9), 747–754.

10 Rodriguez, A., Ehlenberger, D.B., Dickstein, D.L., Hof, P.R., and Wearne, S.L. (2008) Automated three-dimensional detection and shape classification of dendritic spines from fluorescence microscopy images. *PLoS One*, **3** (4), e1997.

11 Wearne, S.L., Rodriguez, A., Ehlenberger, D.B., Rocher, A.B., Henderson, S.C., and Hof, P.R. (2005) New techniques for imaging, digitization and analysis of three-dimensional neural morphology on multiple scales. *Neuroscience*, **136** (3), 661–680.

12 Berthold, M., Cebron, N., Dill, F., Gabriel, T., Kötter, T., Meinl, T., Ohl, P., Sieb, C., Thiel, K., and Wiswedel, B. (2008) KNIME: The Konstanz Information Miner, in *Data Analysis, Machine Learning and Applications*, Studies in Classification, Data, Analysis, and Knowledge Organization (eds C. Preisach, H. Burkhardt, L. Schmidt-Thieme, and R. Decker), Springer, Berlin, pp. 319–326.

3
ImageJ Macro Language

Kota Miura

European Molecular Biology Laboratory, Meyerhofstraße 1, 69117 Heidelberg, Germany
National Institute of Basic Biology, Okazaki, 444-8585, Japan

3.1
Aim: Why Do We Write ImageJ Macro?

The aim of this small chapter is to teach the basics of automation of image processing and analysis using ImageJ macro language. Why do we write ImageJ macro? We write macros to decrease our workloads in image processing: less clicking and less repetitive procedures.

3.2
Introduction

3.2.1
ImageJ Macro Makes Your Life Easier

To customize the functions in ImageJ, a typical way is to write a Java plug-in that directly accesses the application interface of ImageJ. This is a powerful method of customizing your own tool, but in many cases it is a bit too much for small tasks that we often encounter in biological research projects. Compared to the Java programming, ImageJ macro is much easier to access and to quickly solve problems.

A typical usage is to automate repetitive tasks involving hundreds of times of mouse clicking. Clicking can range from menu selection to inspection of single pixel value. By writing a macro, we can save such exhausting job to a single execution of a macro file, which is a text file with a sequence of image processing commands. As ImageJ macro functions directly mirror the GUI menu items, one can intuitively learn how to write one's own macro even without much experience in programming.

Bioimage Data Analysis, First Edition. Edited by Kota Miura.

Another important aspect of writing a macro is its role as a documentation: As the processing becomes complex, we often forget the steps and details of the procedures and the values of parameters that were used for that task. Even if your job is not a repetitive one, a macro written for a task becomes a valuable record of what was done to the image and ensures the *reproducibility* of your image analysis.

3.2.2 Other Ways to Customize ImageJ

This and the next section explain the general capability of extending ImageJ by programming. If you are not interested in general aspects, you can skip these sections.

ImageJ can be extended by writing a Java plug-in. Although you need to know or learn Java programming, this capability affords almost infinite possibilities; you will be able to write any kind of processing/analysis functions you could imagine. Compared to the plug-in development by Java, ImageJ macro language is much easier and lighter but has some limitations, which are worth mentioning here:

1) If you need to process large images or stacks, you might recognize that it is slow. Some benchmarks indicate that a plug-in would be about 40 times faster than a macro.
2) Macro cannot be used as a library.[1)] In Java, once a class is written, this can be used again later for another class.
3) Macro is not efficient in implementing real-time interactive input when the macro function is being executed, for example, if you want to design a program that requires real-time user input to select a ROI interactively. Macro can do such interactive tasks by only closely related macro set with each macro doing each step of interaction.
4) Macro is tightly coupled to GUI (Image Window); so when you want to process images without showing them on desktop, macros are not really an optimal solution.

If you become unsatisfied with these limitations, learning more complicated but more flexible Java plug-in development is recommended.

3.2.3 Comparison with Other Scripting Languages

Besides ImageJ macro, there are several scripting languages that can be used for programming with ImageJ. The bare ImageJ supports JavaScript (Rhino). Recent

1) It is possible to write a macro in a library fashion using the function `eval` and use it from another macro, but this is not as robust and as clear as it is in Java, which is a language designed to be so.

versions of ImageJ (>1.47 m, since March 6, 2013) have included Jython in the menu as well. In the Fiji distribution, you can use the following languages off the shelf [2]:

- JavaScript
- BeanShell
- Jython (Java implemented Python)
- JRuby (Java implemented Ruby)
- Clojure
- Groovy

If you set up an environment by yourself, other languages such as Scala can be used.

Compared to these general scripting languages, ImageJ macro has the following advantages:

- It is easy to learn. ImageJ macro built-in functions are mirrors of ImageJ menu, so scripting is intuitive if you already know ImageJ. Macro recorder is a handy tool for finding out the macro function you need.
- A significant hurdle for coding with general scripting languages is that one must know the *ImageJ Java API* well, meaning that you need to know the fundamentals of Java programming language for using these scripting languages.
- You can have multiple macros in one file (called "macro set"). This is useful for packaging complex processing tasks.

Thus, ImageJ macro language is the easiest way to access the scripting capability of ImageJ.

There are several disadvantages of ImageJ macro compared to other scripting languages. First is its generality. Since others are based on major scripting languages, you do not need to learn a lot if you already know one of them. For example, if you already know Python, it will be easy for you to start writing codes in Jython (*note:* but you also need to know about Java).

The second disadvantage of ImageJ macro is its extendability. Codes that you have written can be recycled only by copying and pasting.[3] With other scripting languages, once you write a code, it can be used from other programs.[4]

Finally, although ImageJ macro processes with a speed comparable to JavaScript and Jython, it is slow compared to Clojure and Scala.

2) As of June 2015.
3) One can also use getArgument() and File related functions to pass arguments from a macro file to the other, but ImageJ macro is not designed to construct a library of functions.
4) Calling other Javascript file from another Javascript file had been difficult, but it became easily possible in the Fiji distribution from March 2012.

3.2.4 How to Learn Macro Programming

In this course, you will encounter many example codes. You will write example codes using your own computer and run those macros. Modifying these examples by yourself is an important learning process as in most cases; this is the way to acquiring programming literacy. There are many excellent macro codes available on the Internet, which can be used as starting points for writing your own code.[5)]

3.2.5 Summary

ImageJ macro radically decreases your workload and it is a practical way to keep your image analysis workflow in text file. Less workload provides us more time for analyzing details of image data. The potential of macro is similar to other scripting languages and Java plug-ins, all adding capability to customize your image analysis. For coding interactive procedures, plug-in works better than macro. Macro cannot be used as a library. Image processing by macro is slower than that by Java written plug-ins.

3.3 Tools

- ImageJ (Fiji distribution).
  - http://fiji.sc
  - We use the Fiji distribution since it has script editor, a handy editor for writing macro.
- ImageJ plug-in: EMBL_sampleimages-1.0.2.jar
  - This plug-in allows you to download sample image stacks. All the image files mentioned in this chapter can be opened by selecting the file name in `[EMBL > Sample Images >]`. Specified image file in the EMBL Web server is then downloaded and thus appears on your desktop.
  - This plug-in can be installed in one of the two following ways:
  - First, use the "Update Sites" function in Fiji. Add "CMCI-EMBL" to your update sites by the updater interface that can be accessed via `[Help > Update Fiji]`.
  - Second, download the plug-in from http://cmci.embl.de/downloads/coursemodules.

5) 200+ macros are available in ImageJ website: http://rsb.info.nih.gov/ij/macros.

3.4 Basics

3.4.1 "Hello World!"

We first try writing a simple macro that prints "Hello World!" in the log window of ImageJ. For this, we use a text editor that comes with Fiji, called "script editor." It has some convenient features such as automatic coloring of macro functions. In programming world, we call this feature "syntax highlighter."

If you are using native ImageJ and not Fiji, it is no problem as there is a simpler but perfectly working text editor. Macros that we write in this textbook works exactly the same in both editors. If you want to use the simple text editor for the following tutorial, its usage is explained after the explanation about Fiji editor (Section 3.4.1.1).

Let's open the "script editor" by `[File -> New -> Script]`. It should look like Figure 3.1. There should be a blank text field where you write your macro. Since the editor allows you to write different scripting languages as well, you should first select the language you are going to use. From script editor's own menu, select `[Language -> IJ1 Macro]`.

Then, write your first macro as shown below (see also Figure 3.1):

```
print("Hello World!");
```

Don't ignore quotation marks, parenthesis, and the semicolon! Syntax highlighter offers automatic coloring of ImageJ functions, because you selected the language "IJ Macro" above. It increases the readability of codes.

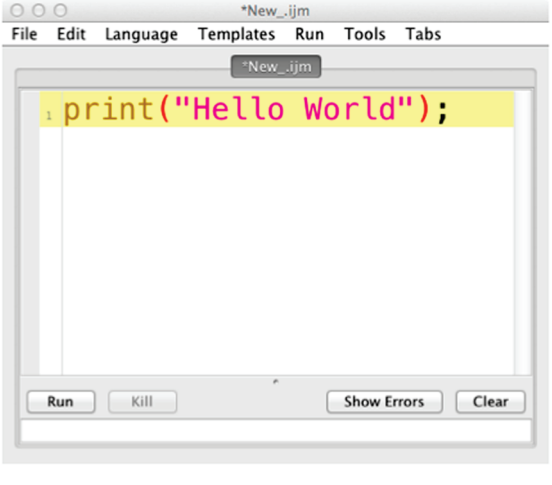


Figure 3.1 Script editor of the Fiji distribution.

Figure 3.2 Hello World output.

Then in the bottom-left corner of the script editor, there is a button labeled "Run." Clicking this, you will see that a log window is created (if it is already there, then it will have a new line) printing "Hello World!" (Figure 3.2). Another way to run the macro is via script editor menu, `[Run -> Run]`. You can use Ctrl-R (Windows) or Command-R (OSX) as well.

Later when you want to start writing another macro, you can just create a new tab by `[File > New]` and then select `[Language -> IJ1 Macro]` again.

3.4.1.1 Simple Text Editor in Native ImageJ

If you are using native imageJ, then the macro editor launches by selecting `[PlugIns -> New -> Macro]` from the menu (Figure 3.3). Please write the following line in the editor:

```
print("Hello World!");
```

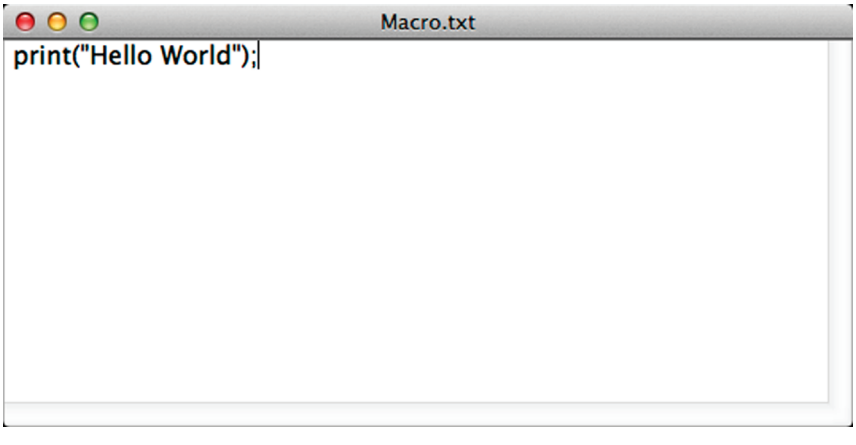


Figure 3.3 Macro editor of ImageJ.

From the menu of the macro editor (in OSX, the menu switches to the editor's own menu when the editor window is active), select [Macros > Run Macro]. You should then see "Hello World!" printed in the log window.

The macro editor has simple debugger function, which is not present in Fiji script editor. Debugger assists you to correct mistakes in the code. ImageJ macro can be written in any text editor such as "Notepad" in Windows, but of course there is no debugger function available in this case.

3.4.1.2 Anatomy of "Hello World!"

Let's see more details of what the single line code we wrote is doing.

`print()` is a built-in macro function that requests ImageJ to take the content within the parenthesis and print that in the "Log" window. This content, which we generally call the *argument* of the function, is an input value given to the function. The output of this function is the printed text in the Log window. Note that when a text is given as an argument, it must be surrounded by double quotes (" ").

Where do we get information as such for other macro functions? The best reference for ImageJ macro functions is in the ImageJ Web site.[6] For example, you can find definition of `print("")` function on the Web site as mentioned below.

> **print(string)**
> Outputs a string to the "Log" window. Numeric arguments are automatically converted to strings. The print() function accepts multiple arguments. For example, you can use print(x,y,width, height) instead of print(x+" "+y+" "+width+" "+height). If the first argument is a file handle returned by File.open(path), then the second is saved in the referred file (see SaveTextFileDemo).
>
> Numeric expressions are automatically converted to strings using four decimal places, or use the d2s function to specify the decimal places. For example, print (2/3) outputs "0.6667", but print(d2s(2/3,1)) outputs "0.7." . . .

As `print` can do many things, its explanation is extraordinarily long; however, a careful reading of it will save your time in future as you will be well acquainted with the wide spectrum of things that the `print` function can do, for example, directly save text as a file.

Macro can be saved as a file. In the editor, do `[File -> Save]`. Just save the file wherever you want in your file system. When you want to use the macro again, load the macro by `[File > Open]`.

6) http://rsbweb.nih.gov/ij/developer/macro/functions.html.

Exercise 3.1

Add another line `"print("\\Clear");"` before the first line (below, code 1.51. Don't forget the semicolon at the end!).

```
1 //code 1.51
2 print("\\Clear");
3 print("Hello World!");
```

code/code01_51.ijm

Then test another macro as well: Put the same line after "Hello World!." What happened? Any difference in the behavior?

```
1 //Code 1.76
2 print("Hello World!");
3 print("\\Clear");
```

code/code01_76.ijm

Answer:
The first code prints "Hello World!", while the second code prints nothing. This is because `print("\\Clear")` is a command that clears the Log window. In the first code, "Hello World" is printed after the window clearing; and in the second case, the Log window is wiped out right after the printing of "Hello World". Effectively, it looks like nothing has happened.

Exercise 3.2

Try modifying the third line in code 1.51 and check that the modified text will be printed in the "Log" window.

Multiple macros can exist in a single file. We call this *macro sets*. To distinguish each macro, each of them should have a specific name. For this, each macro should start with a special word "macro" followed by the name of the macro, and then a pair of curly braces must encapsulate its macro functions. See the code below:

```
1 macro "print_out"  {
2   print("Hello World!");
3 }
4
5 macro "print_out2"  {
6   print("Bye World!");
7 }
```

code/code01_8.ijm

Exercise 3.3

Modify the code you already wrote in the script editor to wrap it within a pair of macro bounds, the curly braces ({}). Then copy and paste the same under the first macro. The second macro should be modified to have a different name. In the example shown in Figure 3.4, the second macro is named "print_out2". When macro is properly declared in this way, you can install the macro to have it as a menu item. To do so, in the editor menu select

[Run -> Install Macro]).

In the main menu, you will not be able to see the macro names under `[Plugins > Macros >]` (Figure 3.5).

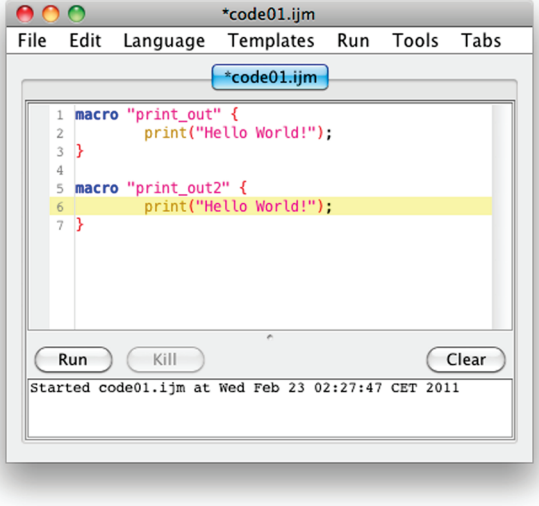


Figure 3.4 Macro set.

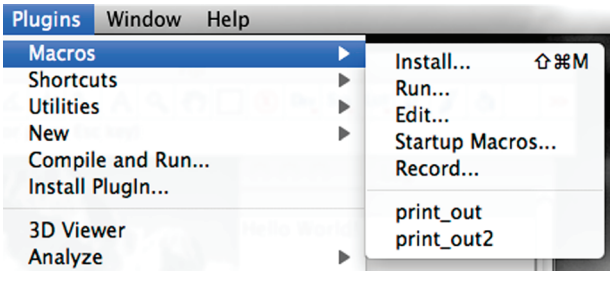


Figure 3.5 Custom macro commands in ImageJ menu.

3.4.2 Variables and Strings

Texts such as "Hello World!" can be represented by a variable.[7] Let's understand this by examining a short macro below:

```
1 //Code 2
2 macro "print_out 2"  {
3   text = "Hello World";
4   print( text);
5   text = "Bye World";
6   print( text);
7 }
```

code/code02.ijm

`text` is a "String Variable" or simply a "String." ImageJ prepares a memory space for this variable, and you can change the content by redefining the content. Two (or may be more) variables can be used to construct another variable:

```
1 //Code 3
2 macro "print_out 3"  {
3   text1 = "Hello";
4   text2 = "World!";
5   text3 = text1 + text2;
6   print(text3);
7 }
```

code/code03.ijm

The above operation concatenates content of `text2` to the content of `text1` and produces a third variable `text3` that holds the result of concatenation. It should be noted here that macro has two ways of usage for +. What we tested in above operation is "concatenation." Another usage is "addition" in the next section.

Exercise 3.4

Add more string variables and make a longer sentence.
Answer:
One example could be as shown in the Figure 3.6.

It is also possible to store a number in a variable. For example,

```
text = 256;
```

7) There is no declaration of types, such as number or string, in ImageJ macro.

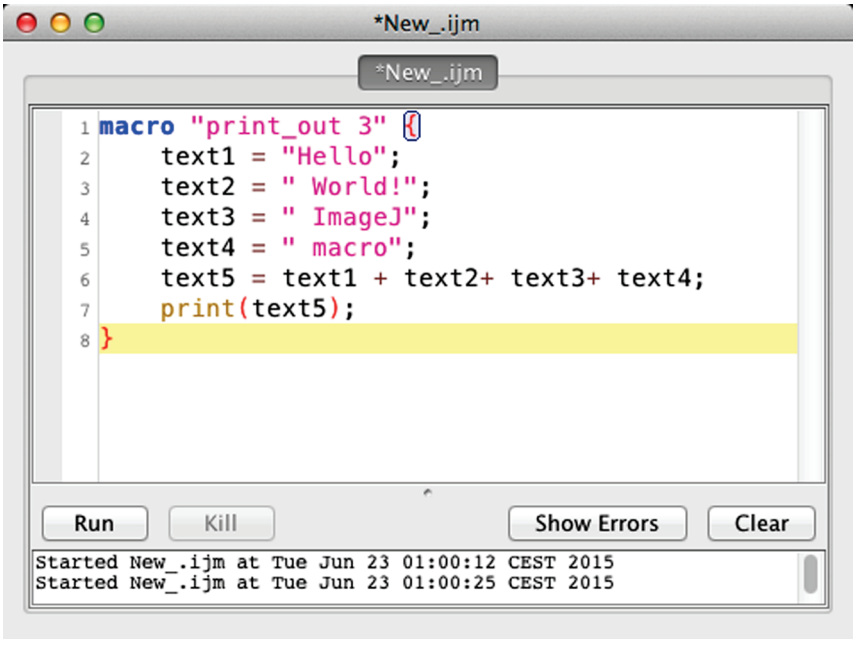


Figure 3.6 Concatenating many strings.

With this assignment, the variable is now a "numerical variable" or simply a "variable." In other programming languages such as C or Java, difference between numbers and characters matters a lot. In ImageJ macro, you do not have to take into account whether the variable is a number or a string (we call them "types") as the types are defined automatically by the type of value provided for a variable; this makes the macro coding light and easy. However, since types are implicitly defined without declaration, it can lead to simple mistakes such as type mismatching. So keep in mind that the difference in types *does* matter, but they are not shown in the code. We will see an example of such confusion, and also a way to avoid the confusion.

Test the following macro to see how the numerical variable works:

```
1  //Code 4
2  macro "print_out_calc"  {
3    a = 1;
4    b = 2;
5    c = a + b;
6    print(c);
7    print(a + "+"+ b + "="+c);
8    txt=""+a + "+"+ b + "="+c;
9    print(txt);
10 }
```

code/code04.ijm

Did you get some results printed out? It should, but you should read the code carefully as there is a small trick in this code. This trick is something special in ImageJ macro language compared to other general scripting languages.

You might have noticed a strange expression in line 8, that is, in the way it assigns the variable `txt`. It starts with double quotation marks.

```
txt= "" + a + "+" + b + "=" + c;
```

Seemingly, this looks meaningless. If you define the variable `txt` without the first "useless" quotation marks, then it will be like

```
txt= a + "+"+ b + "=" + c;
```

Theoretically, this should work, since the double quotes do not have any content and so its presence should be meaningless. But if you try to run this, which seems to be a straightforward assignment, ImageJ returns an error message (Figure 3.7).

This is because when ImageJ scans through the macro from top to bottom, line-by-line, it reaches the line for the assignment of the variable `txt` and first sees the variable `a` and interprets that `txt` should be a numerical variable (or function), since `a` is known to be a number as it was defined so in one of the lines above. Then ImageJ goes on interpreting rightward thinking that this is math. Then it finds a "+" string variable within a numerical function, which is a character that ImageJ cannot interpret as a mathematical operator and so it returns an error message. The macro aborts.

To overcome this problem, the programmer can tell ImageJ that *txt* is a string function at the beginning of the assignment by putting a set of double quotes. This tells the interpreter that this assignment is a string concatenation assignment and not a numerical assignment. ImageJ does handle numerical values within string function, so the line is interpreted without problem and prints out the result successfully. Note that such confusion of string and numerical types is rarely seen in general scripting languages and is specific to ImageJ macro language.

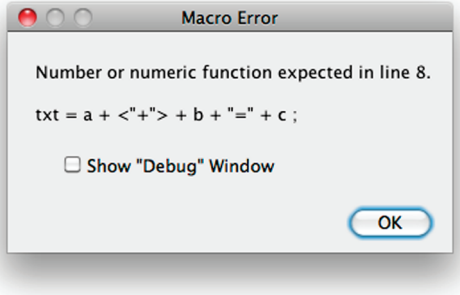


Figure 3.7 Error with variable assignment.

Exercise 3.5

Modify code 4, so that the calculation involves subtraction (–), multiplication (*), and division (/).

Answer:

Add the following lines to print results of calculations. Note that the arguments of `print` are separated by comma, which will be space-separated `text` in the output:

```
sub = a - b;
mul = a * b;
div = a / b;
print(a, "-", b, "=", sub);
print(a, "*", b, "=", mul);
print(a, "/", b, "=", div);
```

3.4.3 Recording ImageJ Macro Functions

There are many commands in ImageJ as you can see them by exploring the menu tree. In ImageJ native distribution, there are about 500 commands. In the Fiji distribution, there are 900+ commands. Some plug-ins are not macro-ready, but except for these special cases, almost all of these commands can be accessed by built-in macro functions. We then encounter a problem: How do we find a macro function that does what we want to do?

To show you how to find a function, we write a small macro that creates a new image, adds noise, blurs this image by Gaussian blurring, and then thresholds the image. There is a convenient tool called *command recorder*. Do `[PlugIns -> Macros -> Record . . . ]`. A window shown in Figure 3.8 opens.

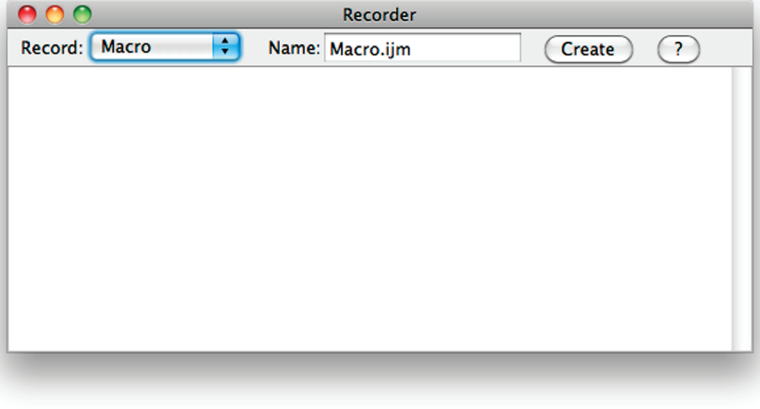


Figure 3.8 Command recorder.

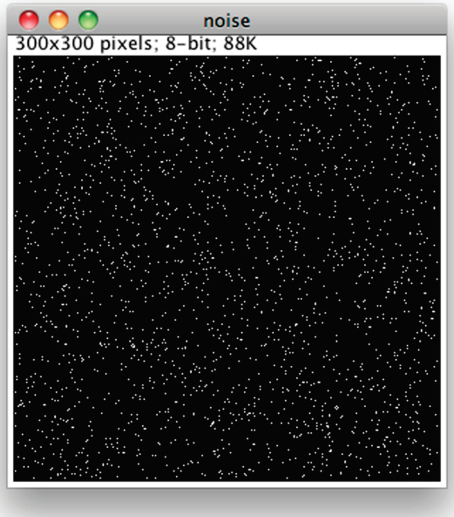


Figure 3.9 A demo image of recording macro.

All the menu commands that you execute will be printed out as a history of macro functions in this window. For composing a macro using this recorder, we first do the processing manually from the menu as follows:

- Prepare a new image using `[File -> New]` command. The size of the image can be anything.
- Then do `[Process -> Noise -> Salt and Pepper]` (Figure 3.9).
- `[Process -> Filters -> Gaussian Blur]` (use Sigma = 2.0).
- `[Image -> Adjust -> Threshold . . . ]`. Toggle the slider to make signals red. Check "Dark Background" and then click "Apply."

Now, check the command recorder window. It should now look like Figure 3.10. Each line is a macro function that corresponds to a menu command you selected.

These texts generated in the recorder can be used as it is in your macro. You can copy and paste them.[8] Compose a macro as given below by copying and pasting the macro functions in the recorder. Delete the lines that are commented out (lines that begin with "//" are lines that are skipped by the macro interpreter).

8) In case of OSX, you might probably need to click "Create" button to generate a duplicate of macro functions in a new script window. Then you can copy the macro functions from there.

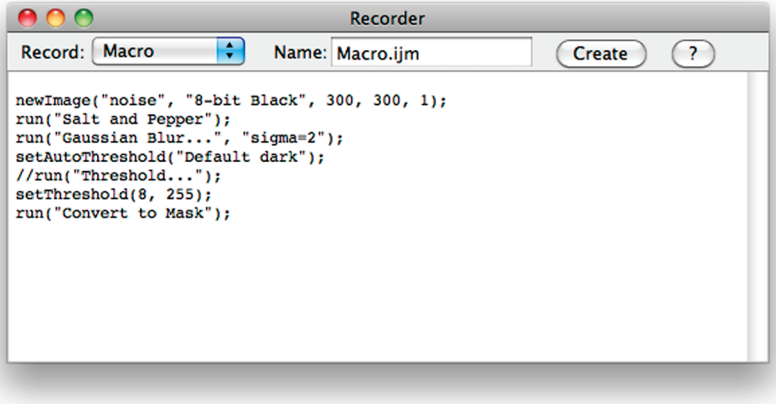


Figure 3.10 Macro recorder after some lines recorded.

```
1 //Code 6.9
2   newImage("test", "8-bit Black", 300, 300, 1);
3   run("Salt and Pepper");
4   run("Gaussian Blur...", "radius=2");
5   setThreshold(32, 100);
6   run("Convert to Mask");
```

code/code06_9.ijm

Run the macro! . . . I hope you must be amazed with the power of macro recorder! Now, you can simply add a line at the top and at the bottom to package this in a named macro by curly braces. This is optional in the current case, but it is always good to keep your macro packaged since the boundary of the macro becomes clear.

```
1 //Code 7
2 macro "GB2_Thr"  {
3   newImage("test", "8-bit Black", 300, 300, 1);
4   run("Salt and Pepper");
5   run("Gaussian Blur...", "radius=2");
6   setThreshold(32, 100);
7   run("Convert to Mask");
8 }
```

code/code07.ijm

The third line in the above macro has a function `newImage()`. This function creates a new image. It has five arguments (in coding jargon, we say there are "five arguments"). To know what these arguments are, the quickest way is to read the Built-In Macro Function page in ImageJ Web site.[9] In case of the function `newImage`, the description looks like this.

9) http://rsbweb.nih.gov/ij/developer/macro/functions.html.

newImage(title, type, width, height, depth)
Opens a new image or stack using the name title. The string type should contain "8-bit," "16-bit," "32-bit," or "RGB." In addition, it can contain "white," "black," or "ramp" (the default is "white"). As an example, use "16-bit ramp" to create a 16-bit image containing a grayscale ramp. Width and height specify the width and height of the image in pixels. Depth specifies the number of stack slices.

Using this information, you can modify the macro to change the size of the image.

Exercise 3.6

Modify code 7 and try changing the size of window to be created.

Other optional lines you can add to the macro are "comments." This does not affect the macro, but adding some comment about what the macro does helps you to understand what the macro is doing when you open the file some time later. There are two ways to add comment. One is the *block comment*. Texts bounded by/* and */will be ignored by interpreter. Another is the *line comment*. Texts in a line starting with double slash(//)will be ignored by the interpreter. Below is an example of commenting code 07.

```
1  //Code 7.1
2  /*
3  This macro creates binary image with randomly
       positioned dots.
4  */
5  macro "GB2_Thr" {
6    //creates a new image window
7    newImage("test", "8-bit Black", 300, 300, 1);
8    //add noise
9    run("Salt and Pepper");
10   //blur the image
11   run("Gaussian Blur...", "radius=2");
12   //binarize the image
13   setThreshold(32, 100);
14   run("Convert to Mask");
15 }
```

code/code07_1.ijm

3.5 Loops and Conditions

In many cases, we want to iterate certain processing steps many times (see "Loops" in Figure 3.11), or we want to limit some of the process in the program only for certain situations (see "Conditions:" in Figure 3.11). In this section, we will learn how to include these loops and conditional behaviors into macro.

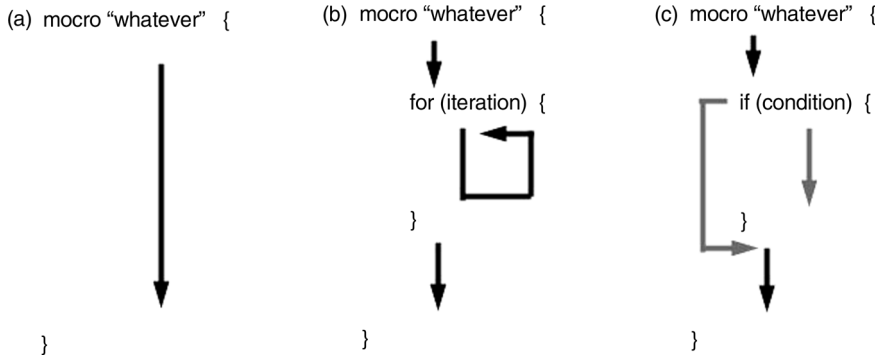


Figure 3.11 Schematic view of conditions and loops. Straightly top to bottom, line-by-line processing (a) and macro with loops (b) and with a condition (c).

3.5.1 Loop: For-Looping

Here is a simple example of macro using for-loop. Write the macro in your editor and run it.

```
1 //Code 8.9
2
3 for( i = 0;  i < 5;  i += 1)  {
4     print( i + ": " + "whatever");
5 }
```

code/code08_9.ijm

The result should look like Figure 3.12.

- Line 3 `for(i = 0; i < 5; i += 1)` sets the number of loops. Three parameters are required for "for-loop." The first parameter defines the variable used for

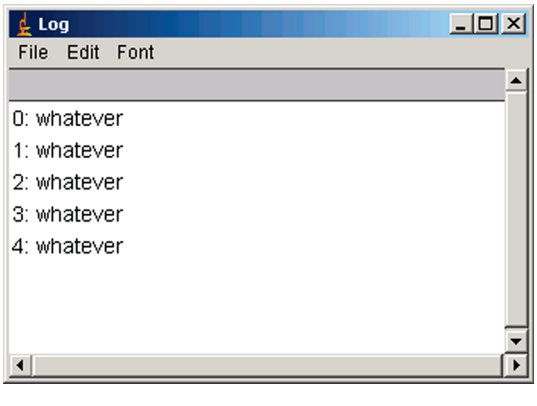


Figure 3.12 Code 8.9 output in Log window.

the counting loop and its initial value (`i = 0`). The second parameter sets the condition for exiting from the loop (`i < 5`). Third parameter sets the step size of i, meaning that how much value is added per loop (`i += 1`, can also be subtraction, multiplication, division, for example, `i -= 1`). Spaces between variables, numbers, operators, and separators (e.g., semicolon and parenthesis) can be ignored and they could be written continuously. Macro runs without those spaces. However, this is not recommended for keeping a better readability of the code. Don't try to rush, make spaces!

- After this `for( . . . ; . . . ; . . . )` statement, there is a brace ({) at the end of line 3 and the second one (}) in line 5. These curly braces tell ImageJ to loop macro functions in between, so the function in line 4 will be iterated according to the parameters defined in the parenthesis of `for`. Between braces, you can add as many more lines of macro functions as you want, including inner `for`-loops and `if-else` conditions.

So when the macro interpreter reaches line 3 and sees `for(`, it starts looking inside the parenthesis and defines that the counting starts with 0 using a variable `i`, and then line 4 is executed. The macro prints out "0: whatever" using the content of `i`, string: and the string variable `txt`. Then in line 5, interpreter sees the boundary } and goes back to line 3 and adds 1 to i (because of `i+=1`). i = 1 then, so `i<5` is true. The interpreter proceeds to line 4 and executes the macro function and prints out "1: whatever." Such looping will continue until i = 5, since only by then `i<5` is no longer true, so interpreter exits from the for-loop.

Exercise 3.7

1. Change the first parameter in `for(i=0;i<5;i+=1)` so that the macro prints out only 1 line.
2. Change the second parameter in `for(i=0;i<5;i+=1)` so that the macro prints out 10 lines.
3. Change the third parameter in `for(i=0;i<5;i+=1)` so that the macro prints out 10 lines.

Answer:

1. `for(i=4;i<5;i+=1)`
2. `for(i=0;i<10;i+=1)`
3. `for(i=0;i<5;i+=0.5)`

3.5.1.1 Stack Analysis by For-Looping

One of the frequently encountered tasks is image stack management, such as measuring dynamics or multiframe processing. Many ImageJ functions work with only single frame within a stack. Without macro programming, you need to

execute the command while you flip the frame manually. Macro programming enables you to automate this process. Here is an example of measuring intensity change over time.[10)]

```
1  //Code 10
2  macro "Measure Ave Intensity Stack" {
3    frames=nSlices;
4    run("Set Measurements...", "  mean  min integrated
         redirect=None decimal=4");
5    run("Clear Results");
6    for(i=0; i<frames; i++) {
7      currentslice=i+1;
8      setSlice(currentslice);
9      run("Measure");
10   }
11 }
```

code/code10.ijm

- *Line 3:* `nSlices` is a macro function that returns the number of slices in the active stack.
- *Line 4:* Sets measurement parameters, from the menu would be `[Analyze > Set measurements . . . ]`. In this case, "mean min integrated" is added as part of the second argument. "mean" is the mean intensity, "min" is the minimum intensity, and "integrated" is integrated density (total intensity). These keys for measured parameters can be known by using the command recorder. You do not have to care for now about the "redirect" argument. "decimal" is the number of digits to the right of the decimal point in real numbers displayed in the results table.
- *Line 5:* Clears the results table.
- *Lines 6–9:* These are loops. Loop starts from count i=0, and ends at i=frame-1. `i++` is another way of writing `i = i + 1`, so the increment is 1.
- *Line 7:* Calculates the current frame number.
- *Line 8:* `setSlice` function sets the frame according to the frame number calculated in line 6.
- *Line 9:* Actual measurement is done. Result will be recorded in the memory and will be displayed in the Results table window.

Open an example stack *1703-2(3s-20s).stk* (Figure 3.13). This is a short sequence of FRAP analysis, so the edge of the one of the cells is bleached and then fluorescence signal at that bleached position recovers by time. Select the

10) What we write as macro here can be done with a single command `[Image > Stacks > Plot Z-Profile]`, but this only measures intensity. If you want to measure other values such as the minimum intensity, a macro should be written.

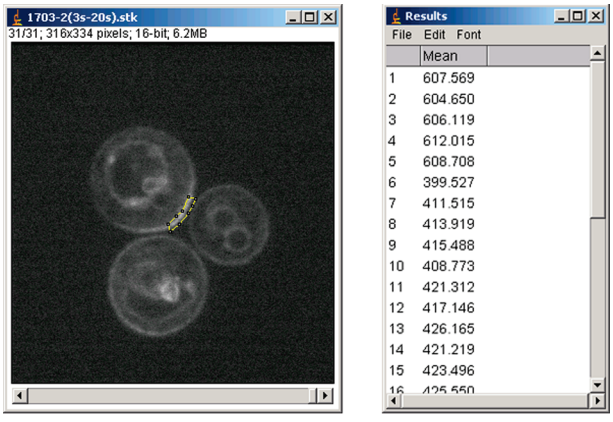


Figure 3.13 Measuring stack intensity series. (a) Setting a segmented ROI at the FRAPped area. (b) Results of measuring mean intensity dynamics.

frapped region by ROI tool (as in Figure 3.13). Execute the macro. Results will be printed in the Results window (see the table on the right of the figure, it shows only the "Mean" column as only "Mean Intensity" was selected in the measurement option).

Measurement parameters can be added as argument by modifying line 4 in code 10.

Exercise 3.8

Modify code 10 to include more measurement parameters (choose several as you wish), and test the macro. Check the results (Figure 3.14).

Answer:

"Set Measurement" can be added with more parameters to be measured, and the digits after the decimal point can be increased by increasing the number after "decimal=." For example,

```
run("Set Measurements...", "area mean standard modal
    min centroid center perimeter bounding integrated
    median stack redirect=None decimal=5");
```

3.5.2 Loop: While-Looping

Another way of letting a part of macro to loop is *while*-statement. In this case, iteration is not defined strictly. Looping continues until certain condition is met. As soon as the condition is violated, macro interpreter goes out from the loop.

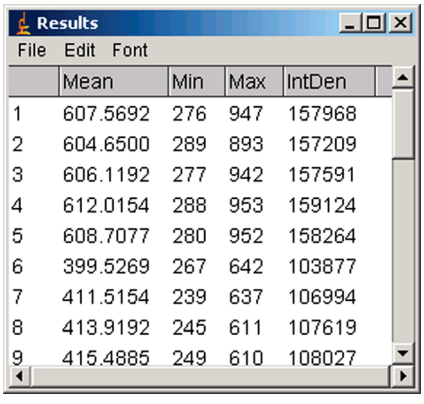


Figure 3.14 An example result after adding more measurement parameters.

3.5.2.1 Basics of While Statement

Here is a simple example of macro using `while`:

```
1 //Code 11
2 macro "while looping1" {
3   counter=0;
4   while (counter<=90) {
5     print(counter);
6     counter = counter + 10;
7   }
8 }
```

code/code11.ijm

This macro prints out characters 0–90 with a 10 increment (Figure 3.15).

- *Line 3:* The macro interpreter first assigns 0 to the counter.
- *Line 4:* The interpreter evaluates if the counter value is less than or equal to 90. Since counter is initially 0, the evaluation results in "true" and the interpreter moves into the loop.
- *Line 5:* Printing function is executed.
- *Line 6:* Counter is added with 10.
- *Line 7:* The interpreter realizes the end of "while" boundary and goes back to line 4. Since counter= 10, which is <= 90, line 5 is again executed and so on. When counter becomes 100 in line 6 after several more loops, counter is no longer <=90. So the interpreter goes out from the loop and moves to line 8. Then the macro is terminated.

Line 6 can be written in the following way as well:

```
counter += 10;
```

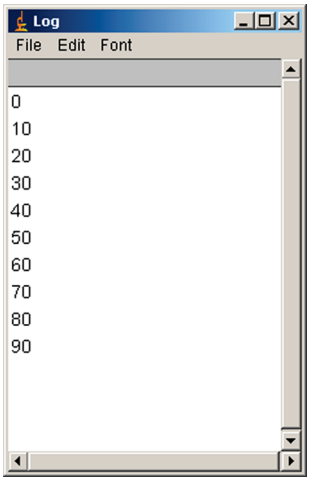


Figure 3.15 Output of code 11.

This means that "counter" is added with 10. Similarly, subtracting 10 from counter is

```
counter -= 10;
```

Multiplication is

```
counter *= 10;
```

Division is

```
counter /= 10;
```

If the increment is 1 or −1 (counter +=1 or counter−=1), then one can also write them as

```
counter++;
 or
counter--;
```

The two last macro functions are said to work faster than +=1 or −=1, but I myself do not see much difference. Computers are fast enough these days.

Exercise 3.9

1. Try changing code 11 so that it uses "+=" sign.
2. Change code 11 so that it uses "++" sign, and prints out integers from 0 to 9.

Answer:
(1) Change line 6 to `counter += 1;`. (2) Change line 4 to `while (counter<=9)` and line 6 to `counter++`.

Exercise 3.10

Change line 4 of code 11 to `while (counter <0)` and check the effect.

Answer:
Nothing will be printed out.

Evaluation of `while` condition can also be at the end of the loop. In this case, `do` should be stated at the beginning of the loop. With `do-while` combination, the loop is always executed at least once, regardless of the condition defined by `while` since macro interpreter reads lines from top to bottom. Write the following code:

```
1 //Code 11.5
2 macro "while looping2" {
3   counter=0;
4   do {
5     print(counter);
6     counter += 10;
7   } while (counter<0);
8 }
```

code/code11_5.ijm

In this example, the exit condition for going out from looping is `counter < 0`, and the initial value of `counter` is 0, which does not satisfy that looping condition. Since this evaluation occurs only after the looping part is executed for the first time, the macro still prints out a line before it exits from the loop.

Condition for the while-statement can be various. Here is a small list of comparison operators.

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal
!=	Not equal to

Exercise 3.11

Modify code 11 so that the macro prints out numbers from 200 to 100, with an increment of −10.

Answer:

There can be slightly various ways to do this modification, but here is one way.

```
macro "while looping1" {
  counter=200;
  while (counter>=100) {
    print(counter);
    counter -= 10;
  }
}
```

3.5.2.2 Why Is There While-Loop?

A question that is often raised with the while-loop is why do we have two types of loops, the for-loop and the while-loop. Answer to this question is that they have different flexibility. The for-loop is rather solid and the while-loop is more flexible. In the example code below, the user is asked for a correct number and if the answer is wrong, the question is asked five times repeatedly. The number of loop is not determined by the programmer; it is rather determined interactively when the code is run. We will study the branching of the program based on if–else in the next section.

```
macro "flexible loop by while" {
  answer_is_wrong = true;
  imagej_first_release = 1997;
  trial = 5;
  while (answer_is_wrong) {
    answer = getNumber("In which year did the first version
      of ImageJ released?", 1900);
    if (answer == imagej_first_release){
      answer_is_wrong = false;
      showMessage("CORRECT! The year" + imagej_first_release);
    } else {
      showMessage("NO. try again: trials left:" + trial);
      trial-;
    }
    if (trial < 1)
      answer_is_wrong = false;
  }
}
```

code/code11_6.ijm

Writing a similar code using the for-loop is possible, but the code becomes tricky. Below is the for-loop version of the above code:

```
1  macro "flexible loop by for" {
2    imagej_first_release = 1997;
3    trial = 10;
4    for (correct = 0; correct < 1;)  {
5      answer = getNumber("In which year did the first
           version of ImageJ released?", 1900);
6      if (answer == imagej_first_release) {
7        showMessage("CORRECT! The year" +
             imagej_first_release);
8        correct++;
9      } else {
10       showMessage("NO. try again: trials left:" + trial);
11       trial--;
12     }
13     if (trial < 1)
14             correct++;
15   }
16 }
```

code/code11_7.ijm

Note that the third argument of for-loop is missing. Since the variable `correct` does not change as long as the answer is wrong, we leave it as neither incrementing nor decrementing. In such case, we can leave the third argument vacant.

3.5.3 Conditions: If–Else Statements

3.5.3.1 Introducing If–Else

A macro program can have parts that are executed depending on some conditions. Here is an example of macro with conditions (Figure 3.16):

```
1  //Code 12
2  macro "Condition_if_else 1"{
3    input_num = getNumber("Input a number", 5);
4    if (input_num == 5) {
5      print(input_num+ ": The number is 5 ");
6    }
7  }
```

code/code12.ijm

- *Line 3:* The macro asks user to input a number and the number is substituted to the variable `input_num`.
- *Line 4:* Content of `input_num` is evaluated. If `input_num` is equal to 5, line 5 is executed and prints out the message in the Log window. Otherwise, macro interpreter jumps to line 7, and ends the operation. By adding "else" that will be executed if `input_num` is not 5, the macro prints out message in all cases (see code 12.6 for this if–else case).

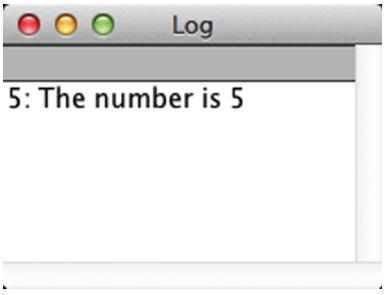


Figure 3.16 Output of code 12.

- *Line 4:* We used double equal signs for evaluating the values on right and left sides (e.g., `if (a==5)`). Note that the role of the sign = is different from assignments, or substitution (e.g., `a = b + c`).

Now, we examine the content within the parentheses after "if" in more detail. Write the following code in your script editor and run it:

```
1 a = (5==5);
2 print(a);
```

code/code12_1.ijm

The output in the Log window should be *1* indicating that "`(5 == 5)`" is *1*. Next, modify the code like below and run it:

```
1 a = (5 == 4);
2 print(a);
```

code/code12_2.ijm

The output is now 0, indicating that "`(5 == 4)`" is 0. Here the double equal signs == are comparing the numbers on the left and right sides, and if the numbers are the same, it returns 1 and if they are not the same, it returns 0. In fact, 1 and 0 are representing *true* (= 1) or *false* (= 0), the *Boolean values*.

We can also test if they are *not* equal. For this, replace == by !=.

```
1 a = (5 != 4);
2 print(a);
```

code/code12_3.ijm

Run the code above, and it returns 1, because 5 is *not* 4 and this is true. Now, you can introduce the *if* again as follows:

```
1 if (5 != 4){
2   print("true");
3 }
```

code/code12_35.ijm

In the parenthesis after "if," there is obvious *true* statement (5 is not 4). This is true, so the macro function bounded by curly braces is executed, which is to print out "true" in the Log window.

Try changing the line 2 to `if (5 == 4)`. Running this prints nothing in the Log window, because 5 is not 4 (FALSE!), so the macro function in line 3 is ignored. To avoid such ignorant no-output behavior, you can add "else" as follows:

```
1 if (5 == 4){
2   print("true");
3 } else {
4   print("false!");
5 }
```

code/code12_4.ijm

The code also works with the direct true or false declaration within the if parenthesis. Try the following code:

```
1  if (0){
2    print("true");
3  } else {
4    print("false!");
5  }
6
7  if (false){
8    print("true");
9  } else {
10   print("false!");
11 }
```

code/code12_5.ijm

The above prints two lines of "false!" in the Log window. You can replace the if parenthesis values to 1 and true to check that it works as well.

By now, it is probably pretty clear to you what is going on in the code below:

```
1 macro "Condition_if_else 2"{
2   input_num = getNumber("Input a number", 5);
3   if (input_num == 5) {
4     print(input_num+ ": The number is 5 ");
5   } else {
6     print(input_num+ ": The number is not 5 ");
7   }
8   print("--------------");
9 }
```

code/code12_6.ijm

3.5.3.2 Complex Conditions

In many cases, you might need to evaluate the condition of multiple variables at once. For such demands, several different comparisons can be combined by using following Boolean operators:

&& Boolean AND

|| Boolean OR

Let's first test what these symbols do by directly using `true` and `false` in macro:

```
1 a = true;
2 b = true;
3 if (a && b){
4   print("&& both true")
5 }
6
7 if (a || b){
8   print("|| one of them or both is true")
9 }
```

code/code12_65.ijm

When you run this code as it is, lines 4 and 8 both are executed and print the messages. For the first `if` parenthesis, `&&` operator tests if both sides are true. If both are indeed true, it returns true (1), and that is the case above. If one of them or both are false, then `&&` operator returns false(0).

On the other hand, in the second if parenthesis, || operator tests if one of the two sides is true. Since both are true in the above code, OR operator returns true as it requires at least one of them to be true. Only when both sides are false, the returned value becomes false (0).

> **Exercise 3.12**
>
> Adjust the values of a and b in code 12_65 to `true` or `false` and compose other three possible combinations (e.g., `a = true`, `b = false` will print only one line). Check the output. Next, change the values of `a` and `b` to 0 and/or 1 and check the results.

Below is a more realistic example (although not very useful), an extended version of code 12_6:

```
1 //Code 12.75------------------------------
2
3 macro "Condition_if_else 3"{
4   input_num1 = getNumber("Input a number 1", 5);
5   input_num2 = getNumber("Input a number 2", 6);
6   message0 = ""+input_num1 + ","+input_num2;  //use
        this string four times
7   if ( (input_num1==5) && (input_num2==6))  {
```

```
8       print(message0+ ": The parameter1 is 5 and the
            parameter2 is 6");
9     } else {
10      if (input_num1!=5) && (input_num2!=6) {
11        print(message0 + ": The parameter1 is not 5 and
                the parameter2 is not 6");
12      } else {
13        if (input_num2==6) {
14          print(message0 + ": The parameter1 is NOT 5 but
                the parameter2 is 6");
15        } else {
16          print(message0 + ": The parameter1 is 5 but the
                parameter2 is NOT 6");
17        }
18      }
19    }
20  }
```

code/code12_75.ijm

- Lines 3 and 4 ask user to input two parameters.
- Line 5 is for setting a string variable, to abbreviate a long string assignment that appears four times in the macro.
- Line 6 evaluates these input parameters by comparing each of them separately, but the decision is made by associating two decisions with `&&`.
- Line 9, != compares left and right sides of the operators and returns true if they are *not* equal.

From lines 10 to 17, there are several layers of conditions. Macro programmer should use tab shifting for deeper condition layers as above for the visibility of code. Easy-to-understand code helps the programmer to debug afterward, and also for other programmers who might reuse the code.

3.6 Advanced Topics

3.6.1 User-Defined Functions

As your code becomes longer, you will start to realize that similar processing or calculation appears several times in a macro or through macro sets. To simplify such redundancy, one can write a separate `function` that works as a module for macros.

For example, you have a simple code as follows:

```
1  //Code 15
2  macro "addition" {
3    a = 1;
4    b = 2;
```

```
5    c = a + b;
6    print(c);
7  }
```

code/code15.ijm

In this case, it should be easy for you to expect that this macro will print out "3" in the Log window. From this macro, we can extract part of it and make a separate function.

```
1  //Code 15.1
2  function ReturnAdd(n, m) {
3    p = n + m;
4    return p;
5  }
```

code/code15_1.ijm

This is not a macro, but is a program that works as a unit. Functions can be embedded in macro. `ReturnAdd` in code 15.1 is the name of the function, and the following (`n`, `m`) are the variables that will be used in the function. Within the function, `n` and `m` will be added and the result of which is substituted into a new variable p. `return p` in line 4 will return a value as an output of the function. We call such custom-made function as "user-defined function." Using this function, code 15 can be rewritten as follows:

```
1  //Code 15.2
2  macro "addition with function1" {
3    a = 1;
4    b = 2;
5    c = ReturnAdd(a, b);
6    print(c);
7  }
8  //Code 15.1
9  function ReturnAdd(n, m) {
10   p = n + m;
11   return p;
12 }
```

code/code15_2.ijm

It can be written in a more simple form by nesting the custom-made function within ImageJ native function `print()`:

```
1  //Code 15.3
2  macro "addition with function2" {
3    a = 1;
4    b = 2;
5    print(ReturnAdd(a, b));
```

```
6  }
7  //Code 15.1
8  function ReturnAdd(n, m) {
9    p = n + m;
10   return p;
11 }
```

code/code15_3.ijm

Macro interpreter reads the macro line-by-line. When the interpreter sees `ReturnAdd(a, b)`, the interpreter first tries to find the function within the ImageJ built-in function. If it is not there, the interpreter looks for the function within the same macro file. Here is how it looks like: a macro that uses a function (Figure 3.17).

In this simple case, you might not realize the advantage of the user-defined function, but you will start to enjoy its power once you start writing longer codes. The following are the advantages of using `function`:

1) Once written in a macro file, it can be used as a single-line function as many times as you want in the macro file. This also means that if there is a bug, fixing the function solves the problem in all places where the function is used.
2) Long codes can be simplified to an explicit outline of events. For example:

```
macro "whatever" {
      function1;
    function2;
    function3;
}
```

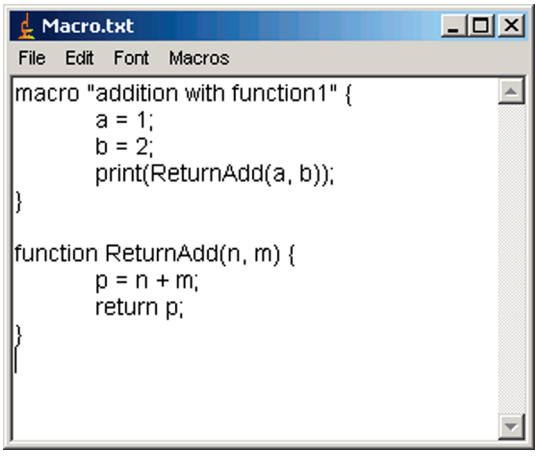


Figure 3.17 A macro file with function.

3.6.2 String Arrays

Array is a powerful tool. Before delving into how to use it, here is an easy explanation. Imagine that an array is a stack of boxes. Boxes can contain either numbers or strings. For instance, you have a following list of strings:

Heidelberg, Hamburg, Hixton, Grenoble, Monterotondo

In this case, an array "EMBL" can be prepared and each array element can contain one of these five strings (Figure 3.18).

If you want to retrieve some name from the array, you may refer to the address within the array. So EMBL[0] will be Heidelberg, EMBL[4] will be Monterotondo, and so on. In such a way, files names contained in a folder can be listed and stored, or *x*–*y* coordinates of free-hand ROI can be stored for further use.

Here is a macro using the EMBL array example:

```
1  //Code 20
2  macro "EMBL array" {
3    EMBL = newArray(5);
4    EMBL[0] = "Heidelberg";
5    EMBL[1] = "Hamburg";
6    EMBL[2] = "Hixton";
7    EMBL[3] = "Grenoble";
8    EMBL[4] = "Monterotondo";
9    address = getNumber("which address [0-4]?", 0);
10   if ((0<=address) && (address<4)) {
11     print("address"+address+" -> "+EMBL[address]);
12   } else {
13     print("That address is somewhere else not EMBL");
14   }
15 }
```

code/code20.ijm

Figure 3.18 EMBL array.

- Line 3 uses a function that creates a new array (`newArray()`), defined by a parameter for number of array elements (in the example case it is 5) and its name `EMBL`.
- From line 4 to line 8, each array from positions 0–4 will be filled with names (array starts with 0th element).
- Line 9 asks the user to input the address (position) within the array. Then this input address is examined if the address exists within the `EMBL` array in line 10. `EMBL.length` returns the number of "boxes" within the array. If this is satisfied, then line 10 prints out the string in that address.

Array can be created and initialized with actual values at the same time, so lines 3–8 can be written as follows:

```
EMBL = newArray("Heidelberg","Hamburg","Hixton",
    "Grenoble","Monterotondo");
for (i = 0; i < EMBL.length; i++)
    print(EMBL[i]);
```

3.6.3 Numerical Array

Array can also contain numerical values, and this way of usage is more common when you do image analysis. Here is a simple example of numerical array that prints out intensity profile along the selected line ROI.

```
1  //code 20.5
2  macro "get profile and printout" {
3    if (selectionType() !=5) exit("selection type must be
         a straight line ROI");
4    tempProfile=getProfile();
5    output_results(tempProfile);
6  }
7  function output_results(rA) {
8    run("Clear Results");
9    for(i = 0; i < rA.length; i++) {
10     setResult("n", i, i);
11     setResult("intensity", i, rA[i]);
12   }
13   updateResults();
14 }
```

code/code20_5.ijm

- *Line 3:* Checks if the selection type is a straight line ROI. If not, macro terminates leaving a message.
  selectionType() returns the selection type, where 0=rectangle, 1=oval, 2=polygon, 3=freehand, 4=traced, 5=straight line, 6=segmented line,

7=freehand line, 8=angle, 9=composite, and 10=point. Returns -1 if there is no selection.

- *Line 4:* Empty array `tempProfile` is loaded with the intensity profile along the line ROI by `getProfile()`. *getProfile()* Runs `[Analyze > Plot Profile]` (without displaying the plot) and returns the intensity values as an array.
- *Line 5:* Passes the array `tempProfile` to function "output_results," which prints the content of array in the table shown in the "Results" window.
- *Lines 7–14:* A function for outputting the profile array in the table shown in the "Results" window. It takes an argument `rA`, which is supposed to be an array.
- *Line 8:* Clears the results table.
- *Lines 9–12:* For-loop to go through the array and to print out each element.
- *Line 10:* Sets the pixel position along the segment in the column labeled "n."
- *Line 11:* Sets the content of the array (pixel intensity) in the column labeled "intensity." *setResult("Column," row, value)* adds an entry to the ImageJ results table or modifies an existing entry. The first argument specifies a column in the table. If the specified column does not exist, it is added. The second argument specifies the row, where 0<=row<=nResults. (nResults is a predefined variable.) A row is added to the table if row=nResults. The third argument is the value to be added or modified.
- *Line 13: updateResults()* updates the "Results" window after the results table has been modified by calls to the setResult() function.

Exercise 3.13

Modify code 20.5 that the macro calculates the sum of all intensities.

Hint:

- You do not need the function anymore.
- `for`-loop should be used.
- Use `tempProfile.length`.

Answer:

The key for getting the sum of values in an array is for-loop to go through all elements of the array. The total sum of array values is calculated by adding up values during this for-loop.

```
macro "get profile and printout" {
  if (selectionType() !=5) exit("selection type must
      be a straight line ROI");
  tempProfile=getProfile();
  sum = 0;
  for (i = 0; i < tempProfile.length; i++) {
    sum += tempProfile[i];
```

```
    }
    print("sum of values:", sum);
}
```

Another way of achieving the similar task is by using array-related function. We will see this later.

3.6.4 Array Functions

Arrays can be directly treated using array functions. Since array is a very usable form of holding numbers and strings, it is good for you to know what they can do. Here is the list:

Array.concat(array1,array2) Returns a new array created by joining two or more arrays or values.

Array.copy(array) Returns a copy of array.

Array.fill(array, value) Assigns the specified numeric value to each element of array.

Array.findMaxima(array, tolerance) Returns an array holding the peak positions (sorted with descending strength). Tolerance is the minimum amplitude difference needed to separate two peaks. There is an optional "excludeOnEdges" argument that defaults to "true." Requires 1.48c.

Array.findMinima(array, tolerance) Returns an array holding the minima positions. Requires 1.48c.

Array.fourier(array, windowType) Calculates and returns the Fourier amplitudes of array. WindowType can be "none," "Hamming," "Hann," or "flat-top," or may be omitted (meaning "none"). See the TestArrayFourier macro for an example and more documentation. Requires 1.49i.

Array.getStatistics(array, min, max, mean, stdDev) Returns the min, max, mean, and stdDev of array, which must contain all numbers.

Array.print(array) Prints the array on a single line.

Array.rankPositions(array) Returns, as an array, the rank positions of array, which must contain all numbers or all strings.

Array.resample(array,len) Returns an array that is linearly resampled to a different length. Requires 1.47j.

Array.reverse(array) Reverses (inverts) the order of the elements in the array.

Array.show(array) Displays the contents of array in a window. Requires 1.48d.

Array.show("title," array1, array2, . . .) Displays one or more arrays in the Results window (examples). If the title (optional) is "Results," the window will be the active Results window; otherwise, it will be a dormant Results window (see also IJ.renameResults). If the title ends with "(indexes)," a 0-based Index column is shown. If the title ends with "(row numbers)," the row number column is shown. Requires 1.48d.

Array.slice(array,start,end) Extracts a part of an array and returns it.

Array.sort(array) Sorts array, which must contain all numbers or all strings. String sorts are case-insensitive in v1.44i or later.

Array.trim(array, n) Returns an array that contains the first n elements of array.

For example, array can be sorted and reversed. Try the following code.

```
EMBL = newArray("Heidelberg","Hamburg","Hixton",
   "Grenoble","Monterotondo");
Array.print(EMBL);
Array.sort(EMBL);
Array.print(EMBL);
Array.reverse(EMBL);
Array.print(EMBL);
```

The output of this code is as follows:

```
1 Heidelberg, Hamburg, Hixton, Grenoble, Monterotondo
2 Grenoble, Hamburg, Heidelberg, Hixton, Monterotondo
3 Monterotondo, Hixton, Heidelberg, Hamburg, Grenoble
```

The first line is printed in the order when the array was initialized. After sorting, names are in alphabetical order. Third line shows the reversed elements.

Some functions return an array rather than taking array/s as argument. See Section 3.7.1 for a list of these functions.

3.6.5 Application of Array in Image Analysis

3.6.5.1 Intensity Profile and Array Functions

To learn the actual use of array in image analysis, we explore several example applications. In the first application, we use `getProfile` function. We have already used `getProfile()` in Section 3.5.3. Here we use it in combination with Array functions to get local minima along the intensity profile – just like finding downward peak positions. We use a sample image Tree Rings.jpg (`[File > Open Samples > Tree Rings]`).

We draw a straight line ROI crossing tree rings, and then the macro detects ring positions along that line ROI and indicate those positions by point ROIs. The macro first reads the line profile from the straight line ROI and then we use `Array.findMinima` function to detect local minima (dark rings). Since this function returns the position of minima only as indices of the line profile array, we need to get `x` and `y` coordinates of minima from their indices in order to plot minima positions in the original image. For this purpose, we resample the straight line ROI to the same number of points as the length of line profile array. Let's write the code and learn by doing.

Note: Before running the macro code20_4.ijm, ensure to have a straight line ROI placed crossing tree rings (Figure 3.19).

```
1  //code 20.4
2  macro "select minima positions" {
3    if (selectionType() !=5)
4      exit("selection type must be segmented line ROI");
5    pA = getProfile();
6    minsA = Array.findMinima(pA, 40);
7    getSelectionCoordinates(xpoints, ypoints);
8    resamplex = Array.resample(xpoints, pA.length);
9    resampley = Array.resample(ypoints, pA.length);
10   minxA = newArray(minsA.length);
11   minyA = newArray(minsA.length);
12   for (i = 0; i < minsA.length; i++){
13     minxA[i] = resamplex[minsA[i]];
14     minyA[i] = resampley[minsA[i]];
15   }
16   makeSelection("point",minxA,minyA);
17 }
```

code/code20_4.ijm

- *Lines 3 and 4:* Check if the selection type is a straight line ROI using function `selectionType`. If not, macro terminates leaving a message.
- *Line 5:* An intensity profile array `pA` is sampled by `getProfile()`.
- *Line 6:* Detect local minima using `Array.findMinima`. The first argument is the line profile array, and the second argument is "tolerance." A larger tolerance value is less sensitive to intensity minimum – less detection. You can try changing this value later to see the effect. An array containing indices of minima positions is returned.

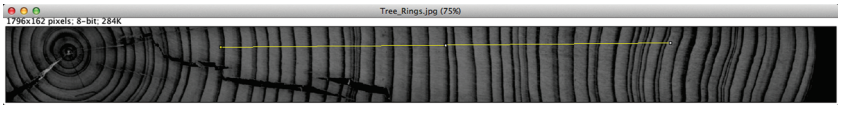


Figure 3.19 A straight line ROI crossing rings.

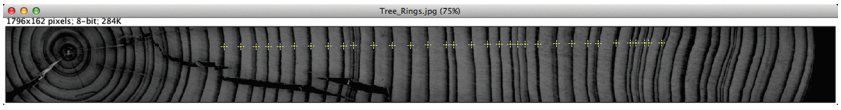


Figure 3.20 Detected ring positions.

- *Line 7:* `getSelectionCoordinates` with straight line ROI stores two arrays, each for start/end x coordinates and start/end y coordinates. Two arrays, in this case `xpoints` and `ypoints`, have length of 2.
- *Lines 8 and 9:* Resampling of straight line ROI by number of points in the line profile array `pA`.
- *Lines 10 and 11:* Prepare two new arrays to store x and y coordinates of minima positions.
- *Line 12:* For-loop to go through minima indices array.
- *Lines 13 and 14:* `minsA[i]` is the index for a single minimum, and using this index, x and y coordinates of that minimum position are retrieved and stored into new arrays prepared in lines 10 and 11.
- *Line 16:* After the looping, x and y coordinates of minima are used in `makeSelection` function to create multiple point ROI.

Run the code, and you will see multiple point ROIs indicating positions of rings (see Figure 3.20). Similar macro can be used to measure striated patterns in tissues or cell edges. In case of fluorescence images, `Array.findMaxima` can be used to detect high-intensity maxima positions.

3.6.5.2 Extending Stack Analysis by Direct Measurements

We have studied how to use for-loops to measure each frame/slice within a stack (Section 3.5.4.1). We did measurements by first setting measurement parameters with `run("Set Measurements...")` and then did measurement by `run("Measure")`. Measured values were shown in the table in the "Results" window. To use these measured values, for example, to calculate statistics or plot the results, one should access the table in the "Results" window and parse all the values. This is possible with the macro language, but we will try a more direct method by directly accessing the measured values. There are two ways to access measured values without using the Results table.

1) `getRawStatistics(nPixels, mean, min, max, std, histogram)`
2) `List.setMeasurement.`

The function `getRawStatistics` measures statistical parameters from the image and returns those values in the variables declared as arguments. In other words, after this function is executed, variable `mean` will have the mean intensity of the image.[11] If a ROI is selected, mean intensity of that ROI will be the value of `mean`. We can loop each slice/frame within a stack and for each loop we can use `getRawStatistics` and store measured values in arrays. But there is a

11) In this example we use a variable named `mean`, but the name can be anything such as `a` or `b`.

drawback of using this function: The available parameters to measure are limited.

The second method `List.setMeasurement` does not have this limitation. One can measure many more parameters because all the available parameters listed in `[Analyze > Set Measurements . . . ]` are accessible with this function. The basic usage is shown in the code below that measures the currently active image, extracts specific measurement value (in this example case, "Mean" intensity), and then prints out that value in the Log window. Try writing this code and test it with any image.

```
1 List.setMeasurements;
2 mean = List.getValue("Mean");
3 print(mean)
```

We can do the measurement using `List.setMeasurement` function for every loop for stack slices/frames and store the results in arrays. Here is the code, a modified version of code 10 (p 37):

```
1  //Code 10.1
2  requires("1.42i");
3  macro "Measure Ave Intensity Stack" {
4    frames=nSlices;
5    meanA = newArray(frames);
6    sdA = newArray(frames);
7    for(i=0; i<frames; i++) {
8      currentslice=i+1;
9      setSlice(currentslice);
10     List.setMeasurements;
11     meanA[i] = List.getValue("Mean");
12     sdA[i] = List.getValue("StdDev");
13
14   }
15   Array.print(meanA);
16   Array.print(sdA);
17 }
```

code/code10_1.ijm

- *Line 2:* Checks the ImageJ version, since `List.setMeasurements` function is available only after version 1.42i.
- *Lines 5 and 6:* Create new arrays with their length equal to the number of frames of the stack. These arrays will be used to store measurement results.
- *Line7:* For-loop going through each frames in the stack.
- *Line 10:* Measures all the parameters, which will be stored in the List.
- *Lines 11 and 12:* Retrieve the results, mean intensity, and its standard deviation.
- *Lines 15 and 16:* Print out results in the Log window.

3.6.6 Working with Strings

With some advanced macro programming, you might need to manipulate strings (texts) from your code. For example, let's think about a title of an image "exp13_C0_Z10_T3.tif." Such naming often occurs to indicate that this image is from the third time point (T3), at the 11th slice (Z10, imagine that the Z slice numbering starts from 0), and it is the first channel (C0).

We might be fortunate enough to read out its dimensional information from the header, but quite often such information is available only in the file name (the title of the image). To extract dimensional information from the file name, we need to know how to deal with strings in macro to decompose those strings and extract information that we need. The following are the built-in macro functions that are related to such tasks with strings.

- lengthOf(str)
- substring(string, index1, index2)
- indexOf(string, substring)
- indexOf(string, substring, fromIndex)
- lastIndexOf(string, substring)
- startsWith(string, prefix)
- endsWith(string, suffix)
- matches(string, regex)
- replace(string, old, new)

Let's go back to the example file name "exp13_C0_Z10_T3.tif" again. If we need to get the file name without file extension, what should we do? Several ways are there, but let us start with the simplest way.

We already know that all the file names are in the TIF format, so all file names end with ".tif". We can remove this suffix by replacing the ".tif" with a string with length 0. We can do this by using `replace`:

```
1 name = "exp13_C0_Z10_T3.tif";
2 newname = replace(name, ".tif", "");
3 print(newname);
```

This will print out "exp13_C0_Z10_T3" in the Log window. In the second line, the function `replace` is used. The old string ".tif" is replaced by a new 0 length string "". So it works!

But what if our lucky assumption that all files end with ".tif" is not true and it could be anything? To work on this, we now need to use a different strategy to know the file extension.

By definition, file extension and the file name are separated by a dot. The length of the extension can be different, as some extension such as a Python file is ".py" and a C code is ".c." Thus, we cannot assume that the length of the file extension is constant, but we know that there is a dot.

For such cases with variable length of file extension being expected, we first need to know about the *index* of the dot within the file name. Each character within the file name is positioned at a certain index from the beginning of the name. In the example we are now dealing with, the index 0 is "e." The index 1 is "x." Since the index starts from 0, the last index will be the total length of the file name minus 1. You can modify the above code as given below to get the length of the file name:

```
1 name = "exp13_C0_Z10_T3.tif";
2 tlength = lengthOf(name);
3 print(tlength);
```

You should see "19" in the Log window. That is the length of this file name. So in this example string, index starts from 0 and the last index is 18.

Next, we use the function `substring(string, index1, index2)`. With this function, you can extract part of the string by giving the start index (index1) and the end index (index2) as arguments. We can try this by again modifying the above code:

```
1 name = "exp13_C0_Z10_T3.tif";
2 subname = substring(name, 0, 3);
3 print(subname);
```

The output after running this code is "exp" printed in the Log window. The second argument of the function `substring` is 0, and the third is 3. This tells the function `substring` to extract characters from the index 0 to the index 2 (so the third argument will be the index just after the last index that would be included in the substring).

Exercise 3.14

Test changing the second and the third argument so that different parts of the file name are extracted.

How could we know the index of the dot? For this, we use `indexOf(string, substring)`. Try the following code:

```
1 name = "exp13_C0_Z10_T3.tif";
2 dotindex = indexOf(name, ".");
3 print(dotindex);
```

Now you know that the index of dot is "15." We can then combine the knowledge we have now to compose a single macro that extracts the file name without file extension.

```
name = "exp13_C0_Z10_T3.tif";
dotindex = indexOf(name, ".");
filename = substring(name, 0, dotindex);
print(filename);
```

Let's make the problem a bit more complicated. If the file name contains multiple dots, what should we do? In the example below, I have added two more dots:

```
name = "exp13._C0._Z10_T3.tif";
dotindex = indexOf(name, ".");
filename = substring(name, 0, dotindex);
print(filename);
```

Output is now "exp13" – far from what we need. To treat such case, we use `lastIndexOf`, which returns the index of the last appearance of the given character. Let's slightly modify the code:

```
name = "exp13._C0._Z10_T3.tif";
dotindex = lastIndexOf(name, ".");
filename = substring(name, 0, dotindex);
print(filename);
```

It should then work again as we want.

Let's change our task: We now want to know the time point when this image was taken. How should we do that? Examining the file name again, we realize that the time point number appears after "T". The number can be any length of digits, but currently it is 0. Then the dot comes right after the number. We then just need to know the index of "T", but wait, we might have "T" anywhere, as this is a single character alphabet that could easily be a file name. Therefore, we will find the index of "_T" that looks more specific:

```
name = "exp13._C0._Z10_T3.tif";
timeindex = indexOf(name, "_T");
print(timeindex);
```

Now we know that "_T" is at index 14, so the number should start from the index 16 (because index 15 will be "T"). Taking this into account, we can extract the time point:

```
name = "exp13._C0._Z10_T3.tif";
timeindex = indexOf(name, "_T");
dotindex = lastIndexOf(name, ".");
timepoint = substring(name, timeindex + 2, dotindex);
print(timepoint);
```

The time point that you have just now captured is a string. You cannot pass this to mathematical assignments. To do so, you need to convert this to a number. For doing so, you can use `parseInt(string)`:

```
name = "exp13._C0._Z10_T3.tif";
timeindex = indexOf(name, "_T");
dotindex = lastIndexOf(name, ".");
timepoint = substring(name, timeindex + 2, dotindex);
timepoint = parseInt(timepoint);
print(timepoint * 2);
```

An example case where conversion of string to a number (in this case an integer) is required would be when you need to compare such file names and get the maximum time points from all the file names. Usage is diverse, but at some point you need to use this. If you need a Float number (numbers with decimal point), use `parseFloat(string)`.

3.7 Appendix

3.7.1 Built-In Macro Functions Using Array

Many built-in macro functions return an array to have multiple numerical values as a singular object. Below is a list of these functions:

```
Dialog.addChoice("Label", items)
Dialog.addChoice("Label", items, default)
Fit.doFit(equation, xpoints, ypoints)
Fit.doFit(equation, xpoints, ypoints, initialGuesses)
getFileList(directory)
getHistogram(values, counts, nBins[, histMin, histMax])
getList("window.titles")
getList("java.properties")
getLut(reds, greens, blues)
getProfile()
getRawStatistics(nPixels, mean, min, max, std, histogram)
getSelectionCoordinates(xCoordinates, yCoordinates)
getStatistics(area, mean, min, max, std, histogram)
makeSelection(type, xcoord, ycoord)
newArray(size)
newMenu(macroName, stringArray)
Plot.create("Title", "X-axis Label", "Y-axis Label",
xValues, yValues)
Plot.add("circles", xValues, yValues)
```

```
Plot.getValues(xpoints, ypoints)
setLut(reds, greens, blues)
split(string, delimiters)
```

Acknowledgment

The author is grateful to Chong Zhang for intensively commenting on this chapter.

4
Introduction to Matlab

Cornelia Monzel[1] *and Christoph Möhl*[2]

[1]*Institut Curie, Physico-Chimie Curie, Light-Based Observation and Control of Cellular Organization, 26 rue d'Ulm, 75248 Paris Cedex 05, France*
[2]*German Center of Neurodegenerative Diseases (DZNE), Image and Data Analysis Facility (IDAF), Core Facilities, Holbeinstraße 13–15, 53175 Bonn, Germany*

4.1
Aim

Within this module you will get familiar with basic concepts of the Matlab programming environment. By the end of this chapter, you will be able to analyze images within Matlab and to visualize the results with statistical plots. With matrix manipulation and advanced indexing, you will learn basic programming concepts to manipulate image data on the pixel level. These concepts are not restricted to Matlab but give you a solid foundation to exploit similar programming languages as, for example, R and Python (especially the numpy package).

4.2
Tools

4.2.1
Matlab (Incl. Image Processing Toolbox)

Matlab is a commercial software for numerical computing. We use Matlab to analyze images, to calculate features of the detected image objects, and for statistical analysis and plotting of results.

Bioimage Data Analysis, First Edition. Edited by Kota Miura.

4.3 Getting Started with Matlab

4.3.1 The Matlab User Interface

Matlab can be considered as a very powerful calculator. When you start Matlab for the first time, two important windows show up:

- *The command window:* Here you can type your calculations and the result is displayed immediately.
- *The workspace:* Here the results of your calculations are stored in variables.

4.3.2 Matlab as a Calculator

Let's start by doing a simple calculation just by typing to the command window and pressing enter:

```
>> 2+3
ans =
     5
```

The result (5) is plotted to the command window and saved into the workspace as a variable called `ans`. If you don't specify the variable name of your result, it is set to `ans` by default. In most cases, it makes more sense to define a variable name. You can use the variables stored in the workspace for subsequent calculation steps:

```
>> a=2+3
a =
     5
>> b=1
b =
     1
>> c=a+b
c =
     6
```

You can, of course, also use other types of arithmetic operators and use brackets for more complicated expressions:

```
>> d=(a+b)*4/(14-2)
d =
     2
```

Check the Workspace panel. There should be five variables ('a'-'d' and 'ans') listed, with their names and their values. Often, you need to perform a series of calculations

multiple times. Instead of retyping them into the command line each time, you can write them into a text file, called "script" (comparable to an ImageJ Macro). To create a new script, go to the HOME tab and click the icon "New Script." A new editor window opens, where we can write down our sequence of calculations:

```
1 a=2+3
2 b=1
3 d=(a+b)*4/(14-2)
```

verb/code1.m

You can execute the commands in the script by using the key `F5` or pressing the run-button (the small green triangle).

A dialog opens to save the script. If the script is not located in the so-called current folder, change the current folder to the folder where the script is located. Do not choose the option "add path" for now.

Generally, your main script should be located in the current folder. This is your working directory. If you want to load or save data (see Section 4.3.18), it is always placed in this directory by default (unless a complete path is specified). If you call functions within your script (see Section 4.3.7), they are first searched for in the current folder and afterwards in all directories that are specified as function paths (you can change the set of function paths with the command `pathtool`)

As before, the results of each calculation executed with the run-button are printed into the command window:

```
a =
     5
b =
     1
d =
     2
```

Use a semicolon behind each command to suppress its command line output:

```
1 a=2+3; %this is the definition of variable a
2 b=1; %this is the definition of variable b
3 d=(a+b)*4/(14-2); %this is a complicated calculation to define
      variable d
```

verb/code2.m

By suppressing the command line output you increase the execution speed. For the simple operations performed here, the effect is negligible, but as soon as you will work with large arrays of numbers, your code will be much faster if you suppress the command line output.

The quick access to command line and variable workspace facilitates an interactive way of programming, that makes data analysis tasks very effective. All variables are available after the execution of a script, and their values can be displayed by typing their name in the command line or clicking them in the workspace window. This makes debugging much easier compared to, for example, the ImageJ macro language, where variables values have to be displayed with the *print* command during script execution. Other packages like *R* or *iPython* (interactive Python) have similar interactive programming features and are a good freeware alternative to Matlab.

In the example code above, you can also see how to add comments to your code. Comments in MATLAB begin with the percent sign (remember: in ImageJ Macro Language you use double backslash for comments).

4.3.3 Vectors

Before we proceed, we remove all variables from the workspace by typing `clear` at the command line. Check the Workspace panel, all the variables you created are now gone. To erase all objects from memory including variables and functions, you can type `clear all`.

Until now, we've only stored a single number in our variables. One variable can contain multiple values, and in this case it is called a vector. In the following example, the variable a is defined as a vector of four values:

```
a=[3 17 45.3 9];
```

To get a single element from a vector, use round brackets and the element's index:

```
a =
    3.0000   17.0000   45.3000    9.0000
>> b=a(1)
b =
     3
>> c=a(2)
c =
    17
>> d=a(end)
c =
    9
```

Note that the indices don't start at 0 as in most programming languages (e.g., R), but at 1. To address the last element, use the word `end` as index.

Exercise 4.1 Simple Calculations with Vector Elements

Vector `a` is given:

```
a= [3 17 45.3 9]
```

Perform the following calculation within one line of code with as little brackets as possible: divide the first element by the third and multiply the result with the sum of the second and fourth element.

4.3.4 Multiple Indexing

With a vector of indices, you can even get multiple elements at once:

```
>> indizes=[1 4]
indizes =
     1     4
>> d=a(indizes)
d =
     3     9
>> d=a([1 4]) %short version
d =
     3     9
```

4.3.5 Creating and Deleting Vector Elements

By using the colon operator, you can create vectors with ascending value with default spacing 1

```
[minValue:maxValue]
```

or user-defined spacing:

```
[minValue:spacing:maxValue]
```

```
>> a=[1:7]
a =
     1     2     3     4     5     6     7
>> b=[1:2:7]
b =
     1     3     5     7
>> c=[10:10:100]
c =
    10    20    30    40    50    60    70    80    90   100
```

You can also combine two vectors to a new one:

```
>> a= [1 2 3];
>> b= [4 5 6 7];
>> c=[a b]
c =
     1     2     3     4     5     6     7
```

Or delete specific elements with `[]`:

```
>> c(1)= []
c =
     2     3     4     5     6     7
>> c(end-2:end)
ans =
     5     6     7
>> c(end-2:end)=[]
c =
     2     3     4
```

Exercise 4.2

Two vectors a and b are given:

```
a = [1 2 3 4]; b = [5 6 7 8]
```

Create a vector `c = [2 4 6 8]` by taking elements from vectors a and b. Use multiple indexing!

4.3.6 Calculations on Vectors

In Matlab, it is very easy to perform element-by-element calculations on a whole vector at once, without building a loop (similar in *R* and the *numpy* package of *Python*). In the following example, the value 2 is added to each element of vector `a`:

```
a =
    3.0000   17.0000   45.3000    9.0000
>> b=a+2
b =
    5.0000   19.0000   47.3000   11.0000
```

In the next example, vector *b* is subtracted element by element from vector *a*:

```
b =
    1     1     1     2
>> c=a-b
c =
    2.0000   16.0000   44.3000    7.0000
```

If you do element-by-element operations using multiplication or division, you have to type a dot before the operator:

```
>> c=a.*b
c =
    3.0000   17.0000   45.3000   18.0000
>> d=a./b
d =
    3.0000   17.0000   45.3000    4.5000
```

Another important operator is "power" (^). If you want to apply (^) element-wise to vectors, you also have to use the dot. In the following example, we square each element of the vector ***a*** by using (`.^2`):

```
>> a
a =
     2     3     4
>> b = a.^2
b =
     4     9    16
```

4.3.7 Functions

The above calculations only work when both vectors `a` and `b` have the same number of elements. To develop a robust code it is, therefore, often needed to check the number of elements before performing such calculations. The number of elements of a vector can be determined with the `numel` function:

```
a =
     1     5     3
b =
     1     5     3     4
>> c=numel(a)
c =
     3
>> d=numel(b)
d =
     4
```

To find out more about the function `numel`, just type `doc numel` at the command line to open Matlab help.

There are countless functions available in Matlab and on the website (e.g., the Matlab file exchange platform http://www.mathworks.de/matlabcentral/fileexchange/). You can also program your own functions.

All functions have in common, that you call them by their name (e.g., `numel`) with parameters separated by commas in round brackets (if required). Here are some examples of basic functions that are used very often:

- `clear()` or `clear`: remove all variables from the workspace.
- `b=sum(a)`: returns the sum of all elements of vector ***a***.
- `b=abs(a)`: all negative elements of a become positive.
- `b=max(a)`: returns the element with the largest value.
- `b=min(a)`: returns the element with the smallest value.
- `c=max(a,b)`: returns a vector of same length as ***a*** and ***b***, and selects the largest value element-wise.

By nesting functions, you can make multiple calculations in one line of the code:

```
c=sum(abs(a))*numel(b)
```

4.3.8 Plotting Data

Visualizing numbers in a graph is done with the `plot` function. Before you plot, you have to create a new figure window by using the `figure` function.

`plot` needs at least the x and y position of your data and, optionally, some color and line style arguments as strings.

```
1  a=1; b=2;
2
3  close all %all figures are closed
4  figure(1); %opens a new figure window
5
6  plot(a,b,'+r') %plots a red cross at position (a,b)
7  hold on % retains current graph in figure
8  plot(b,a,'.g') %plots a green dot at position (b,a)
9
10 axis([0 3 0 3])%defines the extends of x and y axes
```

verb/plotcode1.m

The output is shown in Figure 4.1 below.

Figure windows can be closed with the `close` function. Often it is very convenient to integrate the command `close all` into your script before creating new figures. As shown in the code example, the axes are defined by the following command:

```
axis([xmin xmax ymin ymax])
```

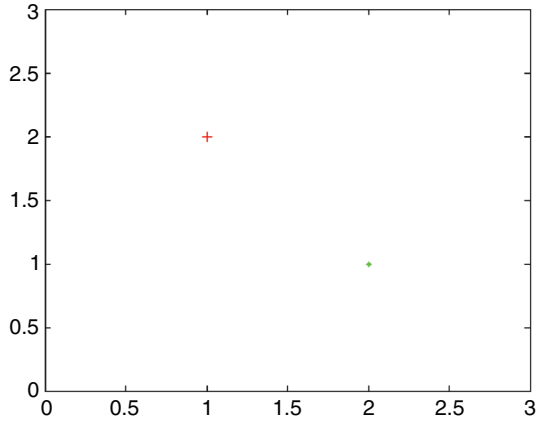


Figure 4.1 Our first plot.

You can plot vectors as easily as single elements:

```
1  x=[1 2 3 4]
2  y=[1 1 2 2]
3
4  close all
5  figure(1)
6
7  plot(x,y) %plot a blue line
8  hold on
9  plot(x,y,'.r') %plot red dots
10
11 axis([0 5 0 3])
```

verb/plotcode2.m

The result of this code is shown in Figure 4.2.

Exercise 4.3 Plot a Sine Wave

Use the function `sin` for calculation (go to Matlab help for more details) and `plot` for visualization.

4.3.9 Matrices

A vector is an array with one dimension. In Matlab, you can also work with arrays of two or more dimensions. From now on, we call two-dimensional arrays matrices and arrays with three or more dimensions multidimensional matrices. In matrices, elements are organized in rows and columns. Filling a matrix with

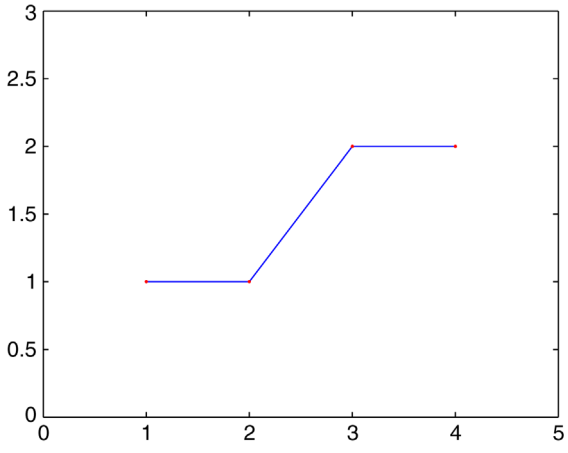


Figure 4.2 Our second plot.

values is very much like filling a 1D vector. The only difference is that the rows are separated by semicolons:

```
>> a=[1 2 3; 4 5 6; 7 8 9]
% or:  a=[1:3; 4:6; 7:9]
a =
     1     2     3
     4     5     6
     7     8     9
```

To get a single element from a matrix, you need indices for row (1st index) and column (2nd index):

```
>> a(3,3)
ans =
     9
```

Here are two examples to get more than one element at once by using the colon (:):

```
a =
     1     2     3     4
     5     6     7     8
     9    10    11    12
>> b=a(1,1:3)
b =
     1     2     3
>> c=a(2,:)
c =
     5     6     7     8
```

`b` contains the first to third element of the first row. `c` contains all elements (:) of the second row.

The number of rows and columns can be determined by the `size` function: `[m,n] = size(X)`, where `m` is the number of rows and `n` is the number of columns. The `numel` function (that we already used with vectors) gives the total number of elements. Another frequently used function for matrix or vector dimension is `length` that gives the number of elements of the largest dimension:

```
a =
     1     2     3     4     5
     6     7     8     9    10
>> n = numel(a) %total number of elements
n =
    10
>> [nRows nColumns] = size(a) %size of all dimensions
nRows =
     2
nColumns =
     5
>> nLongest = length(a) %size of longest dimension
nLongest =
     5
```

Rows and columns can be swapped[1] by using the (') operator:

```
a =
     1     2     3     4
     5     6     7     8
>> b=a.'
b =
     1     5
     2     6
     3     7
     4     8
```

The functions `zeros(nrows,ncolumns)` and `ones(nrows,ncolumns)` are often used to initialize a matrix:

```
>> a=zeros(2,4)
a =
     0     0     0     0
     0     0     0     0
>> b=ones(2,2)
b =
     1     1
     1     1
```

1) Matrix transposition.

The matrices can be filled afterwards by arithmetic operations or specific assignment of elements:

```
>> a=a+2
a =
     2     2     2     2
     2     2     2     2
>> b(:,1)=34
b =
    34     1
    34     1
```

Exercise 4.4 Create the Following Matrix

```
     1     2     3     3     3
     1     2     3     3     3
     1     2     3     3     3
     1     2     4     4     4
     1     2     4     4     4
```

Keep the code as short as possible!

4.3.10 Logical Operations

Logical operations are done with so-called logical operators:

- > (larger than)
- < (smaller than)
- >= (larger than or equal to)
- <= (smaller than or equal to)
- == (is equal to)

```
>> a = 3
a =
     3
>> b = a>1
b =
     1
>> c = a>5
c =
     0
>> d = a==3
d =
     1
```

The outputs *b*, *c*, and *d* have either the value 0 or 1, whereas 1 means TRUE and 0 means FALSE.

If you have a look to those variables in the workspace window, you recognize that they appear with a different icon than regular variables *a*. Variable *a* is a floating number (double) that can be used for arithmetic operations. *b*, *c*, and *d* have logical values that can only be either 0 (FALSE) or 1 (TRUE). They are called Booleans.

The ~ (NOT) operator turns a Boolean value into its opposite, that is, each FALSE becomes TRUE and vice versa:

```
a =
     1
>> b=~a
b =
     0
```

Besides doubles and Booleans, there are also other data types in Matlab, like integers or strings (text), similarly to the ImageJ macro language (see Section 4.3.16).

Logical operations also work with vectors or matrices:

```
a =
     1     2     3     4     5     6
>> a>3
ans =
     0     0     0     1     1     1
```

4.3.11 Conditional Statements If and Else

With the use of Booleans and the if/else statements, you can execute a code section only under certain conditions:

```
1  a=5;
2  b=a>3;
3
4  if b    % if b has value 1 (TRUE)
5      disp('a is larger than 3')
6
7  else    %if b has any other value
8      disp('a is not larger than 3')
9
10 end
```

verb/ifcode1.m

Command line output after execution:

```
a is larger than 3
>>
```

The keywords `if` and `end` enclose the conditional statement. The `disp` function displays text in the command line.

4.3.12 Indexing with Boolean Masks

In Matlab, Boolean vectors can also be used for indexing. Boolean vectors are a very nice way of filtering elements from a vector by logical expressions:

```
a =
     2     2     2     4     4     4
>> b = a>3 %the boolean vector
b =
     0     0     0     1     1     1
>> c=a(b)
c =
     4     4     4
```

Or in a short version:

```
>> c=a(a>3)
c =
     4     4     4
```

The `find` function computes the index vector of a Boolean mask, that is, it gives all indices, with value 1 (TRUE):

```
b =
     0     0     0     1     1     1
>> indizes=find(b)
indizes =
     4     5     6
```

Since Boolean masks and index vectors can both be used for calling elements of a vector, the following commands give identical results:

```
>> c = a(a>3) %call by boolean mask
c =
   17.0000   45.3000    9.0000
>> c = a(find(a>3)) %call by index
c =
   17.0000   45.3000    9.0000
```

Exercise 4.5 Plot a Sine Wave in Different Colors

Plot a sine wave with positive values appearing in blue and negative values appearing in red.

4.3.13 Linear Indexing vs. Subscript Indexing

Instead of using one index for each dimension (subscripts), it is also possible to just use a so-called linear index.

```
a =
     1     2     3
     4     5     6
     7     8     9
>> a(3,3) %subscript indexing
ans =
     9
>> a(9) %linear indexing
ans =
     9
>> a(2) %linear indexing: rows come first!
ans =
     4
```

As you see above, columns come first in the indexing order. The linear index "sees" our matrix like this:

```
[1 4 7 2 5 8 3 6 9]
```

By using the functions `sub2ind` or `ind2sub`, you can convert between linear indices and subscripts. As input argument, you need not only the index vector but also the size of the matrix:

```
a =
     1     4     7
     2     5     8
     3     6     9
>> linearInd=find(a>6) % linear index vector by logical operation
linearInd =
     7
     8
     9
>> [i j]=ind2sub(size(a),linearInd) %
2 index vectors for rows and columns
i =
     1
     2
     3
```

```
j =
     3
     3
     3

>> a(i(1),j(1)) % get element with subscripts
ans =
     7
>> a(linearInd(1)) %get element with linar index
ans =
     7
```

4.3.14 Ordering of Rows and Columns

For addressing elements by subscript indices, columns come always first:

```
element = matrix(row,column)
```

This is a general rule in Matlab and most functions follow this concept. Let's illustrate this with the `sum` function. `sum` computes the sum of a vector:

```
a =
     1     1     1     1
>> b=sum(a)
b =
     4
```

However, if applied to a matrix, the sums are calculated for each column. Thus, the result is a vector (with one row and multiple columns):

```
a =
     1     2     3
     1     2     3
     1     2     3
>> b=sum(a)
b =
     3     6     9
```

By default, `sum` is applied to the first dimension. If you want to sum over the second dimension, you have to use a second, optional, argument for specifying the dimension:

```
>> c=sum(a,2)
c =
     6
     6
     6
```

If you like to sum all elements, you can apply the function twice, or first convert the matrix to one dimension by using the colon operator:

```
>> d=sum(sum(a)) %apply sum twice
d =
    18
>> a1d=a(:) %convert to one dimension
a1d =
     1
     1
     1
     2
     2
     2
     3
     3
     3
>> sum(a1d)
ans =
    18

>> sum(a(:)) %short version
ans =
    18
```

4.3.15 The for Loop

As you have learned, many operations in Matlab can be applied on multiple elements at once. Thus, constructing iterative sequences with loops is often not necessary. However, it is still possible. The most common one is the `for` loop:

```
1 for i=1:4
2     i
3 end
```

verb/code_for_loop.m

The script above produces the following command line output:

```
i =
     1
i =
     2
i =
     3
i =
     4
```

You see that *i* is automatically incremented from 1 to 4. There is no need for an expression like `i=i+1`. The loop is closed by the keyword `end` (similar to the `if` statement).

4.3.16 Data Types

So far we worked with two different data types:

- *double:* floating point numbers, for example, 3.233, −5.0, 34000.423
- *Boolean:* either 0 (FALSE) or 1 (TRUE)

The following data types are also frequently used in Matlab:

- *Integer:* numbers without decimal components, for example, 3, −5, 34 000
- *String:* text, for example, "`heinz`," "`image_01.tif`," "`cell area`'

4.3.17 Operations with Strings

In our course, we use strings mainly to handle file names or graph annotations. Strings are defined between apostrophes:

```
a = 'heinz'
```

Strings are vectors of characters. The above vector `a`, for example, has five elements:

```
>> a='heinz';
>> numel(a) %give number of elements
ans =
     5
>> a(1) %give first element
ans =
h
>> a(2)  %give second element
ans =
e
```

With `strcat` you can concatenate strings:

```
>> doc strcat
>> a='hei';
>> b='nz';
>> c=strcat(a,b)
c =
Heinz
```

The function num2str() converts doubles or integers into strings. This is useful if you want to build a filename from a counter variable, for example:

```
>> oldFilename='image_02.tif'; % this is a string
>> imageNumber=3; %this is a number
>> newFilename=oldFilename;
>> newFilename(8)=num2str(imageNumber) %replace 8th element
newFilename =
image_03.tif
```

By default, doubles are converted with a precision of up to 4 digits. Numbers with higher precision are rounded to four digits before string conversion:

```
>> num2str(2.45379) % 5 digits
ans =
2.4538
```

The precision can be defined with an optional second argument:

```
str = num2str(A, precision)
```

"Useless" decimals with value zero are ignored:

```
>> num2str(2.300)
ans =
2.3
```

4.3.18 Import and Export Variables

With the function save, your workspace or user-defined variables from the workspace can be exported to a mat file. By default, this file is saved to the current folder.

To import the variables from the matfile to the workspace, use the function load. load and save use string arguments to specify the name of the mat file:

```
save('filename')%save complete workspace
save('filename','var1','var2')%save two variables workspace
```

Remember: With clear all you can reset the workspace.

The load function has the same syntax as save:

```
load('filename')%load all variables from the file filename.mat
```

You can use the filename with or without extension (.mat).

4.3.19 The Structure Array (Struct)

Structure arrays are useful to organize data. They are containers for different categories of variables, called fields.

Example: we construct a structure variable called `student`. A student is defined by a set of features (fields). Those features can have different types:

```
1 student.name = 'heinz'; %string vector
2 student.EyeColor = 'blue'; %string vector
3 student.height = 188; % double
4 student.weight= 76; %double
5 student.grades= [3.4 1 2 4 3]; %vector of doubles
```

verb/code4.m

After executing the script, we print the struct in the command line by typing `student` (without semicolon):

```
>> student
student =
        name: 'heinz'
    EyeColor: 'blue'
      height: 188
      weight: 76
      grades: [3.4000 1 2 4 3]
```

Getting values out of the struct is as easy as defining values:

```
>> a=student.name
a =
heinz
>> b=student.grades
b =
    3.4000    1.0000    2.0000    4.0000    3.0000
```

We now create a vector of structs. Here, I just add a second element to `student`:

```
1 student(2).name = 'olaf'
2 student(2).EyeColor = 'grey'
3 student(2).height = 175
4 student(2).grades = [1.3 3 5 4 5]
```

verb/code5.m

Now, our struct has two elements:

```
>> student(1)
ans =
        name: 'heinz'
    EyeColor: 'blue'
      height: 188
      weight: 76
      grades: [3.4000 1 2 4 3]
>> student(2)
ans =
        name: 'olaf'
    EyeColor: 'grey'
      height: 175
      weight: []
      grades: [1.3000 3 5 4 5]
```

You can get specific values out of that structure, for example, the third grade of the second student, like this:

```
>> a = student(2).grades(3)
a =
     5
```

Exercise 4.6 Getting Image Metadata

The function `imfinfo` reads metadata of image files into Matlab. This is often useful to initialize the importation of an image stack. The output of `imfinfo` is organized as a structure.

Copy the sample images `nuclei.tif` and `film_7.tif` into a subfolder of your current folder with name `data`. `nuclei.tif` is a single image (only one frame), `film_07.tif` is a stack of images, a time-lapse sequence with 90 frames (you can check this by having a quick look at it with Fiji).

Apply `imfinfo` to the single image first, and then to the stack. The only argument must be a string with the subfolder name and filename (e.g., "`data/nuclei.tif`"). Observe the data structure of the output. What is the difference between the two outputs? Try to write some lines of code to get the number of frames, as well as width and height for both image files.

4.3.20 Cell Arrays

Cell array is a very flexible data structure. Like normal matrices, cells are arrays of elements of one, two, or more dimensions. However, in a normal matrix, all elements have the same data type.

In a cell array, one element can be a double, the other could be an integer, or a string, or a matrix of Booleans, or even another cell array (!). In a cell array, you can pack anything you want. This is very convenient on the one side, on the other side: It is very likely to mess around with cell arrays and lose the overview of your data structure.

Cell arrays are mainly handled as normal matrices. They are defined with the command `cell(nRows,nColumns)`. The main difference is to use curly brackets instead of round brackets:

```
>> a=cell(2,3)
a =
    []    []    []
    []    []    []
>> a{1,1}=13.7; % a double
>> a{1,2}='heinz'; % a string
>> a{1,3}=[1:5]; % a vector of doubles
>> a{2,1}=cell(2,2), % a cell array
a =
    [ 13.7000]    'heinz'    [1x5 double]
    {2x2 cell}         []              []
```

Cells are arrays of containers, and those can have any content. The containers are accessed with round brackets, the content of the containers is accessed with curly brackets. Here are some examples how to access contents of the cell created above:

```
>> a(1,3) %acces a container element
ans =
    [1x5 double]
>> a{1,3} %access the container content (here: a vector)
ans =
     1     2     3     4     5
>> a{1,3}(1) %access the 1st element of the vector
ans =
     1
```

You may notice that a cell array is similar to a structure. The main difference is, that the containers of a structure (called fields) have specific names, whereas containers of a cell are specified by their position.

With `struct2cell` you can convert structures to cell arrays:

```
>> student
student =
        name: 'heinz'
    EyeColor: 'blue'
      height: 188
      weight: 76
      grades: [3.4000 1 2 4 3]
```

```
>> studentCell=struct2cell(student)
studentCell =
    'heinz'
    'blue'
    [              188]
    [               76]
    [1x5 double]
```

Another useful function is `cell2mat` to concentrate a cell array of matrices that have identical dimensions and data format to one normal matrix:

```
  a=cell(3,1);
 a{1}=[1 1];
 a{2}=[2 2];
 a{3}=[3 3]
a =
    [1x2 double]
    [1x2 double]
    [1x2 double]
>> b=cell2mat(a)
b =
     1     1
     2     2
     3     3
```

4.4 Images in Matlab

Gray value images are nothing else than matrices of pixel intensity values, so you can basically apply all the calculations we learned so far to them.

In the following exercises, you measure the area of nuclei from a fluorescent nucleus stain image. This small project is divided into the following parts:

- We import an image showing a labeled nucleus.
- We detect nuclei inside the image.
- We measure the areas of the nuclei and plot the result.

First, start a new empty Matlab script

```
[File > New... > Script]
```

Save it and copy the image `nuclei.tif` to a subfolder called `data`.

Exercise 4.7 Image Import with Imread

To import an image file to a matrix, `imread` needs two input parameters:

- the path to the image (as string)
- For multidimensional images (e.g., multi-TIF), the index of the image to read. The image `nuclei.tif` is two dimensional so that the image index is not strictly required but we set it to "1."

Run the script `verb/code6.m` (see below) to import the image.

```
1 datname='data/nuclei.tif';
2 im=imread(datname,1); % im is a 2d matrix containing
     gray values of the tif file
3 im=double(im); % conversion
```

verb/code6.m

More input parameters can be used, we will see this next.

Exercise 4.8 Have a First Look

To visualize the image, we use the command `figure` first to create a figure window and then use the function `imshow` to display the image in that window. We will see later that we can display an image and overlay a plot in the same figure window.

```
1 figure(1) %new figure window
2 imshow(im);%show image
```

verb/code7.m

When you run this script, a window with an image appears but the image is just a white plane. Don't be surprised: The image is there, it is just that the brightness and the contrast of the image are not yet properly set (it appears saturated). One way to fix this is to divide the image by some value and visualize it again with `imshow`.

However, changing image data for visualization purpose is normally not recommended. Instead of manipulating gray values a second empty parameter [ ] can be passed to `imshow`. Then, the contrast is automatically stretched to the image intensity range:

```
imshow(im,[])
```

Change your script according to the instruction above! Alternatively you can call `imagesc(im)`. This function also stretches the contrast and applies a specific LUT (that can be changed).

Exercise 4.9 Plot the Gray Value Distribution

As explained in the section "Histogram" of ImageJ Basics an image intensity histogram is a useful representation of an image. To compute the histogram of a vector or matrix, we have to assign each element to a certain bin, based on its value. The binning can be done by the function `hist`. This function does not only generate the histogram data, but can also plot the histogram directly in a figure. Find out more about the command `hist` by using the Matlab help. `hist` only works with vectors in `double` format. Transform the matrix to one-dimensional vector and convert integers into doubles (with function `double`) before using `hist`! `hist` can have many optional arguments. Here, we only give the image (converted to one dimension) as first, and the number of bins (ca. 200) as second argument:

```
hist(im,nBins)
```

You should see a plot as shown in Figure 4.3. Add axis labels with the functions `xlabel` and `ylabel`.

Exercise 4.10 Generate a Nuclei Mask

From the histogram plot in Figure 4.3, you can see a large population of dark pixels and a smaller population of bright pixels, starting at an intensity value of about 4000. The dark pixels belong to the background noise, and everything above the intensity threshold 4000 is most likely nucleus region.

Let's define a variable for the intensity threshold that separates background and nuclei pixels:

```
thresh = 4000
```

Use `thresh` and `im` to generate a nucleus mask. A mask is a matrix of Booleans, with value 1 (TRUE) at nuclei region and value 0 (FALSE) at background region. Use logical operators for calculating the mask (have a look at Section 4.3.10 again). Visualize the mask with `imshow(mask,[])`.

Save the mask image with the `imwrite` function as TIF file. As arguments, you have to provide the matrix you want to save as image file, as well as the desired filename as string, for example:

```
imwrite(mask,'maskImage.tif','tif')
```

Refer to the MATLAB help for more information how to use `imwrite`.

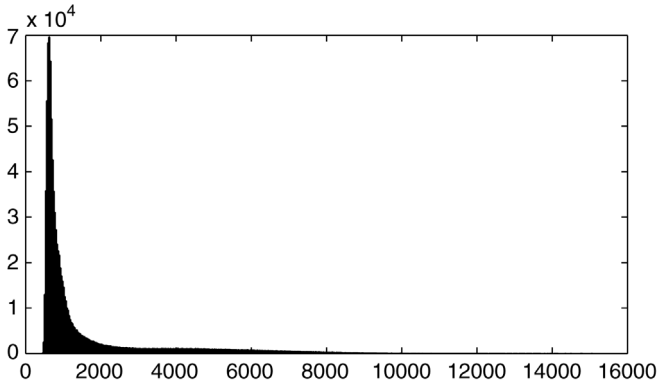


Figure 4.3 The gray value histogram of the nuclei image.

Exercise 4.11 Find Nucleus Objects

For the moment the mask matrix just gives us the information, which pixel belongs to foreground (nucleus region, value 1) and which pixel belongs to background region (value 0). In the next step, we will find and analyze individual nucleus objects in the image.

Both can be done with the `regionprops` function[2)]

```
stat = regionprops(mask)
```

The function produces an array of structures with fields Area, Centroid, and BoundingBox:

```
>> stat
stat =
343x1 struct array with fields:
    Area
    Centroid
    BoundingBox
>> stat(1)
ans =
           Area: 78
       Centroid: [2.9872 185.1154]
    BoundingBox: [0.5000 176.5000 6 18]
```

Each element of stat corresponds to a nucleus object in the image. The fields Area, Centroid, and BoundingBox are features of each nucleus object:

- *Area:* the number of pixels that belong to the nucleus object
- *Centroid:* the *x* and *y* coordinates of the center of the nucleus object
- *BoundingBox:* coordinates defining a bounding box around the nucleus object

2) The `regionprops` function actually performs two steps. First, connected components in the mask will be detected as separate objects and features for these objects (like area or centroid coordinates) are calculated. The object detection step can also be applied separately with the `bwconncomp` function. See Section 8.5.1.2 and Figure 8.7 for further illustration of the connected components function.

Mark each nucleus object with a blue cross as shown in Figure 4.4. Use the Centroid data as positions for plotting the cross.

Plot the centroid points on the nucleus image. This can be done by first using `imshow` and afterward the `plot` function within the same figure. After defining the figure, you have to use the command `hold on`. Otherwise the image in the figure will overwritten by the `plot` command:

```
1 figure(1) %create new figure window
2 hold on % set "hold" to avoid erasing old plots
3 imshow(image,[]); %display image
4 plot(x,y,'+'); %plot x and y positions as crosses
```

verb/code8.m

Hint: You have to execute the `plot` command inside a loop. One iteration for one nucleus object.

Further examples on how to determine and display properties of circular objects are given in Ref. [1].

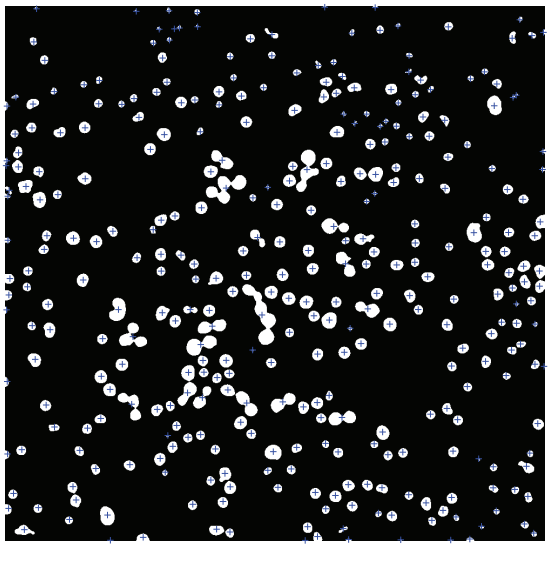


Figure 4.4 Detected nucleus objects.

Exercise 4.12 Plot Distribution of Nucleus Areas

Once we have our objects detected, we can analyze their features by statistical methods. Features could be, for example, intensity-based measures like the mean gray value for each object. A feature we have already computed with the regionprops function is the nr of pixels, that is, the area of the nuclei. By plotting a histogram, we get a first feeling how nucleus areas are distributed (see Figure 4.5). Therefore, first reformat the area data and put it into a plain vector called `areas`. Next we plot a histogram of our data in `areas` with the `hist` function.

```
%% Exrcercise 12: Plot distribution of nucleus areas

%generate a plain vector with nucleus areas
areas = zeros(nObjects,1);
for i=1:nObjects
    areas(i) = stat(i).Area;

end

nrBins =60
figure(5)
hist(areas, nrBins)
xlabel('area [nr of pixels]')
ylabel('nr of nucleus objects')
```

code/mod5_exercise7891011.m

Remember that we also used the hist function to plot the gray value distribution (see Figure 4.3). Since we have, for obvious reasons, much less nucleus objects than pixels in the image we choose a much smaller number of bins for the nucleus area histogram than we used for the gray value distribution.

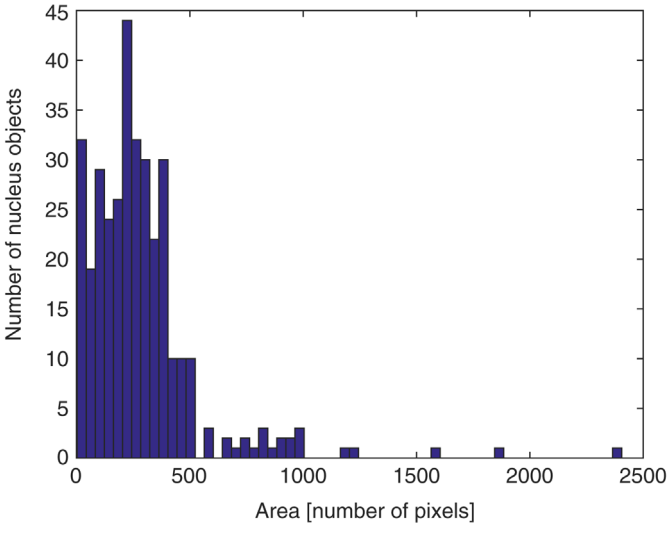


Figure 4.5 Histogram of nucleus areas.

Exercise 4.13 Filter Nucleus Objects by Area

By inspecting the histogram of gray values, you could recognize two populations of nuclei: a large population of small-sized nuclei, and a small population of larger nuclei greater than about 600 pixels.

To check how these larger objects actually look in the image, we plot a small circle on top of the centroids in Figure 4.4. We again construct a loop for plotting, but we plot it only if the condition `area > 600` is true:

```
%% Exercise 13: Filter nucleus objects by area
max_area = 600

figure(4)
for i=1:nObjects
    if stat(i).Area > max_area
        %mark each nucleus that is too large by a red
            circle
        plot(stat(i).Centroid(1),stat(i).Centroid(2),'
            or')
    end
end
```

code/mod5_exercise7891011.m

Note that you can plot in the same figure you have generated in Exercise 4.11. In Figure 4.6 you see a zoomed region of the resulting figure. It turns out that the large objects are mainly a result of bad object detection. In most cases, these objects do not relate to large single nuclei, but incorporate two or more touching nuclei.

To go on further, we could just exclude these objects. A more descent approach would be to split these touching objects, for example, by a morphological watershed (see e.g., Figure 5.7 in Chapter 5).

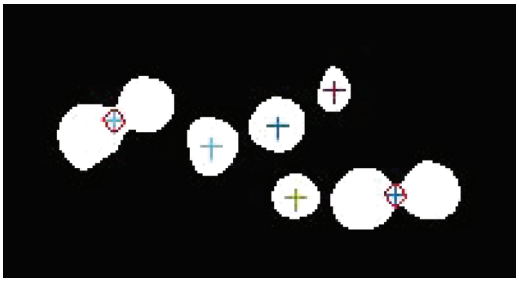


Figure 4.6 Detailed view on nucleus objects classified by an area threshold. The large objects (marked with circle) are mainly wrongly segmented objects. They constist of two or more touching nuclei.

4.5 Appendix

4.5.1 General Guidelines on Programming in Matlab

- include comments
  Write a short note at the top of each function/script. This will allow you to quickly understand the result and the sequence of calculations. Write comments for each block of calculations. This will help you to follow the details of the calculations – in particular when you are not using the function on a daily basis.
- choose variable names wisely
  Variable names should be short and meaningful, in order to be able to recognize its content without looking at it. Also, they should not be identical to the name of an existing function, otherwise the function cannot be executed anymore. Overwrite a variable only when you are sure that the former content is not needed anymore.
- clean up the workspace
  Using `clear ...` to delete individual variables from the workspace. This helps to keep better keep track of your variables and to use less memory.
- use matrix/vector-based calculations
  Whenever basic mathematical operations (+,−,∗,/) are applied, avoid element-by-element calculations, as done for example in a for-loop. Instead, rewrite the variables as vectors or matrices and then use them as a whole in the calculation.
- write small general function
  Try to split up a long task into several small functions, which are then run by the script. When the functions are written in a sufficiently generic way, they can easily be reused as building bricks of future algorithms.

4.5.2 List of Functions

Below you find a list all functions used in the tutorial, sorted by different categories:

4.5.2.1 Workspace and Command Window Functions

- clear all: delete all variables and functions in workspace
- disp: display text or array in command window
- save: save workspace variables to a mat file
- load: load variables from a mat file into the workspace

4.5.2.2 Handling Vectors and Matrices

- numel: number of elements of an array
- size: size of matrix dimensions
- find: Find indices and values of nonzero elements
- ind2sub: linear index to subscripts
- sub2ind: subscripts to linear index
- ones: create a matrix with all elements having the value of 1
- zeros: create a matrix with all elements having the value of 0

4.5.2.3 Figures and Plots

- plot: plot data points
- figure: create new figure window
- close: close figure window
- hold on: current figure window retains current graph
- hold off: next plot command erases the content of the figure window
- axis([xmin xmax ymin ymax]): define axis scaling
- length: number of elements along the largest dimension of an array
- strcat: concatenate strings
- mesh: display a matrix as 3D plot
- hist: create histogram from a vector/display a histogram
- xlabel: label the x axis of the current figure
- ylabel: label the y axis of the current figure
- subplot: create multiple plot windows in one figure

4.5.2.4 Conversions

- double: convert integers into doubles
- num2str: convert doubles or integers into strings
- struct2cell: convert a structure into a cell array
- cell2mat: concatenate cell array elements to a matrix

4.5.2.5 Statistics

- sum: sum of array elements
- min: minimum value of an array
- max: maximum value of an array
- abs: absolute value

4.5.2.6 Images

- imread: read a TIF image to a matrix
- imshow: display a matrix as an image

- imwrite: save a matrix to a TIF file
- regionprops: measure properties of image regions
- bwmorph: apply various morphological operations to an image
- imdilate: apply morphological dilation to an image
- imerode: apply morphological erosion to an image
- imopen: first apply erosion, then dilation
- im2bw: make a binary mask by applying an intensity threshold to an image
- edge: edge filter
- strel: create a structuring element (used as argument for morphological operations)

4.5.2.7 User Interaction

- uigetfile: user dialog for file selection

Solutions

Exercise 4.1

```
a= [3 17 45.3 9];

b= a(1)/a(3) * (a(3)+a(4))
```

Exercise 4.2

```
a=[1 2 3 4]
b=[5 6 7 8]

c=[a b];

%first solution
d=c(2:2:end)

%second solution
c(1:2:end)=[]
```

code/mod5_exercise2.m

Exercise 4.3

```
x=[1:0.01:10];
y=sin(x);

close all
figure(1)
plot(x,y)
```

code/mod5_exercise3.m

Exercise 4.4

```
m = zeros(5,5); %initialize matrix with zeros
m(:,1)=1;
m(:,2)=2;
m(1:3,3:end)=3;
m(4:end,3:end)=4
```

code/mod5_exercise4.m

Exercise 4.5

```
x=[0:0.01:15]
y=sin(x)

xpos=x(y>0);
ypos=y(y>0);

xneg=x(y<0);
yneg=y(y<0);

close all
figure(1)
hold on
plot(xpos,ypos)
plot(xneg,yneg,'r')
```

code/mod5_exercise5.m

Exercise 4.6

```
%module 5 exercise 6

datname1='data/nuclei.tif'
datname2='data/film7_stack.tif'

%get metadata
meta1=imfinfo(datname1);
meta2=imfinfo(datname2);

%nr of frames
nframes1=numel(meta1);
nframes2=numel(meta2);

%image dimensions
height1=meta1.Height;
width1=meta1.Width;
height2=meta2.Height;
width2=meta2.Width;
```

code/mod5_exercise6.m

Exercises 4.7–4.11

```
1
2  %% Exercise 7: import
3
4  datname='data/nuclei.tif';
5  im=imread(datname,1); % im is a 2d matrix containing
       gray values of the tif file
6
7  %% Exercise 8: show image
8
9  close all
10
11 imBright=im*2;%double gray values
12 figure(1)
13 imshow(imBright);%show image
14
15 figure(2)
16 imshow(im,[]); %optimize brightness without
       manipulating image gray values
17
18
19 %% Exercise 9: plot histogram
20
21 nbins=200; %number of bins
22 imdata=double(im(:));%conversion to doubles and one
       dimension
23
24 figure(3)
25 hist(imdata,nbins)
26
27 xlabel('gray value')
28 ylabel('nr of pixels')
29
30
31 %% Exercise 10: Generate a nuclei mask
32
33 thresh=4000;
34
35 mask=im>thresh;
36 imwrite(mask,'maskImage.tif','tif')
37
38 figure(4)
39 imshow(mask,[])
40
41
42 %% Exercise 11: Find nucleus objects
43
```

```
stat=regionprops(mask) %find objects and extract features

nObjects=numel(stat);%nr of detected objetcs
hold on
for i=1:nObjects
    plot(stat(i).Centroid(1),stat(i).Centroid(2),'+')
end


%% Exrcercise 12: Plot distribution of nucleus areas

%generate plain vector with nucleus areas
areas = zeros(nObjects,1);
for i=1:nObjects
    areas(i) = stat(i).Area;

end

nrBins =60
figure(5)
hist(areas, nrBins)
xlabel('area [nr of pixels]')
ylabel('nr of nucleus objects')


%% Exercise 13: Filter nucleus objects by area
max_area = 600

figure(4)
for i=1:nObjects
    if stat(i).Area > max_area
       %mark each nucleus that is too large by a red circle
       plot(stat(i).Centroid(1),stat(i).Centroid(2), 'or')
    end
end
```

code/mod5_exercise7891011.m

Reference

1 Fisher, N.I. (1993) *Statistical Analysis of Circular Data*. http://books.google.com/books/about/Statistical_Analysis_of_Circular_Data.html?id=wGPj3EoFdJwC.

5
FISH Spot Detection in Human Spermatozoids

Ulrike Schulze [1] and Sébastien Tosi[2]

[1]The Francis Crick Institute, Mill Hill Laboratory, The Ridgeway, Mill Hill, London NW7 1AA, United Kingdom
[2]Institute for Research in Biomedicine (IRB Barcelona), Advanced Digital Microscopy, Parc Científic de Barcelona, c/Baldiri Reixac 10, 08028 Barcelona, Spain

5.1
Overview

This module is divided into four separate steps. Each step has an enumerated workflow followed by an exercise. The result of each exercise is a macro. To ensure a smooth progress, the solution macro for each exercise is provided and should be used as the starting point for consecutive exercises.

5.1.1
Aim

FISH (fluorescence *in situ* hybridization) is a complex gene staining technique with numerous variants [1] where the quality of the staining depends on several physical parameters (e.g., level of DNA decondensation). FISH aims at labeling DNA sequences specific to a certain gene (or chromosome) so that they appear as bright fluorescent spots in fluorescent channels. We will write an ImageJ macro to process images from such a FISH assay: first to segment the spermatozoid nuclei from a DAPI staining, and then to classify the nuclei based on their chromosomal content (multiplicity of FISH spots in three different fluorescent channels).

5.1.2
Introduction

The automatic spermatozoids classification as proposed in Ref. [2] is a powerful means to extract statistics of chromosomal anomalies (see Figure 5.1) on large sample data sets (typically >10 000 cells). It can also be used to drive a motorized microscope to perform an "intelligent" scan [3]: The classification is performed

Bioimage Data Analysis, First Edition. Edited by Kota Miura.

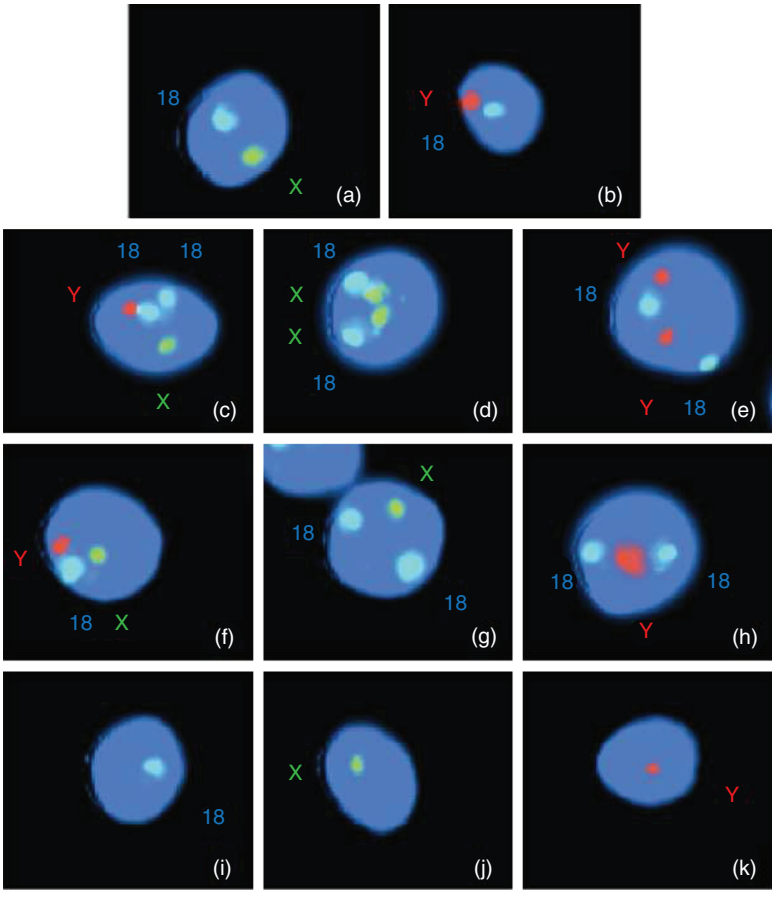


Figure 5.1 The upper cells are normal male and female spermatozoids, all the other cells are abnormal. (Courtesy of Anna Godo, University of Barcelona.)

from a low-resolution scan and a secondary scan (high resolution) is automatically triggered to only acquire the cells showing some specific anomalies.

5.1.3
Data Sets

The nuclei of the spermatozoids were DAPI stained and the chromosomes of interest (here X, Y, and 18) were stained by FISH. The fixed sample was scanned by a motorized stage microscope to tile the whole area of the sample. Several z slices were recorded for each fluorescent channel (DAPI, aqua, orange, and green). The analysis will be performed on the z maximum intensity projection of the images (in each channel).

5.2 Step 1: Initialization – A Short Warm-Up

5.2.1 Workflow

In Fiji, perform the steps described in this section with the command recorder open and create a macro from the recorder. We will use the image Small.tif.

This hyperstack (four channels) is a crop view of a single field from a confocal data set. Each image is a maximum intensity projection, per channel, of the original data. The first image is a staining for the nuclei, and all subsequent images are acquired in the fluorescence channels of the FISH stainings.

1) Open the image file in Fiji by `[File > Open . . . ]` or by drag and drop of the file on the Fiji bar.
2) Examine the hyperstack using the stack browser (slider).
3) Split channels to get independent images for each fluorescence channel `[Images > Color > Split Channels]`.
4) Remove the scale of the opened images by `[Analyze > Set Scale . . . ]`. Use the same settings as in Figure 5.2:
   - *Distance* and *known distance* to "0".
   - *Pixel aspect ratio* to "1".
   - The *unit length* should be "pixel."
   - *Global* should be ticked so that the settings apply to all images (and subsequently opened images).

   Alternatively, press the "click to remove scale" button.
5) Set Binary Options `[Process > Binary > Options . . . ]`. Ensure "Iterations" and "Count" both are set to 1 and that "Black background" and "Pad edges when eroding" are *not* ticked. EDM output should be set to overwrite.

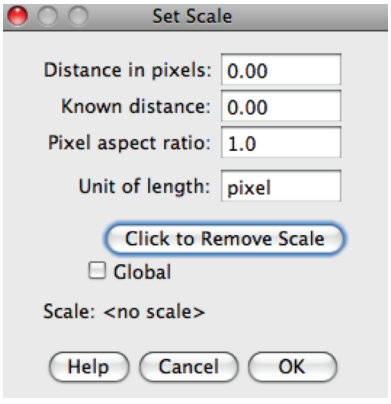


Figure 5.2 Initialization.

These settings correspond to ImageJ default settings and it is a good habit to always set them to predictable values at the beginning of a macro since they control the behavior of many commands that are commonly used.

6) Set Measurements `[Analyze > Set Measurements ...]`:
   - Area
   - Mean gray value
   - Shape Descriptors
   - *Redirect* should be set to "None"
   - *Decimal places* should be 2 or greater

Exercise 5.1

Create a macro (from the recorder) that performs the steps 3–6 of the previous workflow.[1] You can find the solution to the exercises as a complete macro in code/solutions/module3_01.ijm.

Note: We assume that the original hyperstack is already open when the macro is run.

```
1 // Input:
2 // - Hyperstack holding the 4 channels of the FISH assay
     (DAPI channel first)
3 // Output:
4 // - Split channels, no scale
```

code/solutions/module3_01.ijm

It is a good habit to add a preamble to a macro holding author, purpose, version, date and any helpful additional notes. Comments should also be added throughout the macro to summarize the aim of specific subsections (e.g., "Initialization" and "Erase the small particles"), especially if the sequence of commands is not straightforward to understand. These comments will often prove useful to people reading your code and to yourself when reading the code to modify it years after.

5.2.2 Summary of Tools Used in Step 1

Here is a summary of the main ImageJ tools used in this step:

- *Split channels* `[Images > Color > Split Channels]`.
  Split channels to get independent images for each fluorescence channel.

1) If you are using OSX, it sometimes happens that copying from the command recorder and pasting it to the script editor does not work. In that case, try using right mouse click (or control-click) to copy recorded commands. If this still does not work, then click the "Create" button at the top-right corner.

- *Set scale* `[Analyze > Set Scale ...]`.
  Allow calibration of the pixel size.
- *Binary Options* `[Process>Binary> Options ...]`.
  Set the behavior of binary image-related commands.
- *Set measurements* `[Analyze > Set Measurements ...]`.
  Set which features should be measured when calling `[Analyze > Measure]`.

5.3 Step 2: Segment Nuclei

Here, we will segment the nuclei in the image of the first channel "C1-Small.tif". We would like to ignore deformed (Figure 5.3d) and unusually small (Figure 5.3b) or large nuclei. In addition, the algorithm should identify touching nuclei (Figure 5.3c) so that these clusters are either ignored or properly split.

5.3.1 Workflow

We will first perform some manual processing on the image of the first channel to understand each step. After this we will write a macro to perform these operations automatically (see module 2 for an introduction of the macro programming language). Leave the command recorder open to record operations as you manually perform them.

1) Select Image (make window active). In step 1 we will split the channels of the original hyperstack. Now we have four independent images. To process the nuclei channel image, we need to make it "active" by clicking on its window: make "C1-Small.tif" active.

   Note: To select an image from a macro, we use the function selectImage ("name"), where "name" is the name of the image window. Alternatively, the identifier (ID) of an image can be passed to selectImage(). This ID must first

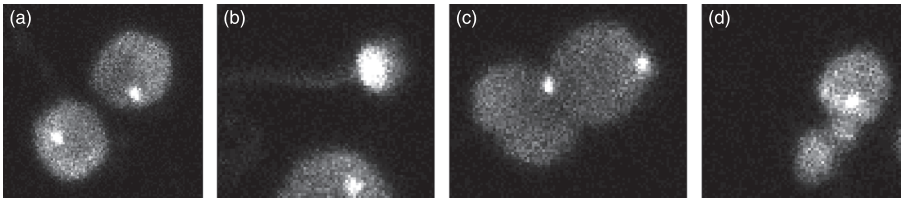


Figure 5.3 Typical images of nuclei in the DAPI channel. (a) Nicely shaped nuclei. (b) Small nuclei. (c) Some merged nuclei. (d) Deformed nuclei.

be retrieved by calling getImageID() at a step where the image is known to be "active."

2) Apply "Laplacian of Gaussian" (see section *Convolution* in Module 1 for an introduction to convolution filters). We use a Laplacian of Gaussian (Log) filter as a preprocessing step to facilitate the segmentation (see Section 5.3.2 for more information about the Log filter). In ImageJ, this filter can be found in `[Plugins > Feature Extraction> FeatureJ > FeatureJ Laplacian]`.

   The first step of the filter is a Gaussian filter whose radius (smoothing scale) must be adjusted to the typical size of the nuclei. The next step is a Laplacian filter. When the smoothing scale is properly adjusted, the nuclei appear as homogeneously dark and surrounded by a bright halo in the filtered image (see Figure 5.4b). A rule of thumb is to set the smoothing scale to about half the expected radius of the objects.

   Note that the output of the filter is a 32-bit image since the intensity values can be negative or positive. FeatureJ Laplacian can also detect zero-crossings, where the intensity changes sign (close to a sharp intensity transition), but we will not use this feature here (leave unticked). Ensure that "Compute Laplacian image" is ticked.

   To better understand the advantage of using a Laplacian prefiltering, you can try to directly apply a threshold on the original nuclei image.

   In Figure 5.4 we can observe the result of the Log filter performed on the nuclei image. The shapes of the nuclei are nicely mimicked in rings of different intensities (Figure 5.4c), as it can be visualized by changing the LUT to a colored LUT (`[Image> LookUp Tables]` to get the list of available LUTs). Try to optimize the smoothing scale of FeatureJ Laplacian to obtain a result similar to Figure 5.4.

3) Set threshold and convert to mask. Here, we will convert this image into a binary image by thresholding (see section *Thresholding* in Module 1). We

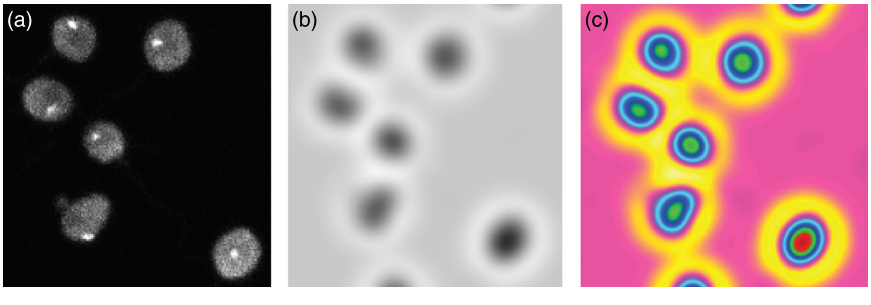


Figure 5.4 Image before and after the application of FeatureJ Laplacian. (a) Before application. (b) After application. (c) After applying a special LUT ("6_shades") to show the iso-intensity levels.

need to set the bounds of the threshold to separate the nuclei from the background. To achieve this, we use the command:

```
[Image > Adjust > Threshold . . . ]
```

Ensure to untick "Dark Background" and "set background pixels to NaN" (appears as pop-up window when clicking "Apply") to obtain a regular 8-bit binary image (mask). Thanks to preprocessing, the nuclei can now readily be segmented by thresholding the pixels with negative values (up to a small negative value) in the filtered image. The result is a black and white image, in which all pixels that belong to an object have an intensity value of 255, while all pixels that belong to the background have an intensity value of 0.

Note: After converting the image to binary image, ImageJ automatically applies (by default) a LUT inversion: The objects now appear black on a white background. See also http://imagej.nih.gov/ij/docs/guide/146-29.html#infobox:InvertedLutMask

4) Fill holes `[Process > Binary > Fill Holes]`. Some objects of the binary image might have holes (see Figure 5.5). A hole is defined as a group of pixels belonging to the background (white pixels) surrounded by pixels belonging to the foreground (black pixels). Since we do not expect the nuclei to exhibit any hole, we use `[Process > Binary > Fill Holes]` to fill them in (see section *Morphology* in Module 1).
5) Dilate `[Process > Binary > Dilate]`. From Figure 5.6a we can see that the detected boundaries mostly follow the contours of the nuclei but they sometimes overlap with them. This may become a problem later on when intending to detect a FISH spot close to the boundary of a nucleus. This problem can be mitigated by enlarging the segmented nuclei by morphological dilation (see section *Morphology* in Module 1). Each round of dilation enlarges the objects by one pixel, several dilations can be performed sequentially.

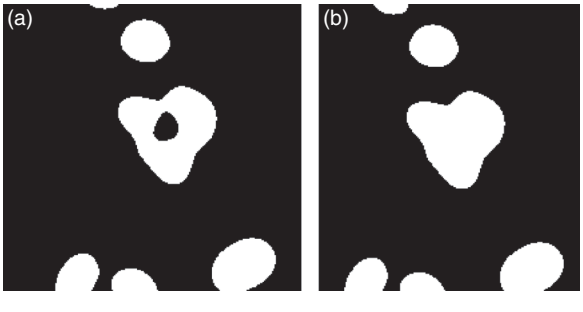


Figure 5.5 Image before and after the application of Fill Holes (image video inverted). (a) Before application. (b) After application.

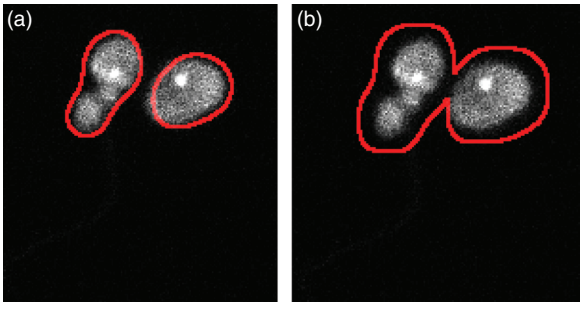


Figure 5.6 Overlay of original image and segmented binary image before and after several dilations (for better visibility only the boundaries of the segmented images are represented). (a) Before dilation. (b) After 4× dilation.

6) Watershed `[Process > Binary > Watershed]`. Try the binary watershed command to separate touching nuclei (Figure 5.7).
7) Analyze particles. At this point we obtained a binary image holding the segmented nuclei. Using `[Analyze > Analyze Particle]`, we can identify and measure several properties of these connected particles. This command also allows us to exclude an object based on its geometry. For our purpose, we want to exclude both the deformed particles and particles that are too small or too big.

   To exclude deformed particles, we can measure their circularity. The circularity describes how closely an object resembles a circle by computing the ratio between its area and its square perimeter. A perfect circle has a circularity parameter = 1. Any other object will have a circularity parameter smaller than 1, but greater than 0 (an infinite line).

   Particles that are smaller or larger than the given critical areas can also be excluded. The area bounds should be first determined empirically: You can do so by measuring the area of a typical nucleus and setting the lower and upper bounds to, for instance, 0.66× and 1.5× this value.

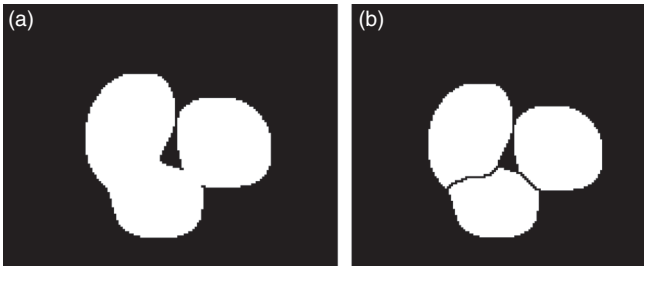


Figure 5.7 Separating nuclei with watershed (image video inverted). (a) Before separation. (b) After separation.

Finally, particles touching a border can be easily discarded by ticking "Exclude on edges" in `[Analyze > Analyze Particles . . . ]`. The purpose is to analyze FISH spots by nucleus; do not forget to also tick "Add to Manager" to add the nuclei analyzed to the ROI Manager so that they can easily be accessed further on.

Exercise 5.2

Following the workflow described above, write a macro to segment the nuclei in the image "C1-Small.tif." The solution to this exercise is provided in code/solutions/module3_02simple.ijm.

Note: The lower bound of the threshold should be set to the minimum intensity of the image; search for a macro function allowing to retrieve this value.

Exercise 5.3

The lower and upper bounds of the nuclei area have been so far empirically set; we will now automate the estimation of these bounds. For this we will first analyze the particles after thresholding without setting any area bounds (do not add the particles to the ROI manager at this point). The areas of the analyzed particles will be measured to results table and copied to an array to be further processed (this can be done by writing a loop). Assuming that valid nuclei are in majority, try to figure out a way to estimate the lower and higher area bounds from the area measurements. Finally, we will analyze the particles again, but this time setting the lower and upper area bounds (and adding them to the ROI manager).

Hint: A useful function that we will make use of is Array.sort(MyArray). The median can be computed by sorting the *n* areas and selecting the *n*/2th area.

Using this technique we can analyze images that have been taken with a wide variety of magnifications without having to manually adapt the "typical" area. You can find the solution to both exercises as a complete macro in code/solutions/module3_02.ijm.

```
1 // Input:
2 // - 4 channels of the FISH experiment in 4 images:
     C1-Small.tif, C2-Small.tif, C3-Small.tif and
     C4-Small.tif
3 // Output:
4 // - Detected nuclei in ROI manager
```

code/solutions/module3_02.ijm

5.3.2 Summary of Tools Used in Step 2

- *FeatureJ Laplacian* `[Plugins > Feature Extraction > FeatureJ > FeatureJ Laplacian].` http://imagescience.org/meijering/software/featurej/

  After applying a Gaussian filter with radius defined by the smoothing scale, this command computes the sum of the second-order spatial derivatives of the intensity along the Cartesian directions at each pixel.

  It is used to emphasize large isotropic intensity curvature (domes) in an image. This combination of filters (Gaussian followed by Laplacian) is called "LoG" for Laplacian of Gaussian and is commonly used for spot or blob enhancement. The optimal smoothing scale is directly related to the radius of the blob-like objects to be enhanced. The LoG is ubiquitous in image processing and was popularized in Ref. [4] for feature detection in the framework of the scale-space theory.
- *Convert to Mask* `[Image > Binary > Convert to Mask].`

  Convert a grayscale image into a binary (black and white) image, the active threshold is used to define whether a pixel is part of the foreground or of the background.
- *Set threshold* `[Image > Adjust > Threshold . . . ].`

  Set the values of the threshold bounds.
- *Fill holes* `[Process > Binary > Fill Holes].`

  Fill holes in the connected particles (objects) of a binary image.
- *Analyze particles* `[Analyze > Analyze Particles . . . ].`

  Find the connected particles in a binary image and optionally filter them (keep/discard) based on their area, geometric properties, or location (touching an edge of the image).
- *Dilate* `[Process > Binary > Dilate].`

  Enlarge the objects in a binary image. This command enlarges the boundaries of the objects by one pixel.
- *Watershed* `[Process > Binary > Watershed].`

  Intent to split touching/overlapping particles to individual particles in a binary image.

5.4 Step 3: FISH Spots Detection

Again we will perform a sequence of image processing steps manually and then include them in a macro. The starting point is to have the channels split and the nuclei stored in the ROI manager (if you lose the information at any time, you can sequentially launch the macros from steps 1 and 2 on the original hyperstack to get to this point).

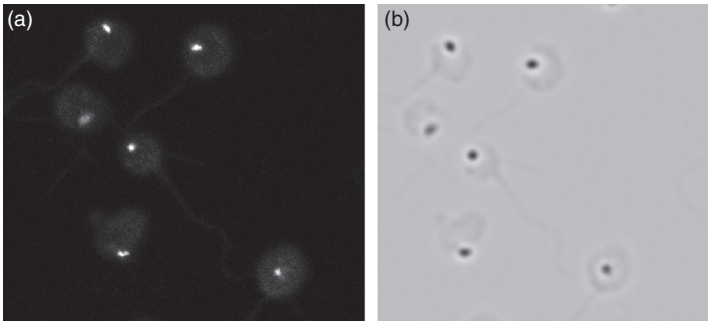


Figure 5.8 A FISH channel before and after applying the Log filter. (a) Before application. (b) After application.

5.4.1 Workflow

In this section, we will segment all the spots in the three FISH channels. To detect the spots, we use a similar prefiltering as for the segmentation of the nuclei (with a different smoothing scale).

1) Apply Laplacian of Gaussian. We prefilter the image of the first FISH channel with `[Plugins > Feature Extraction > FeatureJ > FeatureJ Laplacian]` (Figure 5.8). The smoothing scale should be adjusted to the size of the spots!
2) Detect intensity regional minima with `[Process > Find Maxima . . . ]`. You should tick "Light background" to detect minima and adjust "Noise tolerance" to optimize detection. Select "Single Points" as "Output Type" to create a binary mask with detected minima.
3) Count spots inside nuclei. In step 2 we have already segmented the nuclei and stored them to the ROI manager. Now, we simply need to select the binary mask holding the detect spots and loop through the nuclei ROIs. Next, we count the number of pixels with intensity equal to 255 to retrieve the number of spots per nucleus.

   For each nucleus we will measure the statistics of the intensity with the macro function getRawStatistics(nPixels, mean, min, max, std, histogram), which among others returns the histogram of the pixel intensity inside the active selection.

Exercise 5.4

Complete the macro code/module3_03simple-incomplete.ijm to perform the previous sequence on the three FISH channels. Before launching the macro you need to have C2-Small.tif opened and the detected nuclei stored in the ROI Manager.

Open, read, and test the macro. The macro automatically segments the FISH spots following the workflow we previously described, but the part counting the spots in each nucleus is incomplete. Several tasks have to be performed:

1. Write the missing code to detect the intensity regional minima. If you perform it right, you should see a list of values indicating the number of spots detected in each nucleus in the Log window. Solution is in code/solutions/module3_03simple.ijm.
2. Modify the code to automatically repeat the same workflow on the other two channels. This time you need the three channels opened before running the code. Solution is in code/solutions/module3_03.ijm.

Hint: You will need a loop over the channels. To properly select the correct image at each iteration you can make use of string concatenation (the channel images are called "C2-Small.tif," "C3-Small.tif," and "C4-Small.tif").

```
1 // Input:
2 // - ROI Manager with nuclei selection
3 // - C1-Small.tif, C2-Small.tif, C3-Small.tif and
     C4-Small.tif opened
4 // Output:
5 // - Spot segmentation masks of the 3 channels
6 // - 3 Arrays (on per channel) with spot counted in
     each nucleus
```

code/solutions/module3_03.ijm

5.4.2 Summary of Tools Used in Step 3

- *FeatureJ Laplacian*: See step 2
- *Threshold*: See "ImageJ Basics."
- *Analyze particles:* See "ImageJ Basics."
- *ROI Manager:* See "ImageJ Basics."

5.5 Step 4: Visualization – Making It Look Pretty

5.5.1 Workflow

In this section we will visualize the nuclei and spots previously detected by overlaying them to the original image. The spot markers should be color coded according to their channel and the nuclei boundaries should be color coded

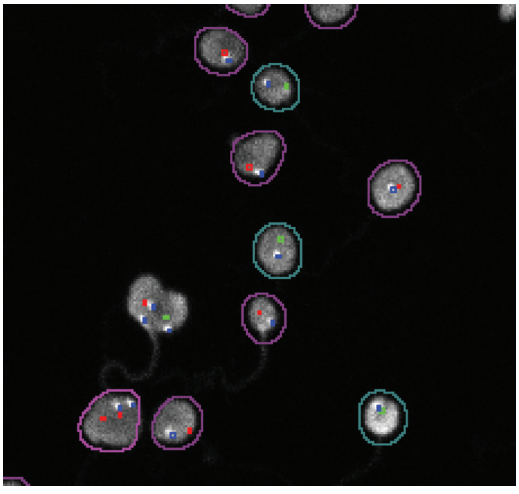


Figure 5.9 All analyzed nuclei are marked with a color code depending on the number and type of FISH spot found inside them.

according to the number of FISH spots detected in each FISH channel (Figure 5.9).

1) Overlaying the FISH spots on the original image. Select "SpotCandidates," the binary image holding the detected spots, threshold the image, and then call `[Edit > Selection > Create Selection]`. The selection can then be transferred (restored) to another image using `[Edit > Selection > Restore Selection]` or saved to the ROI manager. To make the spots more visible, we increase their size in the selection with `[Edit > Selection > Enlarge . . . ]`. You can try to restore the selection on the original image to check the accuracy of the detection.
2) Coloring the nuclei according to the number and type of FISH spots.

   The color of the nuclei should be set according to their multiplicity of spots. For example, a nucleus with two red spots and two blue spots should have a different color than a nucleus with one red spot and one green spot.

   To achieve this, we take advantage of the fact that all colors are displayed as a combination of red, green, and blue. How much of red, green, and blue are used for a color is described by a number from 0 to 255. This number is written as a hexadecimal number (from 00 to FF). In this system, white is coded as FFFFFF (red: intensity 255, green: intensity 255, and blue: intensity 255). Black is coded as 000000 (red: intensity 0, green: intensity 0, and blue: intensity 0). Pure red would be coded as FF0000 (red: intensity 255, blue: intensity 0, and green: intensity 0) and purple would be coded as FF00FF (red: intensity 255, green: intensity 0, and blue: 255).

To color the nuclei, we will loop again through their selections in the ROI manager and assign them a specific stroke color depending on the spot counts. As we do not expect the number of spots to exceed 2 per nucleus, the color will be simply defined as follows:

- red: 63 + 64 times the number of spots in channel 1
- green: 63 + 64 times the number of spots in channel 2
- blue: 63 + 64 times the number of spots in channel 3.

Exercise 5.5

Starting from the macro code/module_03.ijm and following the steps described in the section "Marking the FISH spots," add the missing code to show an overlay of the detected spots (store the selection to the ROI manager with `[Edit > Selection > Add to Manager]`). The code should be added at the end of the loop over the FISH channel.
Hint: You should start by reducing the upper limit of the loop so that only the first channel is processed. Then try to make your macro run only over the three channels (this is a typical debugging trick).

Now try to implement the steps of the section "Marking the nuclei according to the number and type of FISH spots."

Hint: To change the color of a selection of the ROI Manager, you will have to select it first with roiManager("select," . . .) and then call `Edit > Selection > Properties` and update the selection in the ROI Manager with roiManager ("update"). To understand the parameters that should be passed to the command, record it and input a stroke color of the type FFxxxxxx where the xx stands for each of the three hexadecimal values of the RGB color channels (the first field codes the transparency, we set it to FF that is equivalent to nontransparent).

Note: The macro function toHex converts a decimal number to its hexadecimal representation.

Finally, open the solution code/solutions/module3_03-04.ijm to the previous exercise and test it. As you will notice, the overlay of the spots is not associated with the slice (channel) in which they were detected, but they all appear simultaneously. By uncommenting the last section of the code, you can modify this behavior – understand how it works!

5.5.2 Summary of Tools Used in Step 4

- *Create selection* `[Edit > Selection > Create Selection]`.
  This command creates a selection from a binary image (foreground selected).
- *Restore selection* `[Edit > Selection > Restore Selection]`.

Restore the last selection that was active. Shortcut key: Shift-Ctrl-E (Windows), Shift-Cmd-E (Mac), Shift-E (if control or command key is specified as not required)

- *Enlarge* `[Edit > Selection > Enlarge]`.
  Enlarge (dilate) a selection by a given number of pixels.
- *ROI Manager:* See "ImageJ Basics."

Solutions to the Exercises

Exercise 5.1

```
1  // Input:
2  // - Hyperstack holding the 4 channels of the FISH assay
      (DAPI channel first)
3  // Output:
4  // - Split channels, no scale
5
6  // Split channels
7  run("Split Channels");
8
9  // Initialization
10 run("Set Scale...", "distance=0 known=0 pixel=1 unit=pixel
      global");
11 run("Options...", "iterations=1 count=1 edm=Overwrite");
12 run("Set Measurements...", "area mean centroid shape
      redirect=None decimal=2");
```

code/solutions/module3_01.ijm

Exercise 5.2

```
1  // Input:
2  // - 4 channels of the FISH experiment in 4 images:
      C1-Small.tif, C2-Small.tif, C3-Small.tif and C4-Small.tif
3  // Output:
4  // - Detected nuclei in ROI manager
5
6  //Segment Nuclei
7  selectImage("C1-Small.tif");
8  run("FeatureJ Laplacian", "compute smoothing=12");
9  getMinAndMax(min,max);
10 setThreshold(min,-0.05);
11 run("Convert to Mask");
12 run("Fill Holes");
13 for(i=0;i<2;i++)run("Dilate");
14
15 //Split Particles
16 run("Watershed");
17 rename("Mask");
18
19 //Analyze Particles and store to ROI manager
```

```
run("Analyze Particles...", "size=1000-2500 circularity=0.75-
   1.00 show=Nothing display exclude clear include add");
NbNuclei = roiManager("count");

selectImage("Mask");
close();
selectImage("C1-Small.tif");
roiManager("Show All");
```

code/solutions/module3_02simple.ijm

Exercise 5.3

```
// Input:
// - 4 channels of the FISH experiment in 4 images:
   C1-Small.tif, C2-Small.tif, C3-Small.tif and C4-Small.tif
// Output:
// - Detected nuclei in ROI manager

//Segment Nuclei
selectImage("C1-Small.tif");
run("FeatureJ Laplacian", "compute smoothing=12");
getMinAndMax(min,max);
setThreshold(min,-0.05);
run("Convert to Mask");
run("Fill Holes");
for(i=0;i<2;i++)run("Dilate");

//Split Particles
run("Watershed");
rename("Mask");

//Analyze particle to estimate median area
run("Analyze Particles...", "size=0-Infinity circularity=0.75-
   1.00 show=Nothing display exclude clear include");
Area = newArray(nResults);
for(i=0;i<nResults;i++)Area[i] = getResult("Area", i);
Area = Array.sort(Area);
MedianArea = Area[nResults/2];
print("Median area: "+d2s(MedianArea,0));

//Analyze Particles and store to ROI manager
run("Analyze Particles...", "size="+MedianArea*0.66+"-"+
   MedianArea*1.5+" circularity=0.75-1.00 show=Nothing
   display exclude clear include add");
NbNuclei = roiManager("count");

selectImage("Mask");
close();
selectImage("C1-Small.tif");
roiManager("Show All");
```

code/solutions/module3_02.ijm

Exercise 5.4

```
// Input:
// - ROI Manager with nuclei selection
// - C1-Small.tif, C2-Small.tif, C3-Small.tif and C4-Small.tif opened
// Output:
// - Spot segmentation masks of the 3 channels
// - 1 Arrays with spot counted in each nucleus in channel 1

NbNuclei = roiManager("count");

// FISH spots detection
NbSpotsChan1 = newArray(NbNuclei);

// Pre-filtering
selectImage("C1-Small.tif");
run("FeatureJ Laplacian", "compute smoothing=3");
SpotLapID = getImageID();

// Spot segmentation
run("Find Maxima...", "noise=4 output=[Single Points] light");
SpotCandMaskID = getImageID();

// Spot count in each nucleus
selectImage(SpotCandMaskID);
for(j=0;j<NbNuclei;j++)
{
  roiManager("select",j);
  getRawStatistics(nPixels, mean, min, max, std, histogram);
  NbSpots = histogram[255];
  NbSpotsChan1[j] = NbSpots;
}

// Display arrays with counted spots per nucleus
print("Array NbSpotsChan1:");
Array.print(NbSpotsChan1);

run("Select None");
```

code/solutions/module3_03simple.ijm

```
// Input:
// - ROI Manager with nuclei selection
// - C1-Small.tif, C2-Small.tif, C3-Small.tif and C4-Small.tif opened
// Output:
// - Spot segmentation masks of the 3 channels
```

```
// - 3 Arrays (on per channel) with spot counted in each
   nucleus

NbNuclei = roiManager("count");

// FISH spots detection
NbSpotsChan1 = newArray(NbNuclei);
NbSpotsChan2 = newArray(NbNuclei);
NbSpotsChan3 = newArray(NbNuclei);
for(i=1;i<4;i++)
{
  // Pre-filtering
  selectImage("C"+d2s(i+1,0)+"-Small.tif");
  run("FeatureJ Laplacian", "compute smoothing=3");
  SpotLapID = getImageID();

  // Spot segmentation
  run("Find Maxima...", "noise=4 output=[Single Points]
     light");
  SpotCandMaskID = getImageID();

  // Cleanup
  selectImage(SpotLapID);
  close();

  // Spot count in each nucleus
  selectImage(SpotCandMaskID);
  for(j=0;j<NbNuclei;j++)
  {
    roiManager("select",j);
    getRawStatistics(nPixels, mean, min, max, std, histogram);
    NbSpots = histogram[255];
    if(i==1)NbSpotsChan1[j] = NbSpots;
    if(i==2)NbSpotsChan2[j] = NbSpots;
    if(i==3)NbSpotsChan3[j] = NbSpots;
  }

}
run("Select None");

// Display arrays with counted spots per nucleus
print("Array NbSpotsChan1:");
Array.print(NbSpotsChan1);
print("Array NbSpotsChan2:");
Array.print(NbSpotsChan2);
print("Array NbSpotsChan3:");
Array.print(NbSpotsChan3);
```

code/solutions/module3_03.ijm

Exercise 5.5

```
// Input:
// - ROI Manager with nuclei selection
// - C1-Small.tif, C2-Small.tif, C3-Small.tif and C4-Small.tif opened
// Output:
// - Overlay of detected spots (channel colored)
// - Nuclei selection with color based on spot content

roiManager("Associate", "true");
NbNuclei = roiManager("count");

// FISH spots detection
NbSpotsChan1 = newArray(NbNuclei);
NbSpotsChan2 = newArray(NbNuclei);
NbSpotsChan3 = newArray(NbNuclei);
for(i=1;i<4;i++)
{
  // Pre-filtering
  selectImage("C"+d2s(i+1,0)+"-Small.tif");
  run("FeatureJ Laplacian", "compute smoothing=3");
  SpotLapID = getImageID();

  // Spot segmentation
  run("Find Maxima...", "noise=4 output=[Single Points] light");
  SpotCandMaskID = getImageID();

  // Cleanup
  selectImage(SpotLapID);
  close();

  // Spot count in each nucleus
  selectImage(SpotCandMaskID);
  for(j=0;j<NbNuclei;j++)
  {
    roiManager("select",j);
    getRawStatistics(nPixels, mean, min, max, std, histogram);
    NbSpots = histogram[255];
    if(i==1)NbSpotsChan1[j] = NbSpots;
    if(i==2)NbSpotsChan2[j] = NbSpots;
    if(i==3)NbSpotsChan3[j] = NbSpots;
  }

  // Spots overlay
  setThreshold(1,255);
  run("Create Selection");

  //Check if selection
```

```
47    if(selectionType>-1)
48    {
49      run("Enlarge...", "enlarge=1 pixel");
50      roiManager("add");
51      roiManager("select",roiManager("count")-1);
52      if(i==1)roiManager("Set Color", "red");
53      if(i==2)roiManager("Set Color", "green");
54      if(i==3)roiManager("Set Color", "blue");
55      roiManager("Deselect");
56    }
57    selectImage(SpotCandMaskID);
58    close();
59  }
60
61  // Draw color coded outlines of the nuclei
62  selectImage("C1-Small.tif");
63  for(j=0;j<NbNuclei;j++)
64  {
65    roiManager("select",j);
66    ColorCode = "FF"+toHex(63+64*NbSpotsChan1[j])+toHex
        (63+64*NbSpotsChan2[j])+toHex(63+64*NbSpotsChan3[j]);
67    run("Properties... ", "name=Nuc"+d2s(j,0)+"
        stroke="+ColorCode+" width=1 fill=none");
68    roiManager("update");
69  }
70
71  run("Images to Stack", "name=Stack title=[] use");
72  roiManager("Show All without labels");
73
74  // Associate spots to slice
75  for(j=2;j<=4;j++)
76  {
77    roiManager("select",roiManager("count")-5+j);
78    setSlice(j);
79    roiManager("update");
80  }
81  setSlice(1)
82  roiManager("Show All without labels");
```

code/solutions/module3_03-04.ijm

5.6 Assignments

1) Assemble all the sections of the macro and add a dialog box at the beginning to have an easy access to all the detection parameters.

2) Test the macro on the complete field of view (FISH-MIP-Confocal63x.tif) or a larger image (FISH-MIP-Widefild20x-Montage.tif) montaged from widefield images. In each case, adjust the parameters to get a satisfying detection of the nuclei and the spots.

   Hint: Indicative detection parameters (confocal/widefield) – nucleus smooth (12/5), nucleus sensitivity (−0.05/−0.05), spot smoothing (3/2), spot background radius (5/4), and spot level for all channels (25/25).

3) Log the frequency (the number of nuclei over the total number of nuclei detected) for some specific user-defined spot combinations. You should find that there are two large populations of normal male and female spermatozoids in the confocal data set (1/0/1 and 0/1/1), while a majority of nuclei have a single spot in each channel in the widefield data set (1/1/1) and many have anomalies.

   Solution: A complete macro is provided as solution to the three assignments: code/solutions/FISHNucleiAnalyzer.ijm.

Acknowledgment

We are grateful to Anna Godo (Universidad Autonoma de Barcelona (UAB)) for sharing some of her data sets and accepting to expose part of the methodology involved in her own research work.

References

1 Volpi, E.V., Bridger, J.M. *et al.* (2008) FISH glossary: an overview of the fluorescence *in situ* hybridization technique. *Biotechniques*, **45** (4), 385–386.

2 Molina, Ò., Sarrate, Z., Vidal, F., and Blanco, J. (2009) FISH on sperm: spot-counting to stop counting? Not yet. *Fertil. Steril.*, **92** (4), 1474–1480.

3 Tosi, S. *et al.* (2012) ImageJ-driven intelligent high content screening. Euro-Bioimaging Workshop 2012, ISBI Barcelona, Spain.

4 Lindeberg, T. (1993) *Scale-Space Theory in Computer Vision*, Kluwer Academic Publishers.

6
Analysis of Microtubule Orientation

Kota Miura,[1] Thomas Pengo,[2] Simon F. Nørrelykke,[3] and Christoph Möhl[4]

[1]European Molecular Biology Laboratory, Meyerhofstraße 1, 69117 Heidelberg, Germany
National Institute of Basic Biology, Okazaki, 444-8585, JAPAN
[2]University of Minnesota, University of Minnesota Informatics Institute, Cancer and Cardiovascular Research Building, 2231 6th St SE, Minneapolis, MN 55445, USA
[3]ETH Zurich, Scientific Center for Optical and Electron Microscopy (ScopeM), Image and Data Analysis (IDA) Unit, HPI F15, Wolfgang-Pauli-Strasse 14, 8093 Zurich, Switzerland
[4]German Center of Neurodegenerative Diseases (DZNE), Image and Data Analysis Facility (IDAF), Core Facilities, Holbeinstraße 13–15, 53175 Bonn, Germany

6.1
Aim

Using several image processing and analysis tools, we explore strategies for quantifying the microtubule (MT) polarity within cultured cells. The reader learns how to track the movement of spotty signal in image sequences using ImageJ and how to analyze the directionality of movement and treat them statistically using R or Matlab.

6.2
Introduction

Regulation of cytoskeletal orientation is a basic mechanism for controlling cell polarity and the dynamics of coordinated single-cell and multicellular movement. In this chapter, we explore an analysis protocol for studying MT orientation within cultured cells using time-lapse sequences.

We use a two-dimensional time-lapse sequence of microtubule binding protein EB1 (Figure 6.1). As this protein transiently binds to the growing end of microtubule, its signal appears as if it is moving along with the plus end of microtubules. Using this characteristic, the polarity of microtubules within cells could be measured by tracking EB1 and then calculating the direction of their movement. For general reviews on tracking techniques in cell biology, see Refs. [1–3].

Bioimage Data Analysis, First Edition. Edited by Kota Miura.

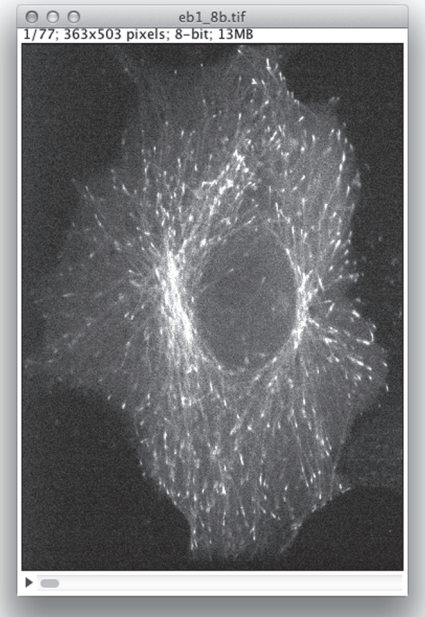


Figure 6.1 EB1-labeled cultured cell.

The directionality analysis we use in this protocol is based on the circular statistics [4]. This is a special statistics for circular data such as angles and dates. As an example to understand why such a special statistics is required, we could think about measurement results with two angles: the first is 1° and the second is 359°. If we average these results by summing them up and dividing them by 2, the result is 180° and not really the mean direction of two given angles. We know that the mean direction should be 0°. To have this correct mean direction, we need circular statistics.

Circular statistics in biology is a valuable method to treat directionality data, which is a rarely analyzed aspect of particle tracking results compared with the speed of movement. We hope that this practical course will be a concise introduction for using the technique to invoke your interest in circular statistics.

For the tracking of EB1 movement (step 1), we use ImageJ and a plug-in for particle tracking developed by MOSAIC group at ETH (Zurich) and MPI-CBG (Dresden). For the analysis of microtubule orientation, we use Matlab to learn how to calculate circular mean and also to estimate multiple preferred directions from tracking data (step 2). For those who want to know more details about circular statistics, a longer section with analysis using R is available (step 3). It is your choice to do step 2 or 3, but it is recommended to start reading step 2 first to know the principle of getting vectors and angles out of tracking results.

Note that the Matlab analysis (step 2) and the R analysis (step 3) have a difference in the way microtubule orientations are calculated. In the Matlab analysis, orientation of each particle trajectory is computed, while in the R analysis orientation of the movement for each time step is computed. The former method depends more on the quality of tracking than the latter method, because the latter method only requires successful linking between two time points.

Step 2 was designed and written by CM, TP, and SN. Other parts are designed and written by KM.

6.3 Data Set

The data we use consist of an image sequence of Vero cell (kidney epidermal cells from African green monkey *Chlorocebus* sp.). This EB1-labeled sequence was kindly provided by Emmanuel Reynaud. Spinning disk confocal laser microscope was used. Capturing interval was 30 s.

6.4 Step 1: Tracking

We first track EB1 signals using the ImageJ Particle Tracker plug-in. The result of this will be a table of *XY* coordinates of EB1 signals in each frame.

The Particle Tracker plug-in we use has two major steps within its algorithm: segmentation and linking. This can be intuitively understood by its interface (Figure 6.2). The upper half is the "Particle Detection" part and the bottom half is the "Particle Linking" part.

6.4.1 Particle Detection (Segmentation)

When we track "particle" using tracking tool, computer should be provided with the definition of "particle," for example, size, brightness, and shape. In the plug-in we use, "particle" definition is based on three parameters:

1) *Radius*: Expected diameter of dot to be detected in pixels.
2) *Cutoff*: A cutoff level for the non-particle discrimination criteria, a value calculated for each particle based on intensity moment orders 0 (m_0) and 2 (m_2). For a larger value, a more strict evaluation for the particle identity is made so that more particles will be discarded from the particle listing.

   More details: m_0 is the total intensity of the particle and m_2 is the total intensity weighted by the squared distance from the centroid of the particle normalized by the total intensity. In other words, m_2 represents the distribution of intensity. For example, for two particles with same total intensity, the

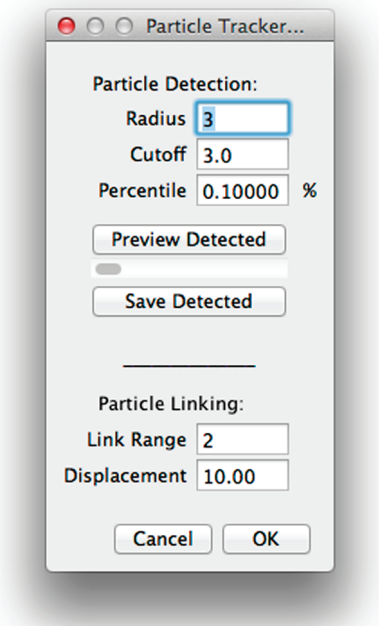


Figure 6.2 Particle Tracker interface.

one that is larger and dimmer has a larger m_2 value. The non-particle discrimination criteria check whether a certain particle is similar with the m_0 and m_2 values of other particles, and if this similarity is below a critical value (the *cutoff*) then that particle would be evaluated as non-particle. Further details could be found in equations and Figure 5 on page 13 of Ref. [5].[1)]

3) *Percentile*: The larger the value, the more particles with dark intensity will be detected. It corresponds to the area proportion below the intensity histogram in the upper part of the histogram.

The plug-in searches through all bright spotty signals and tests whether each particle matches to the parameters that you provided. Signals that are in agreement with the given conditions will be marked as "particle," else they are discarded. This is called *particle detection*, a type of segmentation.

6.4.2 Particle Linking

After the particle detection, there is a list of particles from all frames (you will not see the list until tracking finishes). At this point, particles are only detected,

1) The code for the non-particle discrimination is from line 1172 in the source (https://github.com/cmci/ParticleTrackerExt/blob/master/src/main/java/ij/ParticleTracker_.java).

so there are no tracks yet. From one frame to the next frame, we expect that each particle moves a certain distance and shows up in a new position. However, there is no knowledge yet to link a particle in the current frame to a particle in the successive frame to identify them as a single particle at different time points. The *particle linking* does this job. The easiest way to link a particle to the successive frame is the nearest neighbor method. For a particle in the nth frame, the nearest neighbor method links the particle to a particle in the $(n + 1)$th frame that is the shortest in distance.

This method might work well if the particle density is low, but when the particle density is high, this method encounters a problem called "crossing." If two particles A and B are crossing on their ways such that at some point they are very close to each other, the nearest neighbor method might wrongly link particle A to particle B in the next frame. In addition, there could be another problem if a particle is lost due to occlusion or maybe due to a failure in the particle detection for one or two frames.

To overcome problems due to crossings and missing particles, the Particle Tracker plug-in utilizes a global optimization method. Instead of satisfying local situation such as connecting A to A or A to B, the global optimization algorithm in the Particle Tracker plug-in creates a table (or a matrix) of $i \times j$ dimensions. i corresponds to the number of particles detected in the nth frame, and j corresponds to the number of particles detected in the $(n+1)$th frame. Within this table, the algorithm evaluates all the possible pairing of particles in the nth frame and in the $(n+1)$th frame. There will be many combinations possible.

In addition, such a table could be extended to the second frame after the current frame so that some particle that might have temporally disappeared in the next frame could be linked to a particle in the frame after the next. Whether such stepping over is done is determined by the *Link Range* parameter in the interface.

For each of the possible combination of particle linking, an evaluation is done by first calculating a cost of linking certain particles: this calculation uses an equation called *cost function* (see below). Then the total cost of a combination is calculated by summing up the cost of each linking pair. For all the possible combinations, the total cost of each is calculated. By choosing a combination with the lowest total cost, the global optimization of particle linking is accomplished.

The cost function used in the Particle Tracker plug-in was designed so that the cost of particle pairing becomes dependent on three factors:

- Distance between the paired particles (so to say the displacement by linking the pair).
- Difference of total intensity (m_0) between the paired particles. This assumes that particle intensity should not change so much if they are the same single particle.
- Difference of the second-order intensity moments (m_2) between the paired particles. This assumes that there should be not much changes in the particle intensity distribution between frames.

The cost increases as the distance between particle pairs increases (larger displacement), the difference in the intensity increases, and the difference in the distribution of intensity increases. Hence, particles that are closer, with similar intensities and with similar intensity distributions, are linked as pairs and at the same time globally optimized.

Finally, here are the two parameters to be set in the particle linking part.

- *Link Range*: Defines how many successive frames will be included for searching a particle to be linked.
- *Displacement*: The maximum of expected displacement distance. Particles beyond the distance defined here will not be considered as linking target.

For more details on the algorithm of the Particle Tracker plug-in, see Refs. [5,6].

6.4.3 Tools

We use the Fiji distribution of ImageJ (http://fiji.sc) and two plug-ins.

- *Particle Tracking (MOSAIC_ToolSuite.jar)*: Developed by the MOSAIC group at ETH (they are now at MPI-CBG in Dresden). This plug-in contains the particle tracking plug-in along with other various tools. The tracking function of this plug-in is optimized for spherical dots, as this plug-in was developed for tracking spherical virus [5,6].
  Download link: http://mosaic.mpi-cbg.de/?q=downloads/imageJ.
- *Course Utility Plug-in (EMBL_sampleimages-1.0.0.jar)*: This is a sample image loader.
  Download link: http://cmci.embl.de/downloads/coursemodules.

There are four ways to install the plug-in. You could use any of these methods.

1) Drag and drop the downloaded plug-in file in the Fiji menu bar.
2) `[Plugins > Install . . . ]` and then select the downloaded file.
3) Directly copy the file to plug-ins folder under Fiji directory.[2)]
4) With recent Fiji versions (2014–), "Update Site"-based installation is available for both plug-ins. For the MOSAIC_ToolSuite, use the update site *MOSAIC ToolSuite*, and for the sample image loader, use the update site *CMCI-EMBL*. Addition of these sites could be achieved by "Manage Update Sites" interface accessible from the Fiji updater ([Help > Update Fiji]).

In all cases, restart Fiji to let it recognize the newly added plug-ins.

There are several other object tracking tools bundled with the Fiji distribution. We will not use these trackers in this textbook but it might be worth mentioning

2) In case of OSX, right click (or ctrl-click) Fiji.app and select "Show Package Contents". There, you will find the plug-in folder.

them here. All these tracking plug-ins could be found under `[Plugins > Tracking >]`.

- *Manual Tracking (Manual_Tracking.class)*: This is a plug-in bundled with Fiji that allows you to accumulate track data while you interactively click dots/particles using mouse. The author is Fabrice Cordelières. More information and download link could be found at http://rsbweb.nih.gov/ij/plugins/track/track.html.

  Manual tracking might sound low-tech, but when any of the available software does not auto-track your target, then manual tracking will be the ultimate solution to get coordinates extracted from your image data.
- *MTrackJ (MTrackJ_.jar)*: This is another manual tracking plug-in bundled in Fiji (http://www.imagescience.org/meijering/software/mtrackj/). The author is Erik Meijering. It has more options available to control the tracking conditions and post-editing the tracks you clicked such as track merging and track splitting. Track Color selection capability also suits to artistic scientist. Compared with the Manual Tracking plug-in, options are hidden behind facade GUI, so it might take a bit more time to learn handling options than using the Manual Tracking plug-in.
- *TrackMate*: TrackMate (http://fiji.sc/TrackMate) does automatic tracking of multiple objects. Authors are Nick Perry, Jean-Yves Tinevez, and Johannes Schindelin. TrackMate offers up-to-date algorithms for solving particle linking problem by using linear assignment problem algorithm. The plug-in can also use other linking algorithms such as the nearest neighbor linking and area overlapping in successive frames.
- *Spot Tracker*: This plug-in was developed by Daniel Sage for tracking FISH signal (spotty signal) movement within yeast cell. As yeast cell is very small, noise is prevalent in fluorescence image sequences. The plug-in works well under such noisy image condition. This plug-in is not included in the Fiji distribution, but freely downloadable from the following link: http://bigwww.epfl.ch/sage/soft/spottracker/.

6.4.4 Workflow

1) Get image files.

   Open the image (`[EMBL > Samples > eb18_b.tif]`). This is a sequence taken from single cultured cell labeled with EB1.

   We analyze movement of EB1 signal using three strategies with following steps:

   a) Track signals automatically. When tracking was successful, we have numerical data of moving dots (*xy* coordinates over time).

   b) Using track coordinates, we calculate the direction of EB1 movement (hence MT orientation). See the following section for this calculation.

c) Plot the results in histogram in R. We try to do radial plotting. Evaluate the bias in microtubule orientation using circular statistics.
d) If time allows, do also the similar using *Drosophila* image sequences.

2) Open the file in Fiji.

```
[File > Open . . . ]
```

3) Examine the sequence using stack tools.
Start animation, Stop animation, Change frame rates, Manipulations, . . .
4) Set correct dimensions of the image.

```
[Image > Properties . . . ]
```

. . . Image stacks are by default taken as a *z*-series and not *t*-series. Set slices to 1, and frames to appropriate size (number of frames).
5) Start the Particle Tracker plug-in.
Start the Particle Tracker by

```
[Plugins > Mosaic > ParticleTracker 2D/3D].
```

6) Study dot detection parameters.
This tracking tool has two parts. First, all dots in each frame are detected, and then dots in successive frames are linked. The first task then is to determine three parameters for dot detection: Radius, Cutoff, and Percentile.
Try setting different numbers for these parameters and click "Preview Detected". Red circles appear in the image stack (Figure 6.3). You could change the frames using the slider below the button (scroll bar below the image does not work for interactive particle detection check).
After some trials, set parameters to what you think is the optimum.
7) Set linking parameters.
Now, two parameters for linking detected dots should be set.
a) *Link Range*
. . . could be more than 1, if you want to link dots that disappear and reappear. If not, set the value to 1.
b) *Displacement*
. . . expected maximum distance that dots could move from one frame to the next. Unit is in pixels.
After parameters are set, click "OK". Tracking starts.
8) Inspect the tracking results.
When tracking is done, a new window titled "Results" appears (Figure 6.4). At the bottom of the window, there are many buttons. Click "Visualize All Trajectories" and then a duplicate of the image stack overlaid with trajectories will appear (Figure 6.5a).

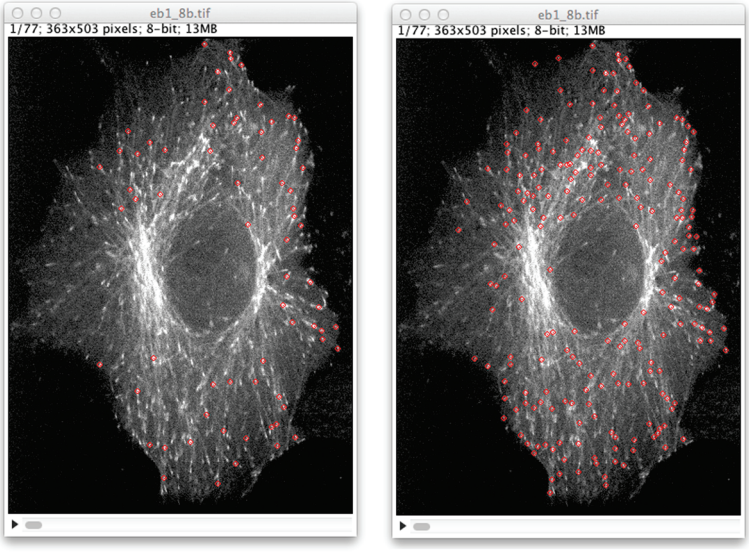


Figure 6.3 Particle Detection parameter search: (a) percentile = 0.1%; (b) percentile = 0.5%.

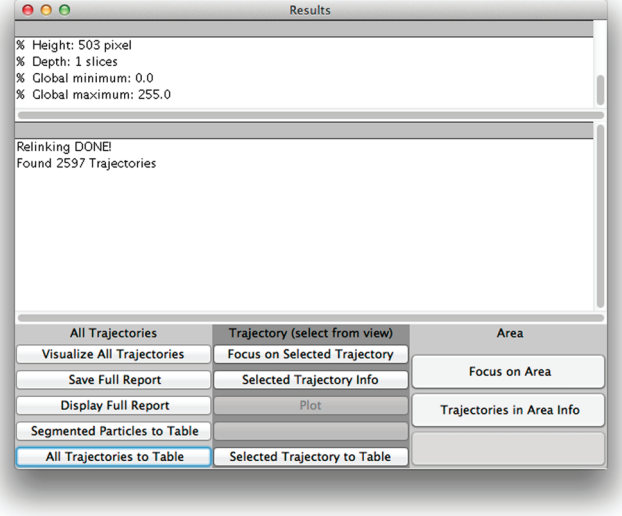


Figure 6.4 Particle Tracker results panel.

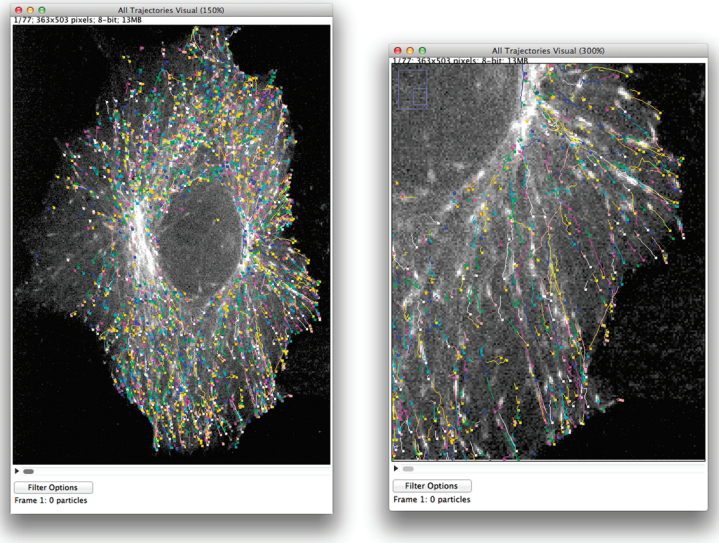


Figure 6.5 Visualization of tracks: (a) full view of the tracks; (b) bottom-right corner zoomed.

Select a region within the stack using rectangular ROI tool and then click "Focus on Area" (Figure 6.6). This will create another image stack, with only that region. Since this image is zoomed, you could carefully check whether the tracking was successful or not.

If you think the tracking was not successful, then you could first redo the linking. In the results panel, there is a menu `[Relink Particles]`. Using this menu command, linking parameters could be altered and redo the linking part.

If results are not satisfactory with re-linking trials, then you should reset all the parameters starting from the particle detection setting again.

9) Export the tracking results.

When you are satisfied with tracks, your data are now ready for further analysis.

To analyze the results in R, data should be saved as a file. To do so, first click "All Trajectories to Table". In the results table window, select `[File > Save As . . . ]` and save the file on your desktop. By default, the file type extension is ".xls", Excel format, but change this to ".csv". CSV stands for "comma separated values" and is a more general data format that you could easily import in many softwares including R.

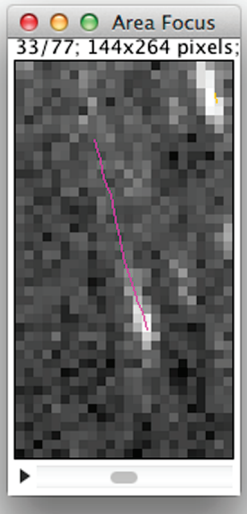


Figure 6.6 Track visualization: "Focus on Area".

6.5 Step 2: Directionality Analysis Using Matlab

With the tracking tool, we collected quite a huge amount of data from one single cell. We measured thousands of coordinates of EB1 spots and assigned them to each other, which resulted in a set of trajectories, each consisting of multiple *xy* coordinates. All this information is stored in a huge CSV table.

It is hard to find out something interesting about our measured cell by just studying the huge data table. In Matlab, we will perform data processing and try to convert the data to a more compact and "understandable" form.

6.5.1 Data Import

First, we have to read the CSV text file into a Matlab variable by using the function `readtable`:

```
data = readtable(datname);
```

where `datname` is a string with the path and file name of the CSV table.

After import, we have our data nicely organized in a `table`, where each field corresponds to one column in the CSV sheet. Here we will print columns 2–5 of the first two rows of the table:

```
>> data(1:2,2:5)

ans =

  Trajectory    Frame       x         y
  __________    _____    _______    _______

  1             0        207.41     7.215
  1             1        206.92     6.664
```

You can access the *x* column with `data.x`, as you would with a `struct`.

As you see in the code snippet, we also import the first frame of the raw images:

```
1 imageName = 'eb1_8b.tif';
2 im  = imread( fullfile(modulePath,imageName), 1);
```

code/module4AnalysisScript_v3_SN.m

Thus, we can display the image with `imshow` and plot the imported data on top (using `plot`). This is useful to quickly check whether our data import worked fine. Make sure that you plot dots rather than lines (which is the default) and don't forget to set the figure to `hold on`. Your result should look similar to Figure 6.7.

6.5.2 Calculating Microtubule Orientation

The orientation of microtubules is one potentially interesting feature that is represented in our tracking data, although somewhat hidden in a huge mess of numbers so far (microtubule orientation is directly related to the orientation of EB1 trajectories).

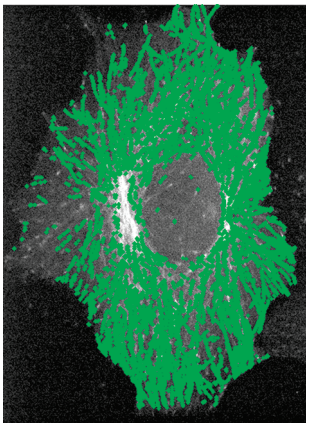


Figure 6.7 Quick import check: plot trajectory data on top of image.

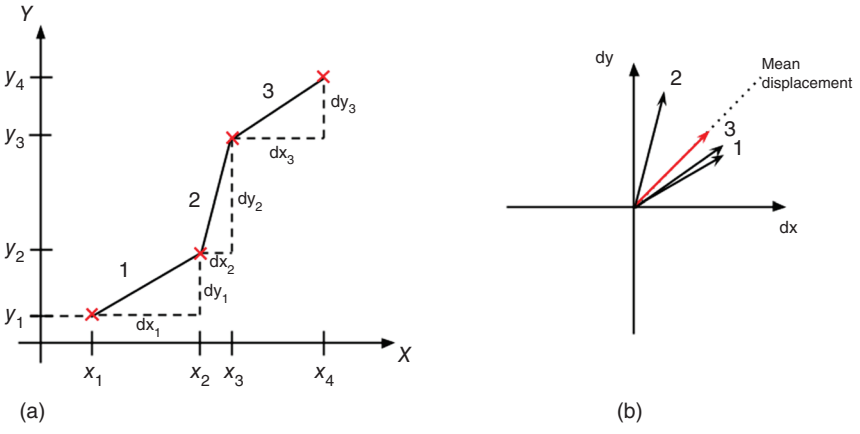


Figure 6.8 Calculating the mean displacement of a trajectory. (a) A trajectory consisting of four *xy* coordinates. The displacement from one coordinate to the next is denoted as dxy_1, dxy_2, and dxy_3. (b) The displacements (red: dxy_1, dxy_2, dxy_3) are averaged to calculate the mean displacement of the trajectory (black arrow).

In this section, we calculate the mean displacement (single per time step) of each trajectory and save the result in a vector called `meanDisp`. To understand how the mean displacement is calculated, take a look at Figure 6.8. The trajectory shown on the left consists of four *xy* coordinates. The displacement from one coordinate to the other is shown by dxy_1, dxy_2, and dxy_3 (dotted lines on the left). Mathematically, it is the difference between adjacent coordinates. On the right, dxy_1, dxy_2, and dxy_3 are drawn in red. The mean displacement (black arrow) is the average of the three displacement vectors. If you compare the mean displacement vectors with the trajectory on the left, you see that it shows the average orientation of the whole trajectory (Figure 6.8).

In fact, you might notice that the mean displacement is equal to the end-to-end vector divided by the number of steps.

$$\begin{aligned}\sum_{i=1}^{N-1}\frac{\vec{x}_{i+1}-\vec{x}_i}{N-1} &= \frac{(\vec{x}_2-\vec{x}_1)+(\vec{x}_3-\vec{x}_2)+\cdots+(\vec{x}_N-\vec{x}_{N-1})}{N-1}\\ &= \frac{(\cancel{\vec{x}_2}-\vec{x}_1)+(\cancel{\vec{x}_3}-\cancel{\vec{x}_2})+\cdots+(\vec{x}_N-\vec{x}_{\cancel{N}-1})}{N-1}\\ &= \frac{\vec{x}_N-\vec{x}_1}{N-1}\end{aligned} \tag{6.1}$$

```
b = diff(a)
```

a = | a_1 | a_2 | a_3 | a_4 |

b = | a_2-a_1 | a_3-a_2 | a_4-a_3 |

Figure 6.9 The diff function: If *a* is a vector, then diff(*a*) returns a vector *b*, one element shorter than *a*, of differences between adjacent elements.

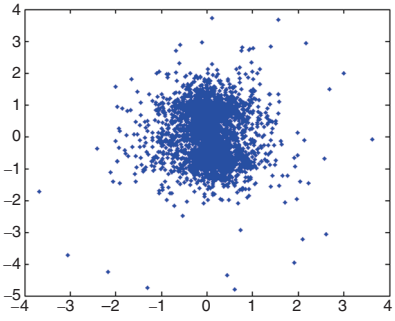


Figure 6.10 Dot plot of microtubule orientations. You already see some orientation preferences (upward and downward), but the plot looks quite messy.

Calculate the mean displacement in x and y directions for each trajectory and save the result in a two-column vector `meanDisp`, where each row represents one trajectory! Here are some hints for the actual implementation in Matlab:

- Find out how many trajectories you have in your data set and save it as the variable `nTraj`. Use the vector `data.Trajectory` for this.
- Loop through all trajectories. Inside the loop, make a subset of x and y coordinates that just contain data from one trajectory. You can do this by using Boolean indexing and the trajectory label number stored in `data.Trajectory`.
- Calculate the mean dx and dy (mean displacement) by using Eq. (6.1).

Plot the mean displacement for each trajectory as dots. This should look like Figure 6.10. From the dot plot, we cannot see a clear orientation tendency. Let's try something different. If you use the `compass` function instead of `plot`, you generate an arrow plot of mean displacements (Figure 6.11). This compares better with the scheme in Figure 6.8b. Still it is not a very clear representation of microtubule orientations.

6.5.3 Representation of Microtubule Orientations in Histograms

The plots shown above look that messy because of the large amount of data. We have to find some way to make the data more compact.

First, we calculate the orientation angle θ (theta) from our Cartesian coordinates. This can be easily done with the function `cart2pol`. See Figure 6.12 for further explanation.

After using `cart2pol` we should have an array `theta`, where each element represents the orientation angle of one trajectory.

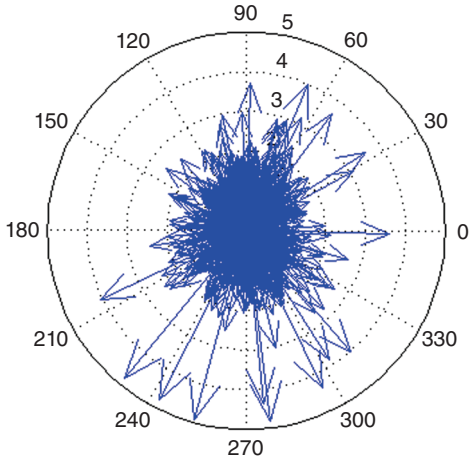
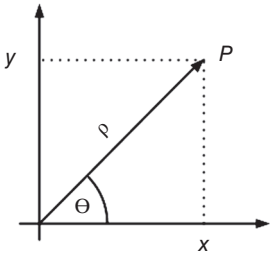


Figure 6.11 Arrow plot of microtubule orientations. Not much better than the dot plot in Figure 6.10. Still a mess!




```
[Θ ρ] = cart2pol(x, y)
```

Figure 6.12 cart2pol (Cartesian to polar) calculates the angle θ and the length ρ from an *xy* coordinate (or an array of *xy* coordinates).

We can visualize the distribution of the angles with a normal histogram (`hist` function) (Figure 6.13) or a circular histogram (`rose` function). Especially with the `rose` plot, we see a clear preference of microtubule orientations toward 80° and 270°.

6.5.4 Directionality Statistical Analysis

In this section, we will perform some statistical analysis on our qualitative assessment in Figure 6.14.

Our first step is to make sure the movement is indeed along a specified direction. To test whether direction is random (uniform) or not, we can use χ^2

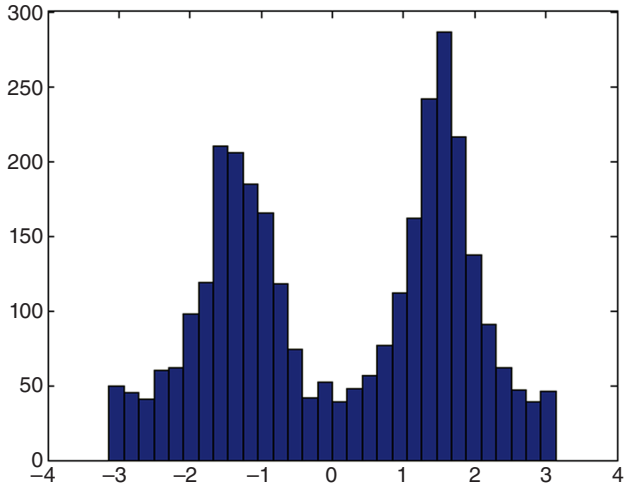


Figure 6.13 Histogram plot: histogram of microtubule orientation angles.

uniformity test (`chi2uniftest`). This will give a small p-value whenever the angle distribution seems to have at least a preferred direction.

We are going to use a custom function `chi2uniftest` for this purpose.

```
>> p = chi2uniftest(theta)

p =

    0
```

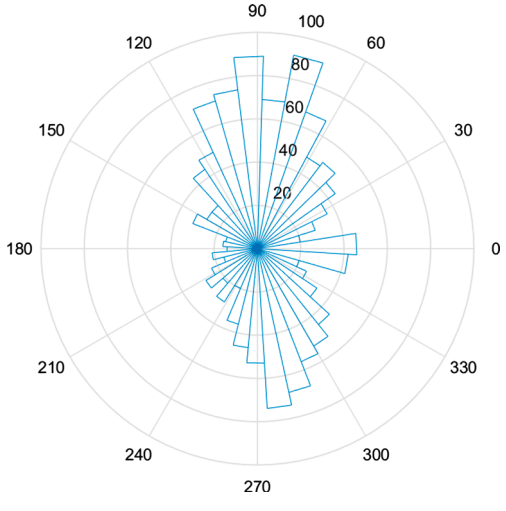


Figure 6.14 Rose plot: histogram of microtubule orientation angles in a circular representation.

The p-value returned by the test is indeed very small, smaller than the machine precision, which indicates a strong preferred direction. To determine the mean direction, we can use a modified version of the sample mean for circular data. This is performed by calculating the mean of the Cartesian coordinates instead (let us assume all the displacements are in an $N \times 2$ matrix `meanDisp`):[3)]

```
>> theta_mean = atan2(mean(meanDisp(:,2)),mean(meanDisp(:,1)));

theta_mean =

  -2.9761
```

We can also look for the principal directions using a fitting procedure and a custom Matlab function `findcircnmodes`. The function maps the orientations onto the unit circle and then clusters the points using a Gaussian mixture model.

The Matlab function is `findcircnmodes`. It requires to specify how many clusters you expect, two in our case.

```
>> pDirs = findcircnmodes(theta,2)

pDirs =

   -1.5292
    1.1031
```

We can convert it to degrees by

```
>> pDirs_deg = pDirs*180/pi

pDirs_deg =

  -87.6187
   63.2029
```

To estimate a confidence interval for the modes, we can use bootstrapping. This particular method of bootstrapping repeats the estimation on a set of N samples taken from the list of orientations. We specify a confidence level `alpha` of 5%.

```
>> alpha = 5;
>> BS = bootstrp(100,@(x) sort(findcircnmodes(x,2)),theta);
>> ci_low=quantile(BS,alpha/2/100)
>> ci_high=quantile(BS,1-alpha/2/100)
```

3) Why can we not calculate a "normal" average by using the mean function? Think about the following example: The average of two angles $\theta_1 = 359°$ and $\theta_2 = 3°$ is in fact 1°. However, a "normal" average calculation would give $(359° + 3°)/2 = 181°$, which is obviously wrong.

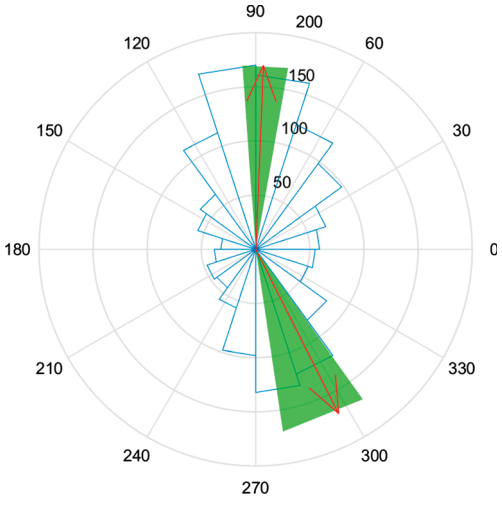


Figure 6.15 Rose diagram with the mean direction of movement plotted in red and the 95% confidence interval as a region in green.

We can plot the rose of orientations, together with the principal directions and the confidence intervals with the `plotmodesci` function (you can see the result in Figure 6.15):

```
>> figure; plotmodesci(theta,pDirs,ci_low,ci_high);
```

In summary, the principal directions of movement from our experiments are $\theta_1 = -63.2°$ and $\theta_2 = 87.6°$.

6.5.5 Summary of Tools Used

6.5.5.1 Matlab

imread: import image data;
readtable: function to import CSV data into a table;
imshow: display matrix as image;
plot: plot data points;
compass: plot arrows emanating from origin;
rose: angle histogram plot;
hist: histogram plot;
diff: calculate difference between adjacent elements of a vector;

cart2pol: transform Cartesian coordinates to polar or cylindrical coordinates;

bootstrp: perform bootstrapping;

chi2uniftest: (custom) perform a χ^2 uniformity test;

findcircnmodes: (custom) calculate the principal directions using a Gaussian mixture model on the unit circle;

plotmodesci: (custom) plot the angle histogram, the principal directions, and the confidence intervals.

6.6 Step 3: Directionality Analysis Using R

R is an open-source statistical analysis tool widely used in the scientific community. Many packages are available as additional modules. There are several interfaces available for R, and we use the RStudio in this practical course. Details on circular statistics could be found in Ref. [4].

6.6.1 Tools

The RStudio is an IDE (integrated development environment) that provides GUI access to R. This software can be downloaded from http://www.rstudio.com/ide/.

6.6.1.1 RStudio: Keeping Your Work as a Project

On start-up, RStudio has three panes: console, workspace, and files (Figure 6.16). It will be convenient for you to handle your work as an R project. From the menu, choose `[File > New Project ...]` and in the dialog select a location in your file system where to save your project. A project has its own directory and all related data and histories will be saved there as an RProject.

R itself already has such a system, which is called "workspace." Data will be stored in an RData file. RProject is an enhanced version of this concept. You could load the project later by double clicking and then all the work you have done in the previous session for that project will revive as it is.

6.6.1.2 A Very Short Introduction to R

We first start out with the very basic usage of R command line.

To set a variable a to 3,

```
a <- 3
```

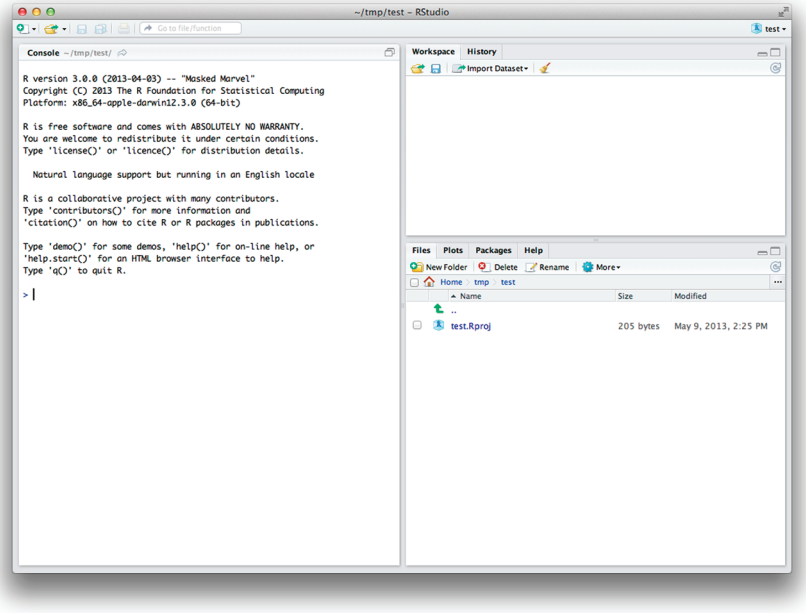


Figure 6.16 RStudio IDE. Console is on the left-hand side, where you input R commands.

Similarly, to set a variable *b* to 5,

```
b <- 5
```

To see the content of variable, simply type

```
a
```

Then you will see a printout in the console:

```
[1] 3
```

To calculate the sum, type

```
a + b
```

Then the printout will be

```
[1] 8
```

To store the result of calculation in *c*,

```
c <- a - b
```

Check the results

```
c
```

prints out

```
[1] -2
```

Till here, we made only one value associated with a variable, but a variable could contain multiple numbers (we call it a "vector"). First try the following:

```
1:5
```

This will print out

```
[1]  1  2  3  4  5
```

Similarly,

```
2:10
```

then the output will be

```
[1]    2    3    4    5    6    7    8    9    10
```

Such numerical sequence could be stored in a variable

```
d <- 1:10
```

Check the content of *d*, a vector.

```
d
```

Output will be

```
[1]    1    2    3    4    5    6    7    8    9    10
```

We know that the length of the vector *d* is 10 already, but you often encounter scenarios where you do not know the length of a vector. To know the length of a vector:

```
length(d)
```

"length()" is a command that inspects the vector and returns its length, so the output will be

```
[1] 10
```

Similarly,

```
length(a)
```

prints out

```
[1] 1
```

We could generate a vector in a bit more complex way by using command "seq(start number, end number, increment)".

```
da <- seq(1, 10, 0.7)
```

```
da
```

will print out

```
[1] 1.0 1.7 2.4 3.1 3.8 4.5 5.2 5.9 6.6 7.3 8.0 8.7
[13] 9.4
```

6.6.1.3 Installation of External Tools

The base package of R already has many useful functions, but when you have a complex and specific task, you could either write your own script for that task or search for a package that does the job. In many cases, searching an appropriate package is more efficient than writing your own. We use following packages:

1) fisheyeR;
2) CircStats;
3) circular;
4) plotrix;
5) movMF.

- Conversion of Cartesian coordinates to polar coordinates.
  - R package: fisheyeR.
  - http://finzi.psych.upenn.edu/R/library/fisheyeR/html/toCartesian.html.
  - We use functions `toPolar(x, y)` and `toCartesian(`θ`, r)`.
  - For explanation see http://en.wikipedia.org/wiki/Polar_coordinate_system.
- von Mises likelihood estimates.
  - http://en.wikipedia.org/wiki/Von_Mises_distribution.
  - R package: CircStats.
  - http://finzi.psych.upenn.edu/R/library/CircStats/html/vm.ml.html.
  - We use function `vm.ml( . . . )`. μ (mean) and κ (concentration parameter) values could be calculated using this function.
  - R package: circular (more parameters).
  - http://finzi.psych.upenn.edu/R/library/circular/html/mle.vonmises.html.
  - We use function `mle.vonmises( . . . )`. Maximum likelihood estimate of concentration parameter (κ) is a good indicator of bias in directionality.
- Mixture of von Mises–Fisher distribution, likelihood estimates.
  - R package: movMF.
  - http://cran.r-project.org/web/packages/movMF/index.html.
  - We use this package also for simulating multimodal distributions.
- Data plotting.
  - R package: CircStats.
  - http://finzi.psych.upenn.edu/R/library/CircStats/html/00Index.html.
  - R package: plotrix.
  - Use function `radial.plot`.
  - http://www.oga-lab.net/RGM2/func.php?rd_id=plotrix:radial.plot.

In RStudio, you could see available packages in the "Packages" tab in the bottom-right panel. All the packages listed there could be simply loaded from your local R distribution. Many of packages that are not listed there should be downloaded from Internet and installed. Downloading and installing actually is not difficult, since there is a menu command for that purpose.

`[Tools > Install Packages]` Type in the package name that you want to install in the second field, and simply clicking "Install" will do all the job for you.

After installation is finished, check the "Packages" tab again and click the check box for the installed package. If the box is checked, you could use functions offered by the package.

In the following, command line interface in the bottom-left panel will be used. Command input field starts with a prompt ">", but will be omitted in this textbook except where a command and its output are shown.

6.6.2 Workflow

6.6.2.1 Basic Analysis

1) Loading tracking data.
   Load data by the following command:

```
ptdata <- read.csv("C:/course/PTresults.csv")
```

The argument within the parenthesis is just an example path, and it should be replaced according to where you have saved the track CSV file.

If you happen to have exported the data in ImageJ without changing the file extension, then the file ending should be ".xls" and in that case you have your data not in comma-separated, but in tab-delimited file. To import such data, you need an added option.

```
ptdata <- read.csv("C:/course/P", sep="\t")
```

The second argument explicitly states that the values are separated by tab ().

If the import is successful, you will find data `ptdata` being listed in the "Workspace" tab in the top-right column. Single clicking that data name in the list will open a table in the top-left panel showing the content of that data variable.

More traditional way of checking data content is from command line.

```
head(ptdata)
```

Function `head()` command will print several values in the beginning of data.

Important note on the data structure: In the output of the Particle Tracker plug-in, x and y coordinates are inverted (values in the "x" column are actually y values and the values in the "y" column are x values). According to the author of the plug-in, they cannot change this for maintaining the consistency with the Matlab script they have. We will see bias in the radial plot but keep in mind that it is 90° rotated.

2) Accessing data in 2D table.
   If data are in a table format (2D vector, such as the case with our data "ptdata"), then an element in the table could be specified by a form "data [row index, column index]". Both row and column numbers start from 1, so if you want to get a number at the top-left corner of the table we have just imported, a way to specify that cell is

```
ptdata[1, 1]
```

   and one column to the right would be

```
ptdata[1, 2]
```

   To specify a column, not only a single cell,

```
ptdata[, columnnumber]
```

   Where a row number should be specified is now a blank. It means that all numbers apply there, which in turn means all the rows available. Another way to specify a column by name of the column header is also available. If we want to specify a column "Trajectory" in the table,

```
ptdata$Trajectory
```

   This means "Column Trajectory in data ptdata". By "dataname + dollar sign ($) + the column header", you could specify a column vector within the table.
3) Extracting vectors out of data frame.
   Our aim is to extract x and y coordinates, and calculate the angle of the movement from one time point to the other. For this, we first extract x and y vectors out of the data. Don't be afraid with the name "vector." It simply is a term that means an "array of numbers." We first try to extract x and y coordinates of a specific trajectory. In the case below, Trajectory No. 2 will be extracted.

```
t1x <- ptdata[ptdata$Trajectory==2, 4]
t1y <- ptdata[ptdata$Trajectory==2, 5]
```

   Two lines of commands are executed individually as you press "return" key. Both lines are doing similar job each for x and y. Translation of these lines into normal words would be something like this:

```
From ptdata, copy column index 4 to t1x
for only those rows which have values equal to 2
in the column Trajectory.
```

We specified a range of rows by `ptdata$trajectory==2`. To see what this is doing, try

```
ptdata$Trajectory==2
```

You will then see a sequence of statements with "True" and "False". This tests for all the values in the Trajectory column, and returns "Yes (True)" if the value is 2. This could also be written by column index rather than column header as the following:

```
ptdata[, 2]==2
```

The returned values will be the same.
You could check whether vectors were extracted successfully by clicking corresponding variable name (`t1x` or `t1y`) in the "Workspace" tab. Alternatively (which actually is the traditional way in R), you could type

```
t1x
```

in the command line. Content of the vector will be printed out. If the length of data is too long, then you could always use head() function to truncate the printout.

```
head (t1x)
```

4) Plotting tracks.
   To visualize tracks, you could simply plot y versus x coordinates. The easiest approach is

```
plot(t1x, t1y)
```

. . . but remember, X and Y are switched in the plug-in, so we swap the vectors. In addition, we do some cosmetics: add axis labels and set the aspect ratio of the plot to 1.

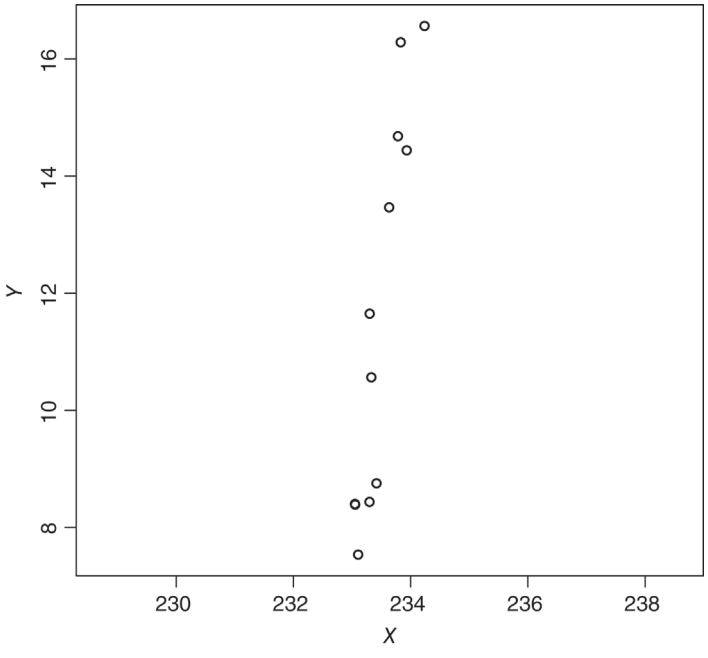


Figure 6.17 Track ID2 plotted on the *XY* plane. Your track might look different, as the tracks would be different depending on how you tracked the particles.

```
plot ( t1y, t1x, xlab ='X', ylab='Y', asp=1)
```

The plot should appear within one of the panels in the RStudio (Figure 6.17). We then try plotting all the tracks. This is relatively easy because we could use all x and y data.

```
tx <- ptdata[ , 4]
ty <- ptdata[ , 5]
plot( ty, tx, xlab ='X', ylab='Y', asp=1)
```

It looks OK, but it is difficult to know which track belongs to which, since all tracks are in black. To color each track differently, we should prepare a vector with different colors. For this, we need to first know the number of trajectories we have in our data.

```
max(ptdata[,2])
```

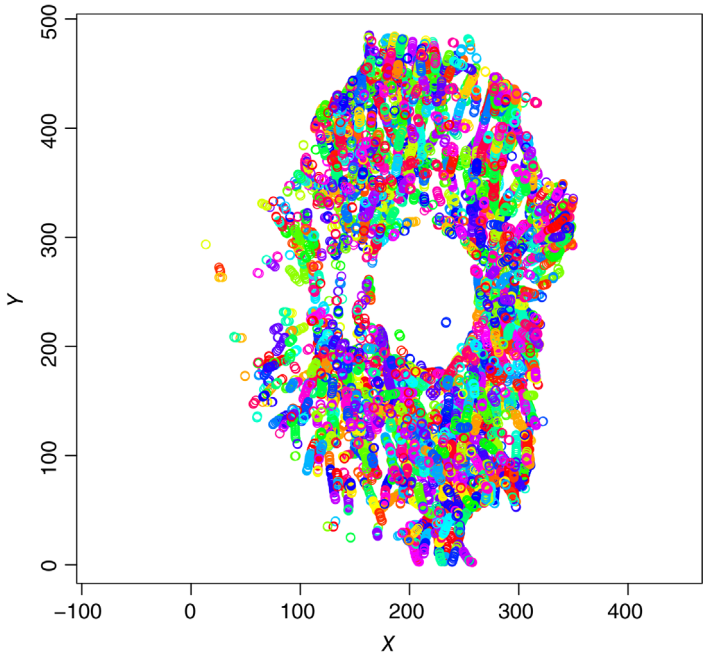


Figure 6.18 All tracks plotted on the *XY* plane.

Since the second column in *ptdata* is the trajectory id, max() function will return the last id (which should be the largest). We then prepare the number of colors using

```
tc <- rainbow(max(ptdata[, 2]))
```

We finally upgrade the plot by

```
plot(ty, tx, col=tc[ptdata[,2]], xlab ='X', ylab='Y', asp=1)
```

The plot should look like Figure 6.18.

5) Putting commands into a script.

If you were successful in plotting tracks, then you could write a script that does all these in one shot. Useful tool for doing this is the "History" tab in the top-right panel.

First, create a new script file using the menu command.

```
[File > New > R Script]
```

You will then see a blank script in a tab in the top-left panel. You could write all the commands by manually typing, but it will be more efficient and there would be less errors if you use the "History", which shows all the command history. At the top of the "History", there are several buttons available. One of them is "To Source", and this could be used to transfer lines you select to the script in the left-hand side.

Scroll back to the position where you did the data importing using read.csv.

```
ptdata <- read.csv("Z:/course/PTresults.csv")
```

Select the line, and then click "To Source". Check the script. You will see that the line is now in the script.

To test using the script, let's remove the data using command line

```
remove(ptdata)
```

This command removes the ptdata from the workspace. Check the workspace and confirm that the data `ptdata` are not there anymore. Now, go back to the script, and select the line that was transferred from the history. Then click "Run" button in the top bar of the script tab. This will execute the selected command in the script file. Check that the data `ptdata` are now in the workspace again. We tested "To Source" and "Run" buttons with a single line, but we could also test "To Source" and "Run" buttons with multiple lines.

Go back to the history, cherry pick the commands that are necessary, and transfer them into the source file. Be careful about the order of commands.

After reconstructing the sequence of commands, you could save the script as a file. File name is automatically appended with extension ".R". After the file is saved, you could execute all lines in the script by "Source" button in the "Script" tab.

Below is an example script that simply plots the tracks.

```
1 ptdata <- read.csv("./Results.csv")
2 tx <- ptdata[, 4] % this could also be ptdata$x
3 ty <- ptdata[, 5]
4 tc <- rainbow(max(ptdata[, 2]))
5 plot(ty, tx, col=tc, xlab ='X', ylab='Y', asp=1)
```

code/trackplotting.R

6) Getting movement vectors for each time point.

We now try to get movement vectors. We need to calculate vectors that are made between the position of dot at frame t and the position in the next

frame $t + 1$. To do so, we extract two vectors, each with one element less than the original coordinate vector and one element shifted in one of the two vectors. This could be done as follows:

```
d1x <- t1x[2:length(t1x)] - t1x[1:(length(t1x)-1)]
d1y <- t1y[2:length(t1y)] - t1y[1:(length(t1y)-1)]
```

Colons in the square brackets are for generating numerical sequence as we did in the short introduction.

In the right-hand side assignment, we use t1x vector two times, but calling different ranges. In the first term, the range starts from second element of t1x vector and extends until the end. The second term calls from the beginning of the vector and extends until the second last element.

. . . the same thing could be done in much simpler way by using `diff()` function.

```
d1x <- diff(t1x)
d1y <- diff(t1y)
```

Check that `d1x` is the difference generated from `t1x`.

7) Getting the direction of movement vectors.

We now want to get the direction of each movement vector. This could be done by using the function offered by the package fisheyeR, which you loaded in the beginning of this protocol. This function converts Cartesian coordinates to polar coordinates.

```
tpol1 <- toPolar(d1x, d1y)
```

The returned value `tpol1` has twice the length of `d1x` or `d1y`. This is because theta values are listed in the first half and magnitudes are listed in the second half. Our interest is in the theta values, the angle or direction of the movement vector. Check the values by

```
tpol1
```

The output should have two types of indices, those with a suffix "t" and the others with a suffix "r". As mentioned earlier, "t" data are angles (theta). These values should be in a range between $-\pi$ and $+\pi$. Rest of the data with "r" are the length of vectors.

To separate vector angles and lengths, we could convert the `tpol1` vector to a matrix.

```
matrix(tpol1, ncol = 2)[,1] % theta
matrix(tpol1, ncol = 2)[,2] % length
```

8) Converting direction data to vectors on unit circle.

For later purpose, we try to convert our data to vectors on unit circle. These vectors should all have a length of 1, so we first prepare such unit length magnitude data using `rep`. Then convert the data back to the Cartesian coordinates.

```
t1theta <- matrix(tpol1, ncol = 2)[ ,  1]
unitlength <- rep(1, length(t1theta))
t1unit <- toCartesian(t1theta, unitlength)
t1unitx <-  matrix( t1unit,  ncol = 2)[ ,  1]
t1unity <-  matrix( t1unit,  ncol = 2)[ ,  2]
```

Plot these vectors on the *XY* plane,

```
plot(t1unitx, t1unity, xlim=c(-1, 1), ylim=c(-1, 1), asp=1)
```

and just to have a guideline, we draw a circle using the package plotrix function `draw.circle`.

```
draw.circle(0,0,1, border="red")
```

See Figure 6.19.

9) Calculate directions of all movement vectors.

Till here we tried calculating direction of movement vectors only in a single trajectory. What we want actually is to calculate all the directions of movement occurring in the image sequence to examine whether there is any bias in the movement direction. To do so, we could use function `diff()`, but there is one problem. Since all data are in a single table and trajectories are in same columns, `diff()` will calculate the movement vector that is made between the last coordinates of trajectory n and the coordinates at the first time point of trajectory $n + 1$. To avoid this, we prepare `diff()` of trajectory id, and use that vector as a flag for elimination of diff data from the `diff()` of x and y coordinates. We first take an example with short vectors.

Prepare a sample sequence of x coordinates and get diff.

```
aa <-c(1:10, 101:110)
daa <-diff(aa)
```

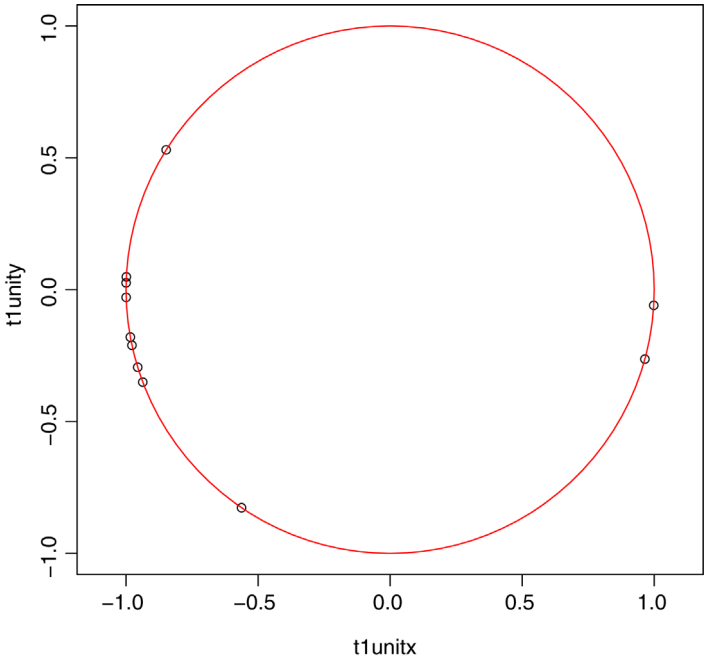


Figure 6.19 t1 track data converted to unit vectors. Red circle is the unit circle, with small circles showing data points.

The function `c()` constructs a vector by combining arguments, so typing aa should output

```
[1]   1   2   3   4   5   6   7   8   9   10   101
102  103  104  105  106  107  108  109
[20] 110
```

Then prepare a sample sequence of trajectory id.

```
bb <- c(rep(c(1),   10),    rep(c(2), 10))
dbb <- diff(bb)
```

Function `rep()` will repeat the numerical sequence in the first argument up to the number given in the second argument. So typing bb should return

```
[1]  1  1  1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2  2  2
```

diff of this vector then should be

```
[1]  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0
```

As such, we obtain a sequence dbb that flags where data should be removed from diff of aa, daa. We then pass a condition for the daa in the following way:

```
daac <- daa[dbb == 0]
```

We apply such data processing to our real data.

```
dtx <- diff(ptdata$x)
dty <- diff(ptdata$y)
dtraj <- diff(ptdata$Trajectory )
dtxc <- dtx[dtraj == 0]
dtyc <- dty[dtraj == 0]
pol = toPolar(dtxc, dtyc)
```

`ptdata$x` will return the column with the header "*x*" from the table in `ptdata`. In the same way, we could isolate *y* coordinates as a vector in the second line. Other lines are same procedure as we tested with the examples. We now have a vector *pol*, which stores polar coordinates of movement vectors.

Since pol is double the length of *x* or *y* coordinate columns, we separate angles and magnitudes.

```
angledata <- matrix( pol, ncol=2)[,  1]
magdata <- matrix( pol, ncol=2)[ ,  2]
outdata <- data.frame(angledata)
```

We now have direction data in *angledata*. We also now have magnitudes of movement vectors, which are the displacement per frame (velocity). The last line stores angledata in a data frame that will be useful to visually look at the data and is optional.

To analyze data, one best way to visualize the bias is plotting a histogram (Figure 6.20).

```
#plotting general histogram
bins <- seq(-pi, pi, pi/50)
hh <- hist(angledata, breaks=bins)
```

We first prepare a vector that defines breaks for the histogram bins. This is done by the function `seq(start, end, increment)`.

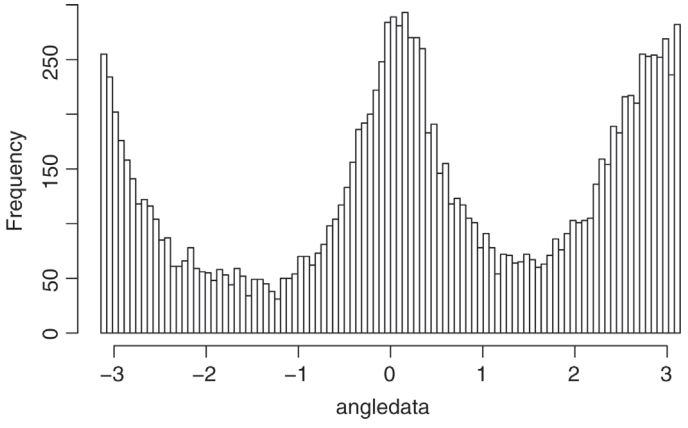


Figure 6.20 Histogram of angles.

Then the function `hist(data, breaks=bins)` is used to get the histogram values. You will then see a plot in the plot tab in the bottom-right panel. To have more information in the plot, or to adjust plotting parameters, see help by

```
help(hist)
```

To further improve the plot, we could see the distribution by using radial plotting function. We first need to load a package required for radial plotting.

```
#plotting   radial   plots
library(plotrix)
```

There are many functions in various packages that allow polar plots (or "radial plot" or "circular plot"). For a small number of sampling, any of them works well but radial plot function in `plotrix` seems to be better for data with a large sampling number. We first extract histogram counts from `hh`

```
# prepare   index
hhcounts <- hh$counts
```

Then prepare index for plotting in the radial plot. Unlike histogram, where breaks define the starting point of a bin and the bin extends until the next break, we have single point for representing data. For this, we need to adjust the position by half the width of bin.

```
radpos <- hh$mids
labpos = seq(-pi, 3/4*pi, by=pi/4)
radlabels <- as.character(format(labpos, digits=2))
```

`hh$mids` is a vector with values at the midpoints of bin breaks. Note that if we calculate the same values from break positions, it would be something like (you don't need to do this)

```
radpos <- hh$breaks[1:length(hh$breaks)-1]
+ diff(hh$breaks)/2
```

See help of "hist" for more information on "mids". Function `as.charactor()` converts numbers that are formatted by `format()` for limiting digits to character. Using these vectors prepared for plotting, we finally do the radial.plot command.

```
radial.plot(hh$counts,radpos,
            rp.type="p",
            main="EB1 directionality",
            line.col="blue",
            labels=radlabels,
            label.pos=labpos,
            radial.lim=c(0, 400),
            mar=c(2, 2, 6, 2))
```

Note that for a command with many options, one could separate them in multiple lines as long as the outer parenthesis is not closed. Maximum value must be adjusted so that the plots fit inside.

Tip: When you start not seeing any plots appearing even though you execute some commands for plotting them, try resetting the graphics by

```
dev.off()
```

6.6.3 Circular Statistics

We now have many vectors to compute basic statistical values about microtubule orientation.

6.6.3.1 Descriptive Statistics

The descriptive statistics of circular data could be easily calculated by using the package *CircStats*. Mean direction $\overline{\theta}$ and circular dispersion V are calculated. For mean direction,

```
circ.mean(angledata)
```

will print out $\overline{\theta}$. Note that we are dealing with circular data, and the mean direction might not mean anything if the data are distributed uniformly. To know how data are dispersed, use the following command:

```
circ.disp(angledata)
```

This will print several outputs:

```
     n         r          rbar           var
1 3340   459.527    0.1375829      0.862417
```

The value `var` is the circular dispersion of the data that ranges between 0 and 1. If closer to 0, then it means that data are concentrated in the mean direction, and if closer to 1, data are dispersed to a large degree. If the dispersion is large, you could probably understand that the mean of the angle data does not represent the results. We should first test the uniformity of the data.

Details: In the above output of the command `circ.disp(angledata)`,

- n is the number of data;
- R is the length of the sum of all vectors, or specifically called the *resultant length*. It should lie in the range (0, n). $R = n$ happens only when all vectors are aligned in a same angle. Note that $R = 0$ does not mean that vectors are directed randomly. As an example, a radially symmetric bidirectional distribution results in $R = 0$.
- $rbar = \overline{R} = R/n$ lies in the range (0, 1).
- var $= V = 1 - \overline{R}$, a value defined as the *sample circular variance*. A larger V indicates that directions are dispersed.

Descriptive statistics characterize our data, but their results do not tell us whether there is any bias in the distribution of directions. To test the presence of bias, uniformity tests are used.

6.6.3.2 Uniformity Test: Kuiper's Test

To test the uniformity (randomness) of direction, *Kuiper's test* could be used. This method is based on the order statistics. Deviation of measured values from the ideally uniform distribution is evaluated. It resembles the Kolmogorov–

Smirnov test as the Kuiper's test also uses the maximum and the minimum of deviations [4].

```
kuiper(angledata, alpha=0.05)
```

We used a significance level of 0.05, and output could be

```
      Kuiper's Test of Uniformity

Test Statistic: 6.9836
Level 0.05 Critical Value: 1.747
Reject Null Hypothesis
```

and this example tells you that the direction is nonuniform.

6.6.3.3 Uniformity Test: Rao's Spacing Test

Another test available in the package is the *Rao's spacing test for the uniformity of data*. Following is a quote from the website of Rao (http://www.pstat.ucsb.edu/faculty/jammalam/html/favorite/test.htm):

> Rao's Spacing Test is based on the idea that if the underlying distribution is uniform, successive observations should be approximately evenly spaced, about 360/N apart. Large deviations from this distribution, resulting from unusually large spaces or unusually short spaces between observations, are evidence for directionality.

When using this test in R, the command is as follows:

```
rao.spacing(angledata, alpha=0.05)
```

then the output could be

```
Rao's Spacing Test of Uniformity

Test Statistic = 153.6403
Level 0.05 critical value = 136.94
Reject null hypothesis of uniformity
```

So if you take a significance level of 0.05, null hypothesis (data are uniformly distributed around circle) is rejected.

6.6.4 von Mises Distribution

If the data are determined to be nonuniform, we expect that microtubules have some preferred direction. Now, if you could visually recognize that the bias seems to be unidirectional, then we could try to quantify this "unimodality" using the *von Mises distribution*.[4] This distribution is similar to the Gaussian distribution but with circular data. Mean value μ and concentration parameter, or degree of bias in the mean direction, κ could be calculated by the following command:

```
vm.ml(angledata)$mu
vm.ml(angledata)$kappa
```

The concentration parameter κ is just like inverse of the standard deviation of the Gaussian distribution. Larger the value, the distribution is more concentrated around the mean μ.

6.6.5 Bidirectional Distribution

The function `vm.ml` is a unimodal distribution: But as we observed already in the radial plot (Figure 6.21), the polarity of microtubules is roughly bidirectional and separated by 180° (e.g., if the movement is biased toward both 0 and 3.14 rad, we call such cases "axial"). In such a case, the mean direction estimated by `vm.ml` should be wrong and located between the two modes of preferred directions. The concentration parameter κ is also invalid since it assumes that the bias is unidirectional.

Therefore, we are now facing the following questions:

- How many preferred directions do the data have?
- For those preferred directions, what are the mean directions of each?

Specifically for axially bidirectional cases, a quick and easy way is to multiply the angle data by 2 [7]. This operation will convert the bidirectional distribution to unidirectional, so the analysis of the concentration parameter κ becomes valid. We will work on a more general approach in the next section, but we will explain this classic method by doubling the angle data.

```
angledata2 <- angledata * 2
```

4) In a strict sense, we need to test the unimodality but as we do not even know how many preferred directions are there, we just mention how to estimate mean direction using single von Mises distribution here and will discuss on the treatment of data with unknown number of preferred direction in the later section.

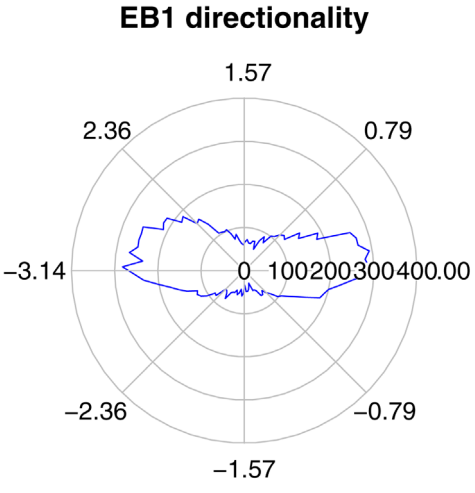


Figure 6.21 Radial plot of movement directions.

Since the doubled angles exceed the range $\pm\pi$, we could recalculate them to fit them within the range of 2π. This is not required for estimating μ and κ, but is better for plotting them.

```
angledata2 <- angledata2 + pi
angledata2 <- angledata2 %% (2*pi)
```

%% is called "modulo," an operator that returns a remainder of division. With this conversion, angles are now distributed between 0 and 2π.

Confirm that the distribution is visually unimodal by plotting this in a histogram. Then assuming that the distribution follows the von Mises distribution, calculate μ and κ using the function `vm.ml`.

```
> vm.ml(angledata2)
        mu        kappa
1 3.015434     0.791312
```

Confidence limits of μ with a 95% confidence level can be calculated using the function `vm.bootstrap.ci` as follows:[5)]

```
angleci <- vm.bootstrap.ci(angledata2, alpha = 0.05)
```

5) `vm.bootstrap.ci` only works for data in a range from 0 to 2π.

This takes a while for computation. Output would look like

```
Confidence Level:    95 %
Mean Direction:              Low = 2.99      High = 3.05
Concentration Parameter:     Low = 0.76      High = 0.82
```

Results could be placed in the plot using the `text` command.

```
text(-100, 80, paste("myu: ",
  as.character(vm.ml(angledata)$mu)))
text(-100, 70, paste("kappa: ",
  as.character(vm.ml(angledata)$kappa)))
```

Function `paste(,)` concatenates characters.

6.6.6 Multimodal von Mises Distributions

With bidirectional but not axial distributions or with even more directions, we need to consider the mixture model of the von Mises distribution.

6.6.6.1 The Mixture Model of von Mises–Fisher Distribution

For using this technique, we use the package movMF.[6] Install the package if you have not done so and then import the package.

This package allows you to deal with the mixture model of von Mises–Fisher (vMF) distribution. The vMF distribution is a generalized form of von Mises distribution and models multimodality of data clustering in n-dimensional sphere. The von Mises distribution is a special case of the vMF distribution when $n = 2$, a circle. The expectation-maximization (EM) algorithm is used for the model estimation.

The first thing we do is to generate sample data that we can work on. From here, we work on vectors on unit circle. Each vector represents a movement in certain direction. We use a command `rmovMF` to generate simulated data. Before using this command, we have to prepare some conditions. We create two directions as vectors that will be preferred directions μ, or centers of the two distributions that we will generate.

We say that these two directions are `c(−0.251, −0.968)` and `c(0.399, 0.917)` and from these two vectors we create a matrix.

6) A quite detailed explanation on this package could be found at http://cran.r-project.org/web/packages/movMF/.

```
>  mu  <- rbind(c(-0.251, -0.968),
+               c(0.399, 0.917))
> mu
       [,1]     [,2]
[1,] -0.251   -0.968
[2,]  0.399    0.917
```

`rbind` is a command that binds vectors passed to the argument by rows. We then set the concentration parameter for each of these two directions:

```
kappa <- c(4, 4)
```

Here, we set that $\kappa = 4$ in both cases, with which now we give the magnitudes to the each of two directions in *mu*, so that

```
>  muv  <- kappa * mu
>  muv
        [,1]      [,2]
[1,]  -1.004    -3.872
[2,]   1.596     3.668
```

Then we set the mixture probabilities for two distributions to determine the relative sizes of two distributions.

```
alpha <- c(0.48, 0.52)
```

The ratio of the population that belongs to each distribution is defined. In the above case, 48% of all data belongs to the first distribution and the rest 52% belongs to the second. Intuitively, mixture probabilities. Intuitively, you can consider these percentages as the mixture probabilities.

Finally, we set seed for the random number generator and then generate the modeled data using the command `rmovMF`.

```
> set.seed(123)
> x <- rmovMF(1000, muv, alpha)
> dim(x)
[1] 1000    2
```

`rmovMF` returns a two-column vector with its row number as we set in the argument (in the above case was 1000). Each row is the unit circle vector. We can plot them as we did already in Figure 6.19.

```
plot(x[,1], x[,2],    xlim=c(-1, 1), ylim=c(-1, 1), asp=1)
draw.circle(0,0,1,    border="red")
```

We now have data with the mixture of two von Mises distributions.

6.6.6.2 Determination of the Number of Distributions

From here we start behaving as if we did not know anything about the data.

We want to find out how many preferred directions are there in our data. For this, we use the command `movFM`. This command is a fitting function for mixed von Mises distributions and needs at least two arguments, the first argument is the data and the second argument is the number of expected distributions (Figure 6.22). We can add number of runs for the expectation-maximization (this is the fitting algorithm) as an optional value `nruns`. Let's first try with the unimodal hypothesis, expecting that there is only a single preferred direction.

```
> movMF(x, 1, run=10)
theta:
        [,1]            [,2]
1 0.1611095    0.001325801
alpha:
[1]  1
L:
[1]  6.458074
```

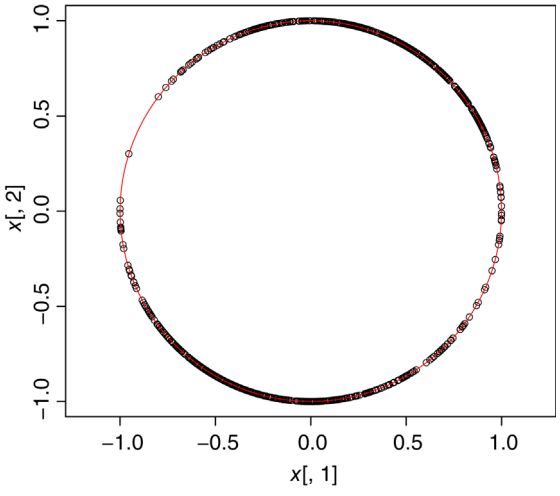


Figure 6.22 Mixture of two von Mises distributions.

The fitting went OK, and we have results. We expected single distribution in the data and the fitted distribution is centered around a vector indicated in the output as "theta", the value of which is `c(0.1611095, 0.001325801)`. "alpha" is the blending ratio of the mixture. As we fitted only a single distribution, alpha is 1. *L* is the log likelihood, a measure of how well the model fits to the data.

. . . then, could we take these fitted parameters and conclude that "we fitted the movement vectors with a single von Mises distribution and the preferred direction was this"? No. We need to compare with other possible numbers of distributions. Let's try fitting a mixture of two distributions.

```
> movMF(x, 2, run=10)
theta:
          [,1]          [,2]
1   -0.9066114    -3.886423
2    1.5488358     3.603321
alpha:
[1]  0.4842455    0.5157545
L:
[1] 355.1654
```

We now have two distributions, each centered around a vector shown in "theta" rows. The log likelihood shows that it is much larger than when a single distribution was fitted. We now fit three distributions:

```
> movMF(x, 3, run=10)
theta:
          [,1]          [,2]
1    0.6192978      5.608412
2    3.0261342      3.911007
3   -0.8911849     -3.827740
alpha:
[1]  0.2355730     0.2780516  0.4863754
L:
[1]  356.9048
```

Checking the log likelihood value, it is now even greater than when we fitted the two-distribution model. Comparing three different models, we might conclude that the three-distribution mixture model seems to be the plausible model for these data. But this is wrong. If you try with more distributions, log likelihood increases with number of distributions.

This is a typical behavior of models: more parameters you have, better the fit is. That does not mean that a model with more parameters is a better model, as we know (now going back to the fact) that this is a mixture of two distributions (Figure 6.23). So how do we compare?

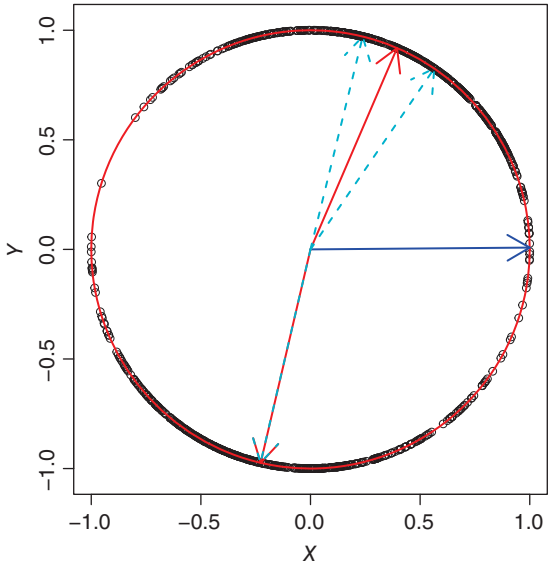


Figure 6.23 Results of fitting three different models. Arrows show the estimated theta values (the mean vectors of each distribution). Blue: one distribution; red: two distributions; cyan: three distributions.

We use a method known as *Bayesian information criterion* (BIC). This evaluation method takes the log likelihood into account, but also gives penalty by the number of parameters involved. Do the following:

```
d1 <- movMF(x, 1, run=10)
d2 <- movMF(x, 2, run=10)
d3 <- movMF(x, 3, run=10)
BIC(d1)
BIC(d2)
BIC(d3)
```

You should now see something like the following result in the console:

```
> BIC(d1)
[1] 0.8993632
> BIC(d2)
[1] -675.792
> BIC(d3)
[1] -657.4203
```

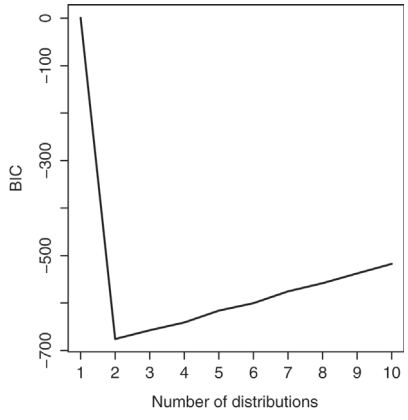


Figure 6.24 Evaluation of various number of distributions fitted to the data by BIC.

Lower the BIC, better the model is. We then take the two-distribution mixture model. See Figure 6.24 comparing the BIC value of fitting mixture models up to 10 distributions.

Exercise 6.1

You could apply similar fitting analysis using the actual data. Prepare the data in Cartesian coordinates using the commands listed below.

```
1 unitsdata <- rep(1, length(angledata))
2 unitd <- toCartesian(angledata, unitsdata)
3 unitdx <- matrix(unitd, ncol=2)[ ,  1]
4 unitdy <- matrix(unitd, ncol=2)[ ,  2]
5 unitdmat <- cbind(unitdx, unitdy)
```

code/multimodalAnalysis.R

When the data are ready, fit them with the mixture of up to five von Mises distributions and compare their BIC. Conclude the number of preferred directions and their mean vectors. If possible, plot a curve of BIC versus the number of distributions, like Figure 6.24.

6.6.7 Calculating the Direction Against a Reference Point

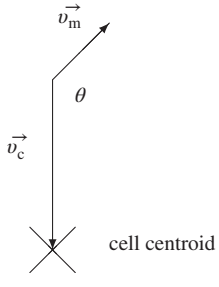
In case of the sequence `"eb1_8b.tif"`, EB1 movement is clearly directed outward from the cell center. To quantify this characteristic, their direction of movement with respect to the cell centroid could be calculated.

If a movement vector we calculated already (those in the *angledata*) is $\vec{v}_{\mathrm{m}}$ and a vector made from the starting point of $\vec{v}_{\mathrm{m}}$ to the position of cell centroid is $\vec{v}_{\mathrm{c}}$, then the angle made between these two vectors would be 0° for inward (retrograde) movement and ±180° for outward (anterograde) movement.

Such relative movement directions with respect to a reference point (cell centroid) could simply be calculated by using dot product and arccos:

$$\cos(\theta) = \frac{\vec{v}_{\mathrm{c}} \cdot \vec{v}_{\mathrm{m}}}{|\vec{v}_{\mathrm{c}}||\vec{v}_{\mathrm{m}}|} \tag{6.2}$$

$$\theta = \mathrm{acos}\left(\frac{\vec{v}_{\mathrm{c}} \cdot \vec{v}_{\mathrm{m}}}{|\vec{v}_{\mathrm{c}}||\vec{v}_{\mathrm{m}}|}\right) \tag{6.3}$$



θ obtained through above calculation ranges between 0 and π since `acos` returns only positive values. To know in which side the vector ($\vec{v}_{\mathrm{m}}$) is oriented about the vector toward cell centroid ($\vec{v}_{\mathrm{c}}$), we need to find out whether each angle is negative or positive (positive would be the counterclockwise rotation and negative would be the clockwise rotation). For this, we could use the cross product between the vectors $\vec{v}_{\mathrm{m}}$ and $\vec{v}_{\mathrm{c}}$, the value of which would be the determinant of 2×2 matrix constructed by

```
cbind(Vm, Vc)
```

If this value is negative, then we multiply the angle by −1.

The script for calculating the relative angle is below. This will be a bit complicated, so it is heavily documented (lines starting with #).

```
# Calculate relative angle
# define the coordinate of reference point
refpoint = c(209, 249)
# first prepare XY coordinates of the starting points of
# movement vectors V_m
stx <- ptdata$x[1:(length(ptdata$x)-1)]
stxc <- stx[dtraj == 0]
sty <- ptdata$y[1:(length(ptdata$y)-1)]
```

```
styc <- sty[dtraj == 0]
# prepare V_m start point to reference point vectors
# in scvecs, V_c vectors will be in each row.
scvecs <- cbind(refpoint[1] - stxc, refpoint[2] - styc)

# if labels in column should be stripped.
# commented out.
# svecs < cbind(stxc, styc, deparse.level=0)

# prepare V_m vectors in a same form as scvecs.
dvec <- cbind(dtxc, dtyc)

# a function for calculating angle between
# two vectors a and b using dot product
relativeangle <- function(a, b){
  ab <- apply(a*b, 1, sum)
  aa <- apply(a*a,  1, sum)
  bb <- apply(b*b,  1, sum)
  return (acos(  ab/ ( sqrt(aa) * sqrt(bb))))
}

# Using the function above, relative angles
relangs <- relativeangle(scvecs, dvec)

# above angle ranges between 0 and pi.
# using the cross product (determinants, as this is 2D),
# we determine if b is rotated clockwise or counter clockwise
# in terms of a. If clockwise, determinant > 0
determ <- function(a, b){
  det(cbind(a, b))
}

# initialize a vector to store determinants
outer <- rep(0, length(relangs))

# loop relangles for determining the rotation.
for (i in 1:length(relangs)){
  outer[i] <- determ(scvecs[i,], dvec[i,])
  if (outer[i] > 0){
    # clockwise rotation => negative angles.
    relangs[i] <-  relangs[i] * -1
  }
}
```

code/relativeAngle.R

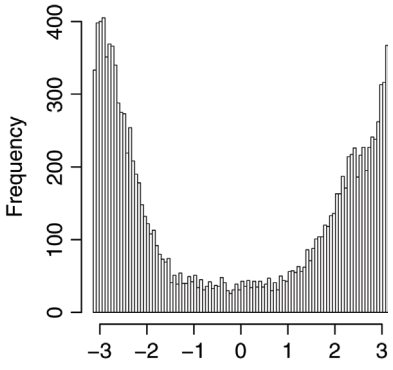


Figure 6.25 Histogram distribution of movement direction relative to a reference point (centroid). Outward movement is clear.

In line 23, we have something new: a function. In R, a function is declared in a way that

```
function_name <- function(arg1, ...) {
...
}
```

You could either explicitly declare a returned value like in line 27 or simply place a command in the last line that returns some value (see line 37 for another function). That value will then be returned from the function automatically.

The function `relativeangle` (line 23) calculates inner products between rows of vectors a and b to calculate θ (see Eq. (6.2) and (6.3)).

Plot the results by

```
hh <- hist(relangs, breaks=bins)
```

See Figure 6.25. Outward movement with respect to the cell centroid is clear.

Exercise 6.2

1. Test this result with its uniformity.
2. Confirm that the single von Mises distribution is the best model using the movMF package.
3. Calculate the mean direction (μ) and concentration parameter (κ).

Answers

1. Using Rao's spacing test, you could confirm that the distribution is not uniform.

```
> rao.spacing(relangs, alpha=0.05)

       Rao's Spacing Test of Uniformity

  Test Statistic = 159.0167
  Level 0.05 critical value = 136.94
  Reject null hypothesis of uniformity
```

2. The code below fits the relative angle data with various numbers of von Mises distribution.

```
# Conversion to unit vectors with Cartesian coordinates

runitdata <- rep(1, length(relangs))
runitd <- toCartesian(relangs, runitdata)
runitdx <- matrix(runitd, ncol=2)[,1]
runitdy <- matrix(runitd, ncol=2)[,2]
runitdmat <- cbind(runitdx, runitdy)

# testing with various peak numbers
rd1 = movMF(runitdmat, 1, run=10)
rd2 <- movMF(runitdmat, 2, run=10)
rd3 <- movMF(runitdmat, 3, run=10)
rd4 <- movMF(runitdmat, 4, run=10)
rd5 <- movMF(runitdmat, 5, run=10)
rd6 <- movMF(runitdmat, 6, run=10)

# Checking BIC
BIC(rd1)
BIC(rd2)
BIC(rd3)
BIC(rd4)
BIC(rd5)
BIC(rd6)
```

code/relAnglecheck.R

The output would be the following (your output values would most likely be different due to difference in the particle tracking parameters you used):

```
> BIC(rd1)
[1] -10250.61

> BIC(rd2)
[1] -10585.27

> BIC(rd3)
[1] -10565.42
```

```
> BIC(rd4)
[1] -10624.53

> BIC(rd5)
[1] -10610.64

> BIC(rd6)
[1] -10572.57
```

The best fit then is the mixture of four von Mises distributions.

```
> rd4
theta:
          runitdx          runitdy
1       0.0132931        0.3846624
2      -0.8619642       -0.6484031
3      -1.0037574       -0.7369927
4      -3.6764255        1.1298416
alpha:
[1] 0.2452032 0.2115761 0.2143157 0.3289051
L:
[1] 5368.025
```

This result is against the visual impression, but if you check the histogram of `relangs`, you will find that there are many vectors oriented in random directions and giving an offset to the overall distribution. The fitting results include these vectors. One way to remove such background effect is to filter the vectors by their length. By ignoring short vectors, we might get the fitting to be more optimized toward single von Mises distribution.

3. Instead of estimating the number of preferred directions, we could assume that there is indeed only a single von Mises distribution. With `vm.ml` function,

```
> vm.ml(relangs)$mu
[1] 3.136759
> vm.ml(relangs)$kappa
[1] 0.9761366
```

Since κ is the concentration parameter, we could represent how strong vectors are oriented in a single direction by presenting this value.

References

1 Miura, K. (2005) Tracking movement in cell biology. *Adv. Biochem. Eng. Biotechnol.*, **95**, 267–295.

2 Meijering, E., Smal, I., and Danuser, G. (2006) Tracking in molecular bioimaging. *IEEE Sign. Proc. Mag.*, **23** (3), 46–53.

3 Meijering, E., Dzyubachyk, O., and Smal, I. (2012) *Methods for Cell and Particle Tracking*, 1st edn, vol. 504, Elsevier.

4 Fisher, N.I. (1993) Statistical analysis of circular data. Available at http://books.google.com/books/about/Statistical_Analysis_of_Circular_Data.html?id=wGPj3EoFdJwC

5 Sbalzarini, I.F. (2006) A MATLAB toolbox for virus particle tracking. ICoS Technical Report, pp. 1–16. Available at https://www1.ethz.ch/mosaic/research/docs/Sbalzarini_SPT.pdf

6 Sbalzarini, I.F. and Koumoutsakos, P. (2005) Feature point tracking and trajectory analysis for video imaging in cell biology. *J. Struct. Biol.*, **151** (2), 182–195.

7 Zar, J.H. (1999) *Statistical Analysis*, 4th edn, vol. 19, Prentice Hall, Englewood Cliffs, NJ.

7
Quantitative Evaluation of Multicellular Movements in *Drosophila* Embryo

Perrine Paul-Gilloteaux[1,2] and Sébastien Tosi[3]

[1]Institut Curie, Centre de Recherche, Paris 75248, France
[2]Cell and Tissue Imaging Facility, PICT-IBiSA, CNRS, UMR 144, Paris 75248, France
[3]Institute for Research in Biomedicine (IRB Barcelona), Advanced Digital Microscopy, Parc Científic de Barcelona, c/Baldiri Reixac 10, 08028 Barcelona, Spain

7.1
Overview

7.1.1
Aim

In this chapter you will learn to track cell movements within the monolayer epithelium of a *Drosophila* embryo. The segmentation of the cells is performed by an ImageJ macro on microscope time lapses (movies) of the embryo presented in the maximum intensity projection (step 1). Another macro can optionally be used to discard weak cell–cell junctions that are likely to be segmentation errors (step 2). The tracking is then performed in Matlab from this binary stack (step 3). Finally, you will learn how to visualize the results of the cell tracking (step 4): the cell area evolution and the cell tracks.

7.1.2
Introduction

Quantitative information on the relative movement of cells and derived properties such as the evolution of cell areas and orientation are crucial to understand the organization of tissue and the fate of different cell subpopulations (Figure 7.1, left). The cell-based approach usually offers much information on the processes during tissue remodeling in comparison to estimating the velocity field performed by particle image velocimetry (PIV) [1,2] alone.

In this chapter, the cell tracking is performed in the apical plane of the tissue, so that the tissue is modeled as a polygon tiling. This is not just a mere simplification, but motivated by the fact that the strongest cell–cell junctions and most

Bioimage Data Analysis, First Edition. Edited by Kota Miura.

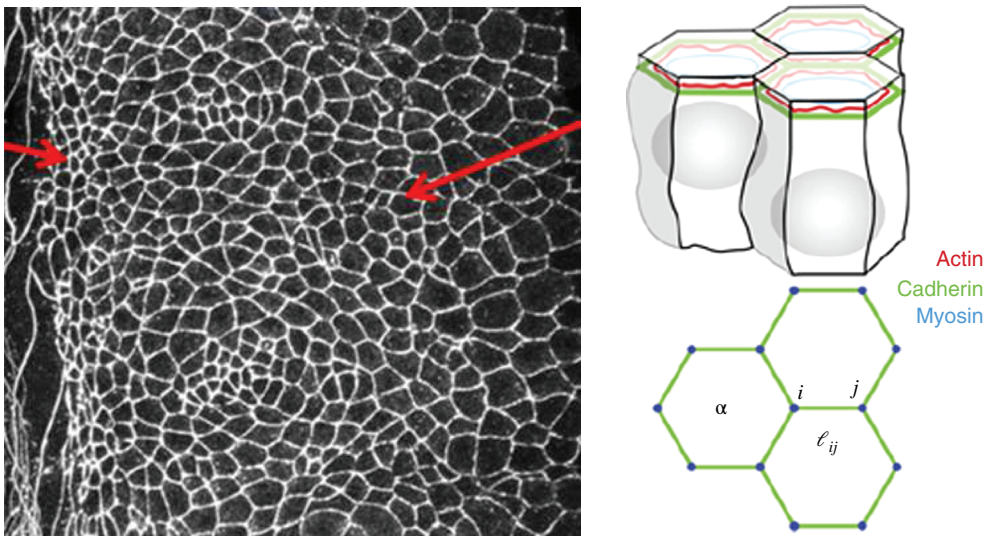


Figure 7.1 *Left:* In the apical plane of the tissue, the cell membrane appears as a polygon tiling. The red arrows point to the ventral (left) and dorsal (right) subregions. *Right:* Vertex model of cell shapes. (Figure taken from Ref. [3].)

of the cell internal shaping forces are believed to be exerted in the apical part of the tissue. For image analyses, the position of the cell vertices is an invaluable input to mechanical models, for example, the vertex model [3]. These models represent the epithelium as a 2D mechanical system with cells modeled as constant volume elastic polyhedrons sitting below the apical plane (Figure 7.1, right).

7.1.3 Data Sets

Data sets show the apical tissue of the embryo, which is imaged by a spinning disk microscope. The cell–cell junctions are labeled by a GFP-E-cadherin construct. Three time lapses of data are provided that show the maximum intensity projection of different tissue morphologies and dynamics. In this chapter we will work only on the *first time lapse*, the other movies can be used in the assignments.

7.2 Step 1: Cell Segmentation

The cell membrane appears brighter than the cytoplasm and, being the natural cell barrier, is highly suitable for cell segmentation. We will segment the cells in each time frame and generate a segmentation mask (binary image) for each

frame. The final result is a binary stack (one image per frame). Here, we focus on a workflow based on the watershed algorithm, another possible solution is provided in the appendix.

7.2.1 Workflow

Get Image Files

Open the time lapse "TissueMovie1.tif" in ImageJ.

Preprocessing

The purpose of the preprocessing is to smooth out the images to facilitate the segmentation. Filter the whole image stack with a Gaussian filter of radius 1.5 pixels: `Process > Filters > Gaussian Blur...`

Regional Minima Detection

The cells are segmented by finding internal starting points (ideally a single point close to the cell center) together with the region covered by the cell utilizing the watershed algorithm. The watershed can be conceptualized by coding pixel intensities as height: The cells then appear as basins separated by higher watershed lines. The starting points are intensity regional minima: a region surrounded by brighter pixels.

To apply the watershed, call `Process > Find Maxima...` with the option "Light Background" ticked (find minima instead of maxima). Choose a good value for "Noise Tolerance" (minimum relative height of surrounding bright barrier) to get rid of spurious minima and obtain exactly one regional minimum in most cells. For this, tick the "Preview" option to tune the noise tolerance.

Once you find a working value, choose the option "Segmented Particles" as output type: This will apply the watershed algorithm from the detected regional minima. This ImageJ command cannot process a whole stack at once, we will have to write a macro to generate the complete segmentation stack slice-by-slice.

Exercise 7.1: Writing a Macro to Segment the Cells in the Complete Stack

Record the sequence of operations that you manually performed in the previous section. Modify the code generated by the macro recorder in order to automatically process all the slices of the time lapse. The result should be a binary stack showing the segmented cells.

Hint 1: You should start by creating an 8-bit empty stack with the same dimensions as the original stack. Each slice of the original should then be processed by the previous sequence of operations.

Hint 2: You can select a slice of the active stack with the macro function setSlice(). A for-loop can be implemented to wander all the slices. The macro function nSlices returns the number of slices of the active stack.

Hint 3: To transfer the resulting segmentation mask to the empty image stack, you can use copy/paste option. At each iteration, ensure to process the correct slice of the original stack and copy it to the corresponding slice of the empty stack. The output stack should look as shown in Figure 7.2.

Use the macro you just wrote to process the first movie (a possible solution is provided in "Step1SampleCodeLoop.ijm"). To check the results of the segmentation, you can use `Image > Color > Merge Channels` to overlay the original stack (in gray) and the cell boundaries (e.g., in green) as shown in Figure 7.3.

Warning: To merge two images, they should have the same bit depth. If necessary, convert the original image to 8 bit.

As you can observe, it is quite likely that you do not get a perfect segmentation; some "false" cell junctions are sometimes created when several regional minima are detected in the same cell (inhomogeneous intensity inside the cell). This is often referred to as "oversegmentation" and can be mitigated by a proper preprocessing (filtering, background subtraction, etc.). In the next exercise we will implement a simple manual correction. Step 2 implements an algorithm to automatically discard weak junctions by measuring their mean intensity.

Note: Undersegmentation is the opposite phenomenon, it takes place when a single minimum is detected inside two neighbor cells (e.g., breach or weak cell junction). This is usually trickier to correct for; however, if cells can be assumed convex, then splitting lines can be drawn between concavities, for instance, by applying `Process > Binary > Watershed`.

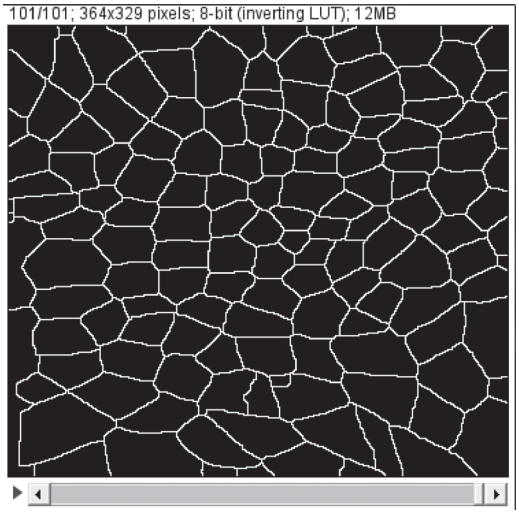


Figure 7.2 Output of the macro.

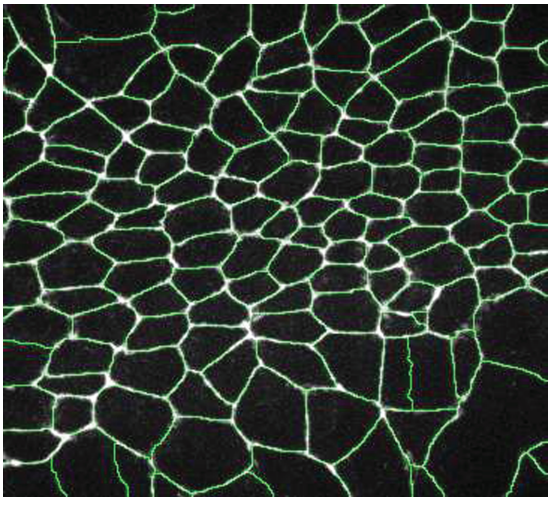


Figure 7.3 Channel merging of original stack (grayscale) and segmentation (green). Some "false" cell junctions have been created, due to the presence of several regional minima in the same cell.

Exercise 7.2: Understanding an Existing Macro

The ImageJ macro "Step1CellSegment.ijm" performs all the steps previously described and also enables the manual correction of the segmentation mask. Try it out and read the code!

The last section of the macro implements the manual correction. Try to understand how this user interaction is written and how the cell merging is implemented. Refer to the macro language documentation if you do not understand some macro functions.

Error detection and manual inspection/correction are fundamental steps of any automatic analysis. Perform a thorough manual correction of the *first frame* of the movie and save the binary stack to file as it will be used in steps 2 and 3.

Note: ImageJ segmentation masks by default are LUT inverted images so that the objects appear as black over a white background. When writing a macro, it is always a good idea to force this default behavior at initialization from `Process` > `Binary` > `Options...`. The options should appear as shown in Figure 7.4.

7.2.2 Summary of Tools Used

- `selectImage(ID)`: Activates the image with the specified ID (a negative number). If ID is greater than zero, it activates the IDth image listed in the Window menu. The ID can also be an image title (a string).

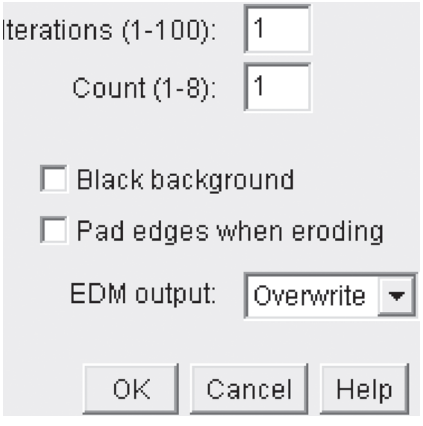


Figure 7.4 IJ default binary options.

- *Gaussian Blur* `[Process > Filters > Gaussian Blur]`: Smoothes out an image by performing convolution with a 2D Gaussian kernel of user-defined sigma. This helps reducing noise and disparities in the image, but smears out object edges and details.
- *Find Maxima* `[Process > Find Maxima]`: The command Find Maxima identifies regional maxima: A regional maximum is the highest intensity pixel that is surrounded by a closed valley of lower intensity pixels. The noise tolerance defines the minimal intensity difference between the regional maximum and the highest intensity of its surrounding valley.
- *Copy/Paste* `[Edit > Copy], [Edit > Paste]`: The default behavior is to copy all the intensity values of the pixels inside a ROI to another (active) image. The command `Edit > Paste control` allows you to change the way the image is copied (e.g., the background can be transparently superimposed).
- *Merge Channels* `[Image > Color > Merge Channels...]`: Merging different channels allows you to simultaneously visualize them in the same hyperstack. It is possible to keep the original LUT of each channel when merging them.

The following are the useful macro functions:

- nSlices: Returns the number of slices/frames in the active stack.
- setSlice(): Change the position of active stack slice slider.

7.3 Step 2: Removal of Weak Segments (Optional)

We observed that oversegmentation can occur when several regional minima are detected in a nonhomogeneous cell. This situation can be revised since the

intensity along a "false" cell junction is likely to be weaker than for a "valid" cell junction. We are now going to measure the mean intensity along each cell junction in the original image and remove all cell junctions with mean intensity below an empirical value.

7.3.1 Workflow

The first task consists in detecting the cell junctions. To do so, we will first use the ImageJ command `Plugins > Skeleton > Skeletonize (2D/3D)` to enforce that the cell junctions are represented by a single pixel wide line.

Note: Ensure to invert the segmentation mask before skeletonization since now the objects of interest are not the cells but the cell junctions! Also note that skeletonize (2D/3D) interprets a stack as a 3D image: You should again process slice-by-slice (for loop).

Next call `Plugins > Skeleton > Analyze Skeleton (2D/3D)` on each skeleton image to code cell junctions and vertices (Figure 7.5) with a different intensity. The cell junctions are now easy to segment from the output of `Analyze Skeleton (2D/3D)`: They are connected particles with a gray value equal to 127.

In the last step of the macro, the detected cell junctions will be wandered (loop) for each image slice and classified as "valid" or "weak" based on their mean intensity in the original image.

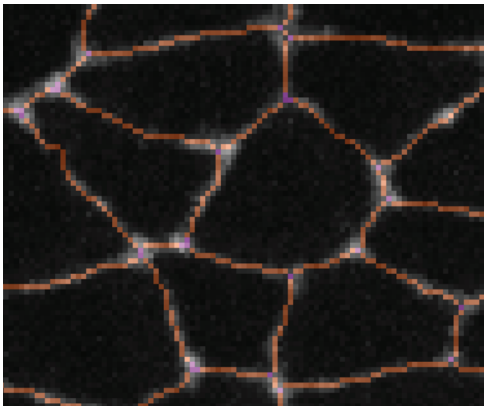


Figure 7.5 Overlay of the original image and the analyzed skeleton. The cell junctions appear in orange (gray level 127), the cell vertices in purple (gray level 70), and the end points in blue (gray level 30, invisible in this image).

Exercise 7.3: Modifying an Existing Macro

For this exercise, we provide a fully functional macro:

"Step2WeakJunctionRemoval.ijm": The original movie and the segmentation mask generated in step 1 both must be opened and respectively named "TissueMovie1.tif" and "ParticlesStack.tif" before launching the macro.

All the operations are by default performed in batch mode (no apparent windows) so that the intermediary steps are not shown. To better understand the sequence of operations, perform the following tasks:

- Comment the lines that are enabling and disabling batch mode: setBatchMode(true) and setBatchMode("exit and display")
- The segmented cell junctions are first added to the ROI manager before their intensity is measured. Add a pause (waitForUser) in the loop over the junctions, just after the ROI selection.
- Change the value of the variable JunctionThr and observe the effect (you can, for instance, use very low and very high values), try to optimize this value.

To compare the results before and after correction, you can again use `Image > Color > Merge Channels` to create a hyperstack with three channels: original image and masks before and after correction (Figure 7.6).

Save the corrected binary mask to file.

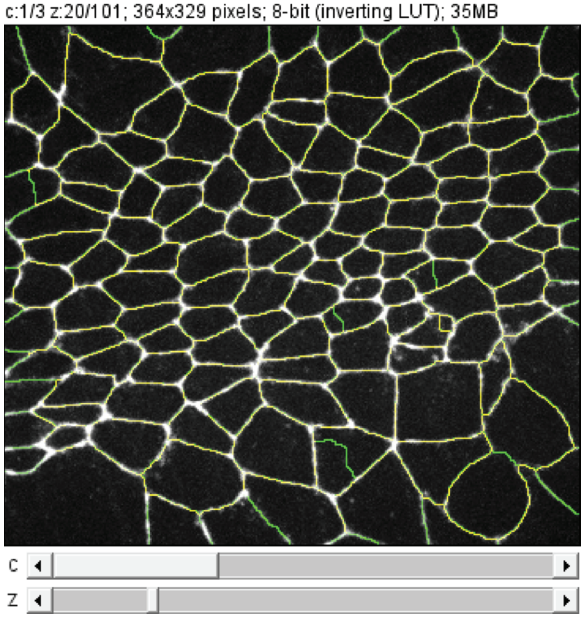


Figure 7.6 An example slice after cell segmentation correction. Removed junctions appear in green and kept junctions in yellow (red + green).

Exercise 7.4: Making the Macro More Generic

It is not very convenient to have the image names and weak segment threshold hard-coded in the macro. Display a dialog box at the beginning of the macro with three UI elements: two drop-down menus (one for the original movie and another for the binary mask) and one numerical field (junction threshold). Drop-down menu can be added with the command Dialog.addChoice and numerical field with the command Dialog.addNumber.

To fill the names of all opened images in the drop-down menus, you will have to first write a small loop over all the opened images to retrieve their titles and store them to an array of strings that will be passed as argument to Dialog.addChoice.

Hint: Even if this is usually not advised, it is possible to select an image by passing positive numerical values to the function selectImage, these indexes run from 0 to nImages-1 and correspond to the order of creation/opening of the images.

A solution to this exercise is the macro:

"Step2WeakJunctionRemoval_dialog.ijm."

Exercise 7.5: Advanced – Automated Threshold Selection

Try to automate the selection of the threshold used to discard the weak cell junctions (the solution is not provided).

Hint: The average intensities of the "valid" junctions can be assumed quite uniform. So a good strategy is to select the threshold as a fraction of the overall median of junction intensities (as long as "valid" junctions are in majority).

7.3.2 Summary of Tools Used

- *addNumber(label,default):* Adds a numeric field to the dialog, using the specified label and default value.
- *addChoice(label,item):* Adds a pop-up menu to the dialog, where items is a string array containing the menu items.
- *Skeletonize* – `Plugins > Skeleton > Skeletonize (2D/3D)`*:* It extracts the medial axis of each connected particle of a binary mask, this medial axis is also known as skeleton (points lying at exactly the same distance from two edges of a connected particle in the segmentation mask). Skeletonization is a morphological operation based on iterative erosions (see section on *Morphology* in Module 1). The command also provides an extensive report of the skeleton branch statistics and can optionally prune the shortest branches.
- *AnalyzeSkeleton* – `Plugins > Skeleton > AnalyzeSkeleton (2D/3D)`*:* This command introduced in Ref. [4] is part of the Fiji distribution. From a skeleton (graph-like binary image), it identifies the vertices (crossing of junctions) based

on the number of neighbors of each pixel. The junctions are coded with gray level 127, the vertices with gray level 70, and the end points with gray level 30.

- *waitForUser:* The command waitForUser("Message") interrupts the macro until the user presses the "OK" button. Interaction with ImageJ such as calls to internal commands or image selection/edition is possible. This behavior is different from the result as you invoke showMessage("message"); in this case, no interaction with ImageJ is allowed: ImageJ is "frozen" until the user presses "OK" (pressing "Cancel" interrupts the macro).

7.4 Step 3: Cell Tracking

In this section, we are going to track the cells in Matlab. The input of the script is the binary stack from step 1 (or step 2) and the output is a *label* mask with each cell filled by a unique gray level throughout the whole stack.

7.4.1 Introduction to Tracking

Most particle trackers such as ImageJ Particle Tracker 2D/3D and TrackMate are built for spot-like particles: The particle linking is performed over a list of detected spots. The linking can be straightforward, for example, linking a spot to the closest spot in the next frame, or more advanced as described in Refs [5,6].

These trackers are not adapted to nonspot-like (or ellipsoid) objects such as the cells of a tissue. A possible solution is to create a binary stack made of detected object centroids from a segmentation mask and to process this stack with the tracker. This is left as an exercise in the assignments.

Another strategy is to make use of the overlap between the particles in two consecutive frames. This will be illustrated in the following.

Exercise 7.6: Analyzing Connected Particles with Matlab

First we review a simple Matlab script importing a binary image stack, analyzing 3D connected particles, and exporting the resulting label mask stack to file.

Open the Matlab script "Step3ConnectedParticles3D.m." Before launching the script, set the variable BaseFolder to the folder where you saved the stack "DummyStack.tif." The script exports the label mask stack to a file "LabeledDummyStack.tif" to the same folder. After launching the script, open the original stack and the exported stack in ImageJ and go through the code to understand how it works.

Hint: The function bwconncomp allows analyzing connected particles in a *n*-dimensional image (here a 3D image). See Figure 7.7 for an illustration in two dimension.

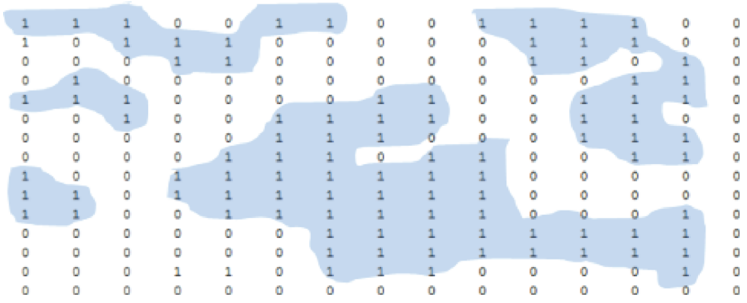


Figure 7.7 Illustration of 2D connected particles in a 2D mask; in 3D the same principle holds but each voxel has 26 neighbors instead of 8.

Exercise 7.7: Tracking Cells with Matlab

The trick we will use is to interpret the binary stack we generated in step 1 as a 3D stack with time being the third dimension: The cells overlapping from frame to frame are now connected in three dimension (see Figure 7.8) !

In the remainder of the text, the word *particle label* refers to the label of a 2D connected particle (cell) in a given time frame (can be different for the same cell in two different frames). The word *object label* refers to the unique label of the cell along time (considered as a 3D object).

Exercise 7.7.1: Understanding the Matlab Script

Open the Matlab script "Step3TissueCellTrackv10_simple_incomplete.m." For now, do not launch the script but read the code and the functional block diagram provided in Figure 7.9. Try to associate each block with a section of the program.

Notes:

The loop that runs the code for each object is not included yet, do not try to find the associated code. Also, an operation to run the code for each frame is missing at this point (next exercise).

Image stack importation and exportation are very similar to the previous example.

The particles touching the edges are discarded to avoid tracking incomplete cells.

The separation between the cells has to be thick enough to avoid any merging of neighboring cells (in 3D), the cell junctions are enlarged by binary dilation (disk element of radius BoundDilate).

To close the top and bottom of the 3D particles, an empty image is added at the beginning and at the end of the movie: The binary stack passed to bwconncomp holds two slices more than the original stack.

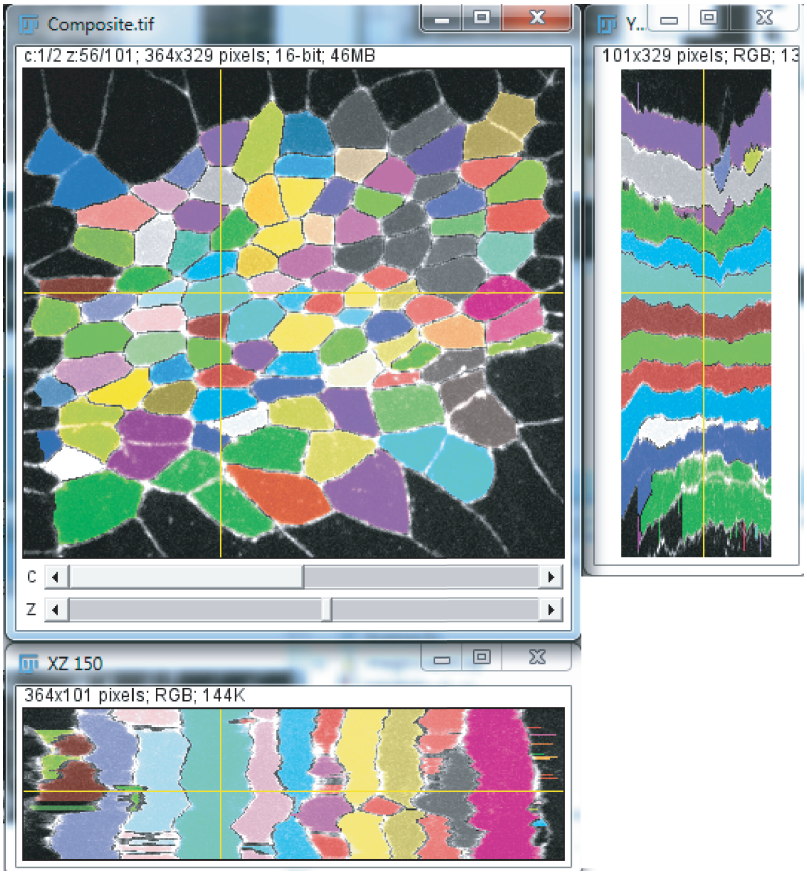


Figure 7.8 The cell overlap in time is apparent from orthogonal views when the movie is viewed as a 3D stack (`Image > Stacks > Orthogonal Views`).

The 3D object label mask (ObjLbl) is created by writing the labels slice-by-slice (second loop). It can also be created by the Matlab function labelmatrix as in the previous script. This is just to illustrate an alternative method.

The cell junctions are eroded back to their initial width at the end.

Exercise 7.7.2: Finding the Missing Operation

Before launching the script, set Basefolder to the folder where the input file (binary mask) is located. The variable fname holds the name of segmentation mask saved in step 1 (or step 2). After launching the script, inspect the exported results Tracking-obj1-10.tif in ImageJ. You will notice that the script is not working since an operation is missing: Identify it and add it to the code.

Hint: The missing operation can be performed by a single line of code. A solution to the exercise is given in

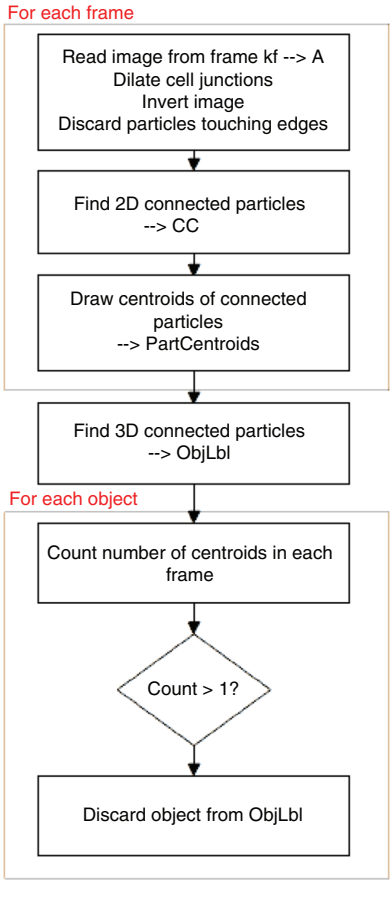


Figure 7.9 Workflow of the cell tracker (Algorithm 1 – TissueCellTrackv10).

"Step3TissueCellTrackv10_simple.m."

To better visualize the exported stack in ImageJ, you can apply the LUT Random.lut provided in the code folder. To do this, you first have to copy the LUT into the subfolder *luts* of ImageJ installation folder and call `Help > Refresh Menus`.

Note: The script also exports a stack holding the centroids of the 2D particles found in each frame. It will be used during the assignments to track the cell centroids.

This simple algorithm suffers from a severe limitation: Dividing or transiently merging cells (undetected cell junctions) are assigned the same label. In the next section, we will implement a procedure to detect these situations and automatically discard the problematic cells.

Exercise 7.7.3: Discarding Erroneous Objects

You can now open the script "Step3TissueCellTrackv10.m." This algorithm is described by the block diagram in Figure 7.9. In the second part, a loop is performed over each 3D object and the number of 2D particles (centroids) inside the object is computed for each frame. If more than one centroid is found in *at least one frame*, then the object is discarded in all frames (label set to 0). Of course, this is not ideal, but following the same idea one could detect events of merge and division and split the 3D object in frames where merge and division occur (set label to 0) before recomputing the 3D connected objects.

7.5 Step 4: Feature Extraction

In this section, we will plot the evolution of the cell area against time (i.e., the recorded frame) as well as track the position of the centroids of some of the cells. The input of all the scripts of this section is the label mask that we have computed (and saved to file) in step 3.

Exercise 7.8: Plotting Cell Area Evolution

Open the Matlab script "Step4PlotCellAreaExercise.m" and check that the base folder and file name are correctly set in the script preamble.

The script only displays the area evolution for a subset of cells of the label mask. The variable displayLbl is a vector taking the indices of the user-defined cells from which the information is to be extracted (ensure that these indices do not exceed the maximum value of the label mask!). The script is incomplete: In the loop over all the time frames, you should add the missing code to compute the areas of the cells of interest and store them in the variable Area(cellindex, frame).

Hint: Inside the loop over the time frames, you should write a loop over all the labels stored in displayLbl, compute the areas of each cell, and store this area to the matrix Area(for the correct cell, for the correct frame index). A way to compute the area of a cell at a given frame is to count how many pixels are set to the label of this cell in that particular frame.

The solution to the exercise is provided in "Step4PlotCellArea.m."

Exercise 7.9: Plotting Cell Centroid Tracks

Try now to retrieve the centroids of the cells in each frame with regionprops (Obj,'Centroid'). Store theses positions to two arrays: X(frame) and Y(frame) and plot the tracks to a 2D map. The solution to the exercise can be found in "Step4PlotCellAreaTrack.m."

7.5.1 Complete Track Plotting

A program plotting all the cell tracks (with start and end points labeled) and overlaying them to the first frame of the label mask is provided in "Step4CellTrackPlotter.m." The workflow of the program is described below and a possible output is illustrated in Figure 7.10 (without the label mask overlay).

First we define a minimal track length MinimalLengthofTrajectory to discard short tracks that are often erroneous.

In order to get the total number of objects, we find the last label index nbmaxcell (pixel with maximum value in the object label stack).

Then we create a vector of Matlab structures called *cells*. The vector is indexed by the object label. Each element of this structure array gathers two features of a given cell: the list of time frames where it is present and the list of its centroid positions in these frames. The structure is filled by the results returned by the Matlab function regionprops. Other features can be easily added to the structure array.

Finally, we plot the cell centroids for each time frame by sequential calls to *plot*. Only the tracks of length above MinimalLengthofTrajectory are plotted and the graphics are directed to the same canvas by calling *hold on*. The tracks are additionally decorated to mark their start and end points.

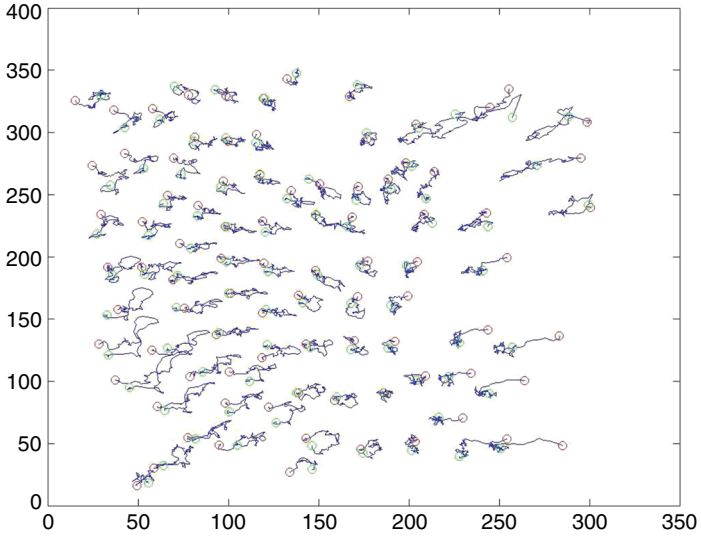


Figure 7.10 Example of cell tracks plotted for movie 1. Each blue line represents a cell trajectory, the first point of the trajectory is a red circle and the last point a green circle.

7.6 Assignments

If you intend to use the other two movies, you will notice that they suffer from a strong lateral drift; it is hence important to first register them, for instance, by applying `Plugins > Registration > StackReg` before the segmentation. This will allow a more efficient tracking in step 3.

- *Tracking of cell centroids:* In step 3 we generated a stack holding the cell centroids extracted from the segmentation mask. We can use this centroids stack as input to a spot-like particle tracking algorithm. In light microscopy, an infinitely small particle would never appear as a single isolated point but rather as a Gaussian-like shape (the point spread function of the microscope). This can easily be simulated by performing a 2D Gaussian blurring (radius around 1 pixel) of the centroid stack.

  After this filtering process, the centroid stack with `Plugins > Mosaic > Particle Tracker` or `Plugins > Tracking > TrackMate`. You should set the particle size to the minimum setting and prefer DoG (Difference of Gaussian) to LoG (Laplacian of Gaussian) detector in `TrackMate`.
- *PIV analysis*: To complement the analysis, we will now apply PIV to the same movies. The PIV is a procedure to estimate the velocity field of a time lapse at discrete positions and for each time frame. The PIV will be computed by ImageJ with `Analyze > Optic flow > PIV Analysis`. This function generates two images: The image U codes the X component of the velocity field, while the image V codes its Y component. We will export these two images to file, import them in Matlab, and visualize the velocity field (vector field) with the function *quiver*. To test this workflow, follow these steps:
  1) Create a two-frame movie by duplicating an image with `Image > Duplicate...` and slightly shift the second frame. Save this stack as "twoframes.tif."
  2) Call ImageJ `Analyze > Optic Flow > PIV analysis` on the stack. The displacements are represented as a color-coded image and they are also provided as two images (one for each speed component).
  3) Spatially average the U and V components by applying `Process > Smooth`.
  4) Save the filtered images U and V *as text files* with `File > Save As > Text`. When saving an image as text file, the values of the pixels are sequentially written as text (strings) separated by spaces. The values are written line-by-line with a return carriage at the end of each line.
  5) Launch the following Matlab script ("XAssignVisPivMatlab.m"):

```
% Folder where "U.txt" and "V.txt" are saved
Basefolder = '...\';

% Load the text files and copy the values to the
    matrices u and v
u = load([Basefolder,'U.txt']);
```

```
v = load([Basefolder,'V.txt']);

% Load first frame of the time-lapse
imagetissue = imread([Basefolder,'twoframes.tif'],1);

% Display the vector field overlaid on the first
    frame of the time-lapse
imshow(imagetissue,[]);
hold on;
quiver(v,u);
```

code/XAssignVisPivMatlab.m

The ImageJ macro "XAssignPIVCompute.ijm" performs the previous workflow. The script "XAssignVisuOutputPIV.m" is an advanced version of the previous Matlab script to visualize the vector field in Matlab. The output should look as in Figure 7.11.

- *Overlap-based cell tracking:* The tracking algorithm based on 3D connectivity is simple, but the results are not fully satisfactory as dividing and merging cells are discarded from the label mask. The algorithm proposed in this section will correctly track most of the cells up to a division/merging.

  Here the cell linking is performed iteratively and by processing pairs of consecutive time frames. The tracking starts from the first frame and proceeds through the time lapse, it is *extremely important* to check that the segmentation mask is valid in the *first frame*.

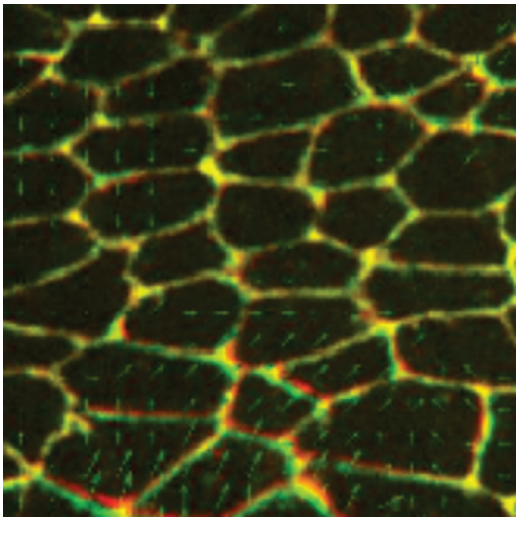


Figure 7.11 Example result of the PIV analysis. The first frame is green and the second frame is red. The displacements between first and second frames are indicated by green arrows.

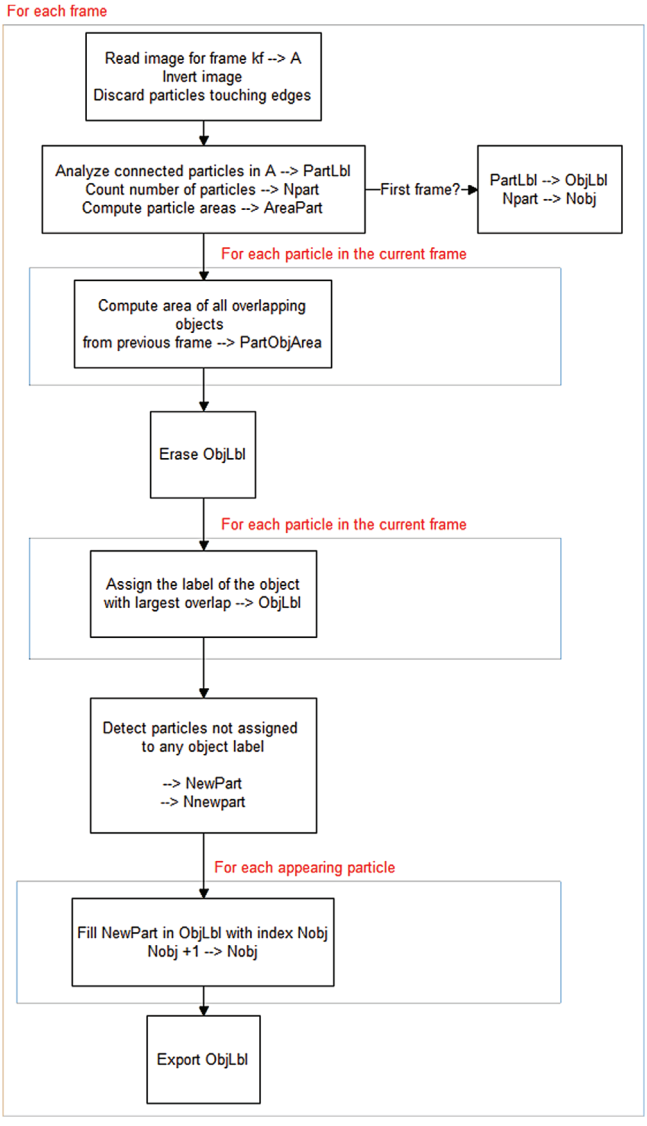


Figure 7.12 Workflow of the cell tracker (Algorithm 2 – TissueCellTrackv20).

Open the script "XAssignTissueCellTrackv20.m" and launch it after setting BaseFolder. A block diagram of the script is provided in Figure 7.12.

The object label mask (ObjLbl) is now built recursively: In the first frame, the 2D connected particles (PartLbl) are copied to the object label (ObjLbl) of the first frame. Particles here means connected objects in one frame, i.e candidate cells to be assigned to an object (i.e tracked cell). The 2D particles are then

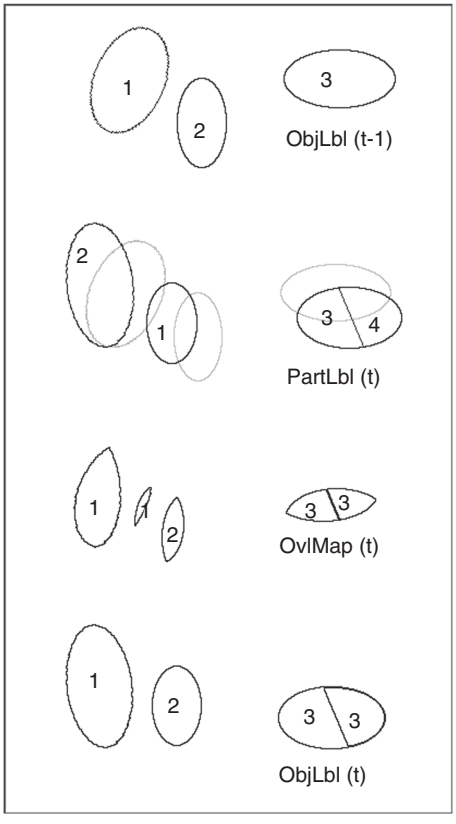


Figure 7.13 Steps involved to build the object label mask ObjLbl at time t from the object label mask at time t-1 and the particle label mask at time t. OvlMap(t) represents the overlap between the particles at time t and the objects at time t-1. The numbers represent the labels of the objects.

analyzed in the next frame t (PartLbl t) and their overlap with the object(s) in frame t-1 (ObjLbl t-1) are computed. Each particle is then assigned the most likely object label from the previous frame (maximum overlap), this label is written to the object label mask of the current frame (ObjLbl t). The process is then iterated until the last frame of the movie.

The label masks are illustrated for a simple case in Figure 7.13. It can be clearly understood that if there is no overlap between the same cell in two consecutive frames (large drift or fast movement), the linking will fail. Two other problems arise if the largest overlap does not take place between the correct pair of cells or if two cells get transiently merged.

Try to modify the script "XAssignTissueCellTrackv20.m" so that a particle is assigned the label of the object with largest overlap *only* if this overlap is larger than a user-defined area threshold (e.g., 70% of the area of the particle). The solution to the exercise is found in "XAssignTissueCellTrackv20_solution.m."

Solutions to the Exercises

Exercise 7.1

```
// Initialization
NoiseTol = 5;
run("Options...", "iterations=1 count=1 edm=Overwrite");

// Filter input image
run("Duplicate...", "title=Filtered duplicate");
run("Gaussian Blur...", "sigma=1.5 stack");

// Create empty stack of same size as active image
newImage("ParticlesStack", "8-bit Black", getWidth(),
     getHeight(), nSlices);
run("Invert LUT");

// Main loop
for(i=1;i<=nSlices;i++)
{
  selectImage("Filtered");
  setSlice(i);
  run("Find Maxima...", "noise="+d2s(NoiseTol,2)+"
     output=[Segmented Particles] light");
  rename("Particles");
  run("Copy");
  selectImage("ParticlesStack");
  setSlice(i);
  run("Paste");
  selectImage("Particles");
  close();
}
run("Select None");
```

"code/Step1SampleCodeLoop.ijm"

Exercise 7.2.3: The Code Should Be Opened from the Provided Material ("Step1CellSegment.ijm" and "Step2WeakJuctionRemoval.ijm")

Exercise 7.4

```
//Parameters
if(nImages<2)exit("At least two images should be opned!");
ImageNames = newArray(nImages);
for (i=0; i < nImages; i++)
{
  selectImage(i+1);
  ImageNames[i] = getTitle();
```

```
}
Dialog.create("Select Images");
Dialog.addChoice("Original Movie", ImageNames, ImageNames[0]);
Dialog.addChoice("Binary Mask", ImageNames, ImageNames[1]);
Dialog.addNumber("Threshold", 50);
Dialog.show();
OriginalMovie = Dialog.getChoice();
SegmentedCells = Dialog.getChoice();
JunctionThr = Dialog.getNumber();

// Initialization
run("Options...", "iterations=1 count=1 edm=Overwrite
   do=Nothing");
run("Set Measurements...", "  mean redirect="+OriginalMovie+"
   decimal=2");
setBatchMode(true);
newImage("CorrectedMask", "8-bit Black", getWidth(),
   getHeight(), nSlices);

// Main loop over the slices
for(s=1;s<=nSlices;s++)

{

  // Set current slice in all stacks
  selectImage("CorrectedMask");
  setSlice(s);
  selectImage(OriginalMovie);
  setSlice(s);
  selectImage(SegmentedCells);
  setSlice(s);

  // Skeletonize and identify junctions / vertices
  run("Duplicate...", "title=Copy");
  run("Invert", "slice");
  CopyID = getImageID();
  run("Skeletonize (2D/3D)");

  run("Analyze Skeleton (2D/3D)", "prune=none prune");
  AnalyzedSkeletonID = getImageID();

  // Add all junctions to ROI manager and measure mean
      intensity in original stack
  setThreshold(100, 255);
 // Threshold junctions: vertices value: 70, junctions value: 127
  run("Analyze Particles...", "size=0-Infinity circularity=0.00-
     1.00 show=Nothing display clear add");
  roiManager("Show None");
```

```
51
52    // Select non null pixels in the skeleton --> mask
53    selectImage(AnalyzedSkeletonID);
54    setThreshold(1, 255);
55    run("Convert to Mask", "method=Default background=Dark
         black");
56    run("Invert LUT");
57    resetThreshold();
58
59    // Erase the weak junctions
60    N = roiManager("count");
61    for (i=0;i<N;i++)
62    {
63      if(getResult("Mean",i)<JunctionThr)
64      {
65        roiManager("Select", i);
66        run("Set...", "value=0 slice");
67      }
68    }
69
70    // Copy to CorrectedMask
71    run("Select All");
72    run("Copy");
73    selectImage("CorrectedMask");
74    run("Paste");
75
76    // Cleanup
77    selectImage(AnalyzedSkeletonID);
78    run("Close");
79    selectImage(CopyID);
80    run("Close");
81  }
82
83  // Exit
84  selectImage("CorrectedMask");
85  run("Invert", "stack");
86  run("Invert LUT");
87  run("Select None");
88  setBatchMode("exit & display");
89  run("Set Measurements...", "  mean redirect=None decimal=2");
```

"code/Step2WeakJunctionRemoval_dialog.ijm"

Exercise 7.6

```
1  BaseFolder = '...\';
2  fname = strcat(BaseFolder,'DummyStack.tif');
3  expfname = strcat(BaseFolder,'LabeledDummyStack.tif');
```

```
% Gather ParticleStack image info
info = imfinfo(fname);
num_images = numel(info);
Height = info(1).Height;
Width = info(1).Width;

% Initialize buffer image
DummyStack = zeros(Height,Width,num_images);

% Read images (loop over frames)
for kf = 1:num_images
    DummyStack(:,:,kf) = imread(fname, kf, 'Info', info);
end

% Find connected particles (3D)
CC = bwconncomp(DummyStack);

% Generate label mask
L = labelmatrix(CC);

% Erase output file (if exists)
if exist(expfname, 'file')
    delete(expfname);
end

for kf = 1:num_images
    imwrite(uint16(L(:,:,kf)), expfname, 'WriteMode', 'append',
       'Compression','none');
end
```

"code/Step3ConnectedParticles3D.m"

Exercise 7.7

```
%% Initialization
clear all;
close all;
BaseFolder = '...\';
fname = strcat(BaseFolder,'ParticlesStack.tif'); %
     Segmentation mask
expfnameobj = strcat(BaseFolder,'Tracking-obj1-10.tif'); %
     Object labels
expfnamepartcentroids = strcat(BaseFolder,'Tracking-
     Centroids-v-10.tif'); % Centroid stack

%% Parameters
startframe = 1;
```

```
BoundDilate = 5;

%% Gather ParticleStack image info
info = imfinfo(fname);
num_images = numel(info);
Height = info(1).Height;
Width = info(1).Width;

%% Create image buffers with same size of the original image
     (+ 2slices)
ImageCopy = zeros(Height,Width,num_images+2);
ObjLbl = zeros(Height,Width,num_images+2);
PartCentroids = uint8(zeros(Height,Width,num_images+2));

%% Loop over image frames (2D processing)
for kf = 1:num_images
    disp(kf);

    % Import mask for this frame
  A = imread(fname, kf, 'Info', info);

    % Dilate cell junctions
  A = imdilate(A, strel('disk',BoundDilate));

  % Process mask and analyze connected particles (2D)
  A = 255-A; % Mask coming from ImageJ (inverted LUT)
    A = imclearborder(A); % Remove objects touching the image
       borders

    % Store current frame to ObjLbl
    ImageCopy(:,:,kf+1) = A;

    % Find connected particles in current frame and compute
       their centroids
    CC = bwconncomp(A);
  centroids = regionprops(CC,'centroid');

  % Draw centroids of connected particles
  for l = 1:length(centroids)
    centroidl = centroids(l).Centroid;
        PartCentroids(round(centroidl(2)),round(centroidl(1)),
           kf+1) = 1;
    end

end

%% Analyze connected particles (3D) to identify objects and
     fill label mask
```

```
objList = bwconncomp(ImageCopy);
Nobj = objList.NumObjects;
for label = 1:Nobj
   ObjLbl(objList.PixelIdxList{label}) = label;
end

%% Images post-processing and exportation

% Erase output file (if exists)
if exist(expfnameobj, 'file')
    delete(expfnameobj);
end

% Erode the cell junctions and export the object label mask
for frame=2:num_images+1
     ObjLbl2D = imdilate(ObjLbl(:,:,frame),strel('disk',
         BoundDilate));
     imwrite(uint16(ObjLbl2D), expfnameobj, 'WriteMode',
         'append',  'Compression','none');
end

% Erase centroid stack file (if exists)
if exist(expfnamepartcentroids, 'file')
    delete(expfnamepartcentroids);
end

% Export the centroid stack
for frame = 2:num_images+1
  imwrite(uint8(PartCentroids(:,:,frame)),
    expfnamepartcentroids, 'WriteMode', 'append',
    'Compression','none');
end
```

"code/Step3TissueCellTrackv10_simple.m"

Exercise 7.8

```
clear all;
close all;
startframe = 1;
displayLbl = [54 57];
BaseFolder = '...\';
fname = strcat(BaseFolder,'Tracking-obj1-10.tif'); %
     Exported stack (object)

%%%% Read the object label mask (same label = same
     cell over time)%%%%%%%
info = imfinfo(fname);
```

```
num_images = numel(info);
endframe = num_images;

% get information from last frame
A = imread(fname, endframe, 'Info', info);

% Find highest label: total number of cells
nbmaxcell = max(max(A));

% Initialize arrays
Area = nan(nbmaxcell,endframe);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

for kf = startframe:endframe
   A = imread(fname, kf, 'Info', info);
   for Lbl=displayLbl
       Obj = (A==Lbl);
       Area(Lbl,kf) = sum(sum(Obj));
   end
end
plot(startframe:endframe,Area(displayLbl,:));
```

"code/Step4PlotCellArea.m"

Exercise 7.9

```
clear all;
close all;
startframe = 1;
displayLbl = [54 57];
BaseFolder = '...\';
fname = strcat(BaseFolder,'Tracking-obj1-10.tif');  %
     Exported stack (object)

%%%% Read the object label mask (same label = same
     cell over time)%%%%%%%
info = imfinfo(fname);
num_images = numel(info);
endframe = num_images;

% get information from last frame
A = imread(fname, endframe, 'Info', info);

% Find highest label: total number of cells
nbmaxcell = max(max(A));

% Initialize arrays
```

```
Area = nan(nbmaxcell,endframe);
X = nan(nbmaxcell,endframe);
Y = nan(nbmaxcell,endframe);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

for kf = startframe:endframe
   A = imread(fname, kf, 'Info', info);
   for Lbl=displayLbl
       Obj = (A==Lbl);
       Area(Lbl,kf) = sum(sum(Obj));
       stats = regionprops(Obj,'Centroid');
       stats = cat(1,stats.Centroid);
       if(~isempty(stats))
        X(Lbl,kf) = stats(1,1);
        Y(Lbl,kf) = stats(1,2);
       end
   end
end
plot(startframe:endframe,Area(displayLbl,:));
figure;
plot(X(displayLbl,:).',Y(displayLbl,:).');
```

"code/Step4PlotCellAreaTrack.m"

7.7 Appendix

In the alternative workflow "XAssignMacroSegmentationalternative.ijm," the cells are segmented by video inverting the fluorescence signal so that they appear as bright on a dark background. A step of background subtraction facilitates thresholding and then a binary watershed is applied with an intent to split the merged cells.

Acknowledgment

We thank Annalisa Letiza (IBMB-CSIC) for sharing the spinning disk movies. Sébastien Tosi collaborated with her to design the automated tissue analysis tools that she used in her own research.

References

1 Desprat, N., Supatto, W., Pouille, P.A., Beaurepaire, E., and Farge, E. (2008) Tissue deformation modulates twist expression to determine anterior midgut differentiation in *Drosophila* embryos. *Dev. Cell*, **15** (3), 470–477.

2 Petitjean, L., Reffay, M., Grasland-Mongrain, E., Poujade, M., Ladoux, B.,

Buguin, A., and Siberzan, P. (2010) Velocity fields in a collectively migrating epithelium. *Biophys. J.*, **98** (9), 1790–1800.

3 Farhadifar, R., Röper, J.C., Aiuluy, B., Eaton, S., and Jüllicher, F. (2007) The influence of cell mechanics, cell–cell interactions, and proliferation on epithelial packing. *Curr. Biol.*, **17** (24), 2095–2104.

4 Arganda-Carreras, I., Fernandez-Gonzalez, R., Munoz-Barrutia, A., and Ortiz-De-Solorzano., C. (2010) 3D reconstruction of histological sections: Application to mammary gland tissue. *Microsc. Res. Tech.*, **73** (11), 1019–1029.

5 Jaqaman, K., Loerke, D., Mettlen, M., Kuwata, H., Grinstein, S., Schmid, S.L., and Danuser, G. (2008) Robust single-particle tracking in live-cell time-lapse sequences. *Nat. Methods*, **5** (8), 695–702.

6 Sbalzarini, I.F. and Koumoutsakos, P. (2005) Feature point tracking and trajectory analysis for video imaging in cell biology. *J. Struct. Biol.*, **151** (2), 182–195.

8
Cell Polarity: Focal Adhesion and Actin Dynamics in Migrating Cells

Perrine Paul-Gilloteaux[1,2] and Christoph Möhl[3]

[1]Institut Curie, Centre de Recherche, Paris 75248, France
[2]Cell and Tissue Imaging Facility, PICT-IBiSA, CNRS, UMR 144, Paris 75248, France
[3]German Center of Neurodegenerative Diseases (DZNE), Image and Data Analysis Facility (IDAF), Core Facilities, Holbeinstraße 13–15, 53175 Bonn, Germany

8.1
Aim

In this chapter, we learn to identify image objects in time-lapse movies of single migrating cells. We analyze their shape, growth dynamics, and movement of the associated actin network, which is recorded in a separate channel. This way, we get familiar with optical flow analysis and learn how to integrate the information of different image data sources to discover relationships between different object features.

8.2
Introduction

Effective locomotion of migrating cells depends on the coordinated interplay between protrusive, contractile, and adhesive components of the cytoskeleton and the plasma membrane. Traction forces are generated by a viscoelastic network of actin filaments interacting with myosin II motors. These forces are applied to the substrate through distinct focal adhesions (FAs). These are protein clusters at the cell plasma membrane coupling actin filaments to extracellular matrix proteins.

Forward movement is enabled by polarized growth dynamics of FAs (see Figure 8.1): At the cell front, new FAs are assembled, while FA disassembly mainly occurs at the cell's rear and center. The coupling of actin to FAs, and hence the transduction of tractile forces to the substrate, may also vary between different types of FAs. This coupling can be measured indirectly by the movement of actin above FAs. While tight coupling is indicated by slow actin flow

Bioimage Data Analysis, First Edition. Edited by Kota Miura.

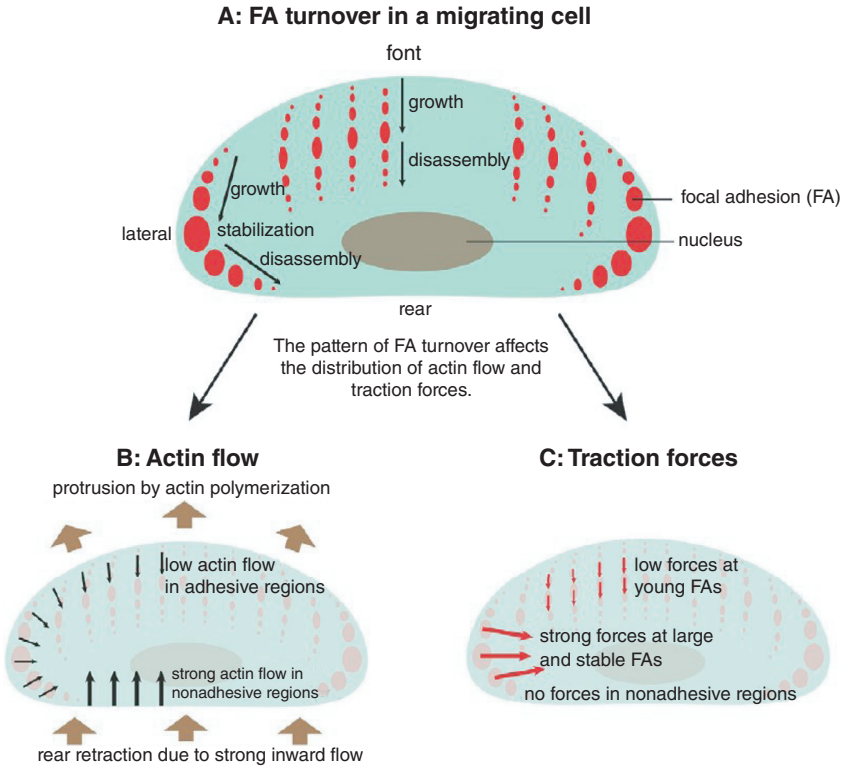


Figure 8.1 The interplay of focal adhesion and actin dynamics as well as resulting traction forces in a migrating cell.

due to higher friction, weak substrate coupling correlates with fast actin movement [1,2].

8.2.1 Questions to Solve

Identify FAs in time-lapse movies of migrating cells and measure their size, growth rate, and the actin flow above them.

- What is the relationship between growth rate, actin flow, and size?
- Do growing FAs (positive growth rate) exhibit a stronger or weaker actin coupling compared with disassembling FAs (negative growth rate)?

8.2.2 Data Set

We analyze four time series with one migrating cell in each (human keratinocyte on fibronectin-coated glass substrate). The cells express the fluorescently labeled

FA protein vinculin (label: dsRed) and actin (label: GFP). Each time series consists of two movies recorded one after the other:

1) Red epifluorescence: 6 min time lapse of focal adhesion dynamics at low temporal frequency (2 frames/min; movies suffixed with `_1`, called movie 1).
2) Green TIRF[1]: 80 s actin dynamics at high temporal frequency (15 frames/min; movies suffixed with `_2`, called movie 2).

Due to the high sampling rate for actin flow quantification, FA and actin dynamics were not recorded simultaneously.

8.2.3 Overview of Data Processing

- Step 1: Segmentation of movie 1 to identify FA objects (Fiji).
- Step 2: Quantification of actin flow above FA objects in movie 2 (Matlab).
- Step 3: Calculation of features of FA objects: area, growth rate, and mean actin flow (Matlab).
- Step 4: Statistical analysis of the relationship between FA area, growth rate, and actin flow (Matlab).

We use Fiji[2] for FA segmentation and Matlab to quantify the actin flow and calculate features of the detected FA objects (from the analysis in Fiji), for statistical analysis and plotting of results. In addition to the main distribution of Matlab, we will use the image processing toolbox (for labeling operations and displaying images).

8.3 Step 1: Identification of Focal Adhesions

8.3.1 Workflow

We first develop a routine in ImageJ Macro language to segment FAs over time in time-lapse movie 1.

This routine should execute the following tasks:

- Import raw data in native microscopy format: movie suffixed _1 in zvi format.

```
(1_1.zvi, 2_1.zvi, 3_1.zvi, 4_1.zvi)
```

1) Total internal reflection microscopy: A method to image only fluorescent signals in close proximity to the cover slip. It is used here to image actin flow only in regions of cell adhesion, where the cell membrane is adjacent to the glass. Another advantage is the exclusion of cytoplasmic background signal.
2) ImageJ Version 1.49p.

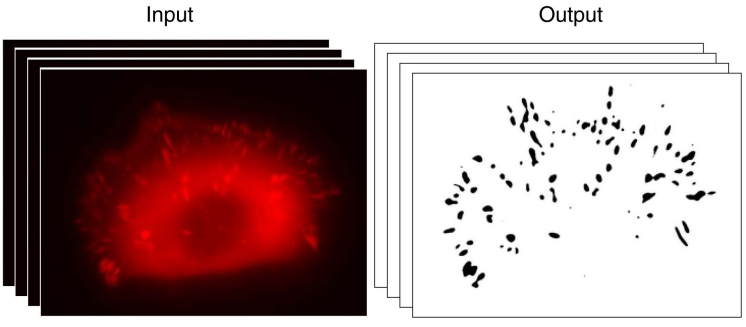


Figure 8.2 Input and output of the ImageJ Macro.

- Automated detection of FA regions: A black and white mask image of the segmented objects is created for each frame.
- Data export: The masks are exported in TIF format (see Figure 8.2): The TIF stacks can later be imported in Matlab for feature processing and integration with the actin flow data.

To automate the processing of the image time series, we write a script in ImageJ Macro language by recording the individual steps.

1) Open the recorder with `[Plugins > Macros > Record ...]`.
2) Use the LOCI importer to load image data in zvi format `[Plugins > LOCI > Bio-Formats Importer]`.
3) If you have a look through the whole time series, you will notice that the sample bleaches out over time. Since we want to detect FA objects by an intensity threshold, we have to correct for the bleaching. Otherwise, the objects tend to be smaller (as they become darker) toward the end of the movie. We correct for bleaching by applying `[Image > Adjust > Bleach Correction]`. You can choose several options for bleach correction. By visual inspection, it turns out that the "histogram matching" method works best. Here, the shape of the gray value distribution for each frame is adapted to the first frame.[3)]
4) Smooth raw images with a Gaussian filter over x, y, and time `[Process > Filters >Gaussian Blur 3D...]` The smoothing removes the image noise and will result in smoother outlines of the FAs we want to detect later. Sigma corresponds to the radius of the filter window. Here the third dimension is the time, but it is called z in the command. Suggested parameters for sigma are $x = 4$, $y = 4$, and $z = 6$.

3) More information about the different bleaching correction methods implemented in Fiji can be found at fiji.sc/Bleach_Correction.

5) As images were collected in epifluorescence, cytoplasmic background signal is very strong.[4] We remove the background by applying the "Subtract Background" function `[Process > Subtract Background...]`. This function executes two steps: It first applies a large average filter, resulting in a strongly blurred image. If the filter window is significantly larger than the objects of interest, the blurred image is a good approximation of the cytoplasmic background (suggested value for the radius is 17). In the second step, this background image is subtracted from the original image resulting in the background corrected image. The resulting stack will be referred to as "corrIm".

```
1   //bleach correction
2   run("Bleach Correction", "correction=[Histogram Matching]");
3      corrIm=getImageID();
4   //filtering++++++++++++++++++++++++++++++++++++++++++++
5   //smooth with gaaussian
6   run("Gaussian Blur 3D...", "x=4 y=4 z=6");
7   //subtract background
8   run("Subtract Background...", "rolling=17 stack");
9   run ("Enhance Contrast","saturated=0.1");
```

code/Step1_FA_segmentation.ijm

6) Up to now, we corrected our time series for bleaching, noise, and cytoplasmic background and we are ready to detect FA objects by simple intensity thresholding. We can test several algorithms for calculating an automatic threshold by executing `[Image > Adjust > Auto Threshold]` and setting "Method" to "Try all". It will take a while to calculate all different thresholds. You can follow the progress in the log window. Finally, an image montage is displayed showing binary masks for all the different algorithms. This montage allows us to quickly pick an algorithm where the threshold is suitable. In our case, the "IsoData" method does a good job in detecting the FA objects. We run again `[Image > Adjust > Auto Threshold]` with the option "IsoData". We check also the options "Stack" and "Use Stack Histogram". With the "Stack" option checked, masks will be generated for the whole time series. The "Use Stack Histogram" option ensures that one global threshold is calculated for the whole time series, and not a new threshold for each frame. Since we corrected for bleaching in the first step, we do not have to adapt the threshold over time.

4) In epifluorescence, the optical section is not restricted in *z* (which is the case in confocal, light sheet, or TIRF microscopy). Due to the wide optical plane, a lot of fluorescent signals from cytoplasmic vinculin-GFP are imaged.

```
    //segmentation
        ++++++++++++++++++++++++++++++++++++++++++++
    run("Auto Threshold", "method=IsoData white stack
        use_stack_histogram");
    //+++++++++cleaning++++++++++++++++++++++++++++++++++
    setSlice(1);
    //be careful about your settings for binary (black
        background)
    run("Convert to Mask","stack");
    run("Fill Holes", "stack");
    //remove too small and too large particles
    run("Analyze Particles...","size="+minsize+"-"+maxsize+"
        circularity=0.00-1.00 show=Masks exclude stack");
    wait(waittime);
    saveAs("Tiff", dirresults+prefix+"_FA.tif");
    run("Close All");
    } //closing if
}//closing loop on images in directory
IJ.log("Done");
```

code/Step1_FA_segmentation.ijm

The "Auto Threshold" function will produce a mask image where FAs appear in black and background in white. You may notice that some detected objects are too small, or some objects have holes. Holes can be simply closed by a morphological operation: `[Process > Binary > Fill Holes]`.

Subsequently, we exclude too large and too small segments with the `[Analyze > Analyze Particles...]` function with the "Show Masks" option checked. We define the area range of the detected particles by setting variables for minimal and maximal sizes and feed them to the "Analyze Particles" command:

```
minsize=0.06;//minimal size of focal adhesions
maxsize=25;//maximal size of focal adhesions
```

code/Step1_FA_segmentation.ijm

```
 //remove too small and too large particles
 run("Analyze Particles...","size="+ minsize +"-"+maxsize+"
     circularity=0.00-1.00 show=Masks exclude stack");
```

code/Step1_FA_segmentation.ijm

7) Finally, we export the mask stack as a single TIF file. `dirresults` (folder where results are saved) and `prefix` (part of the file name) can be either hard coded or interactively asked from users with a dialog window.

```
    saveAs("Tiff", dirresults+prefix + "_FA.tif");
```

code/Step1_FA_segmentation.ijm

8.3.2 Summary of Tools Used in Step 1

- *LOCI Bio-Formats reader* `[Plugins > LOCI > Bio-Formats Importer]`, bundled in Fiji, is very well maintained and supports most of the microscopy formats. Further information is available at fiji.sc/Bio-Formats.
- *Bleaching Correction* `[Image > Adjust > Bleach Correction]`: We use the plug-in for keeping the gray value distribution stable over the whole time series. This simplifies subsequent intensity thresholding, because it allows us to threshold with one fixed value.
- *Background Subtraction* `[Process > Subtract Background...]`: This command first computes a local background image by applying a large average filter and subtracts this background from the original. The critical step is to get a reasonable approximation of the local background. The filter window should be very large (more than twice the size of your objects of interest). Otherwise, the foreground objects are "interpreted" as background. On the other hand, if the window is too large, local background intensity changes are not reflected anymore. The best way to optimize the filter settings is to display the background image and compare with the original (check "create background" in the options).
- *Auto Threshold* `[Image > Adjust > Auto Threshold]`: The function comprises a set of different algorithms to compute a global intensity threshold. Thresholds can be calculated for each frame individually or for the whole time series. *Hint*: By using the command `[Image > Adjust > Auto Local Threshold]`, so-called local thresholds can be computed, also with a set of different methods. With local thresholding, a threshold is calculated for each pixel of an image individually, depending on the local neighborhood of this pixel. This might be useful if you have to deal with different local background intensities; that is, a higher threshold is appropriate in regions of higher background signal.

In our example, we corrected the image both for bleaching and for local background variation beforehand. Thus, we were able to apply a relatively simple global threshold in the end.

8.4 Step 2: Quantification of Actin Flow

Now we want to quantify actin flow in time-lapse movie 2. Open one of the movies with suffix 2 in Fiji (e.g. movie `3_2`) and look for actin dynamics: adjust the contrast and try to identify the flow visually. Could it be tracked using a particle tracking algorithm? The answer is no because we cannot identify single objects as we did for the microtubule tracking in Chapter 6.

Optical flow analysis is generally advised when density, the lack of prominent features, and their complex motion prevent the individual extraction of objects of interest. Here, we are interested in measuring the actin flow. Several methods

already exist to estimate flow: some are based on intensity conservation equation (Horn and Shunk), and others on block matching techniques[5] based on correlation. Here we choose to use Matlab for better understanding of the steps involved in measuring a flow and because Matlab provides native functions for correlation computationally efficiently. Alternate solutions in Fiji are described at the end of this section.

To import the images to Matlab, we first convert them to 8 bit TIF files with Fiji. You can use the batch convert command (under process) for that or write a short macro.

A Matlab script for step 2, processing all the movies and saving the results for Matlab, is provided to you.

Data will be accessed by indicating the path[6] of the directory containing data and results, relatively to the current directory (. . . allow to move one branch above), and then concatenating the file name with this path to open it:

```
1 path='../data'; % where all data and results have been saved
```

code/Step2_ComputeFlowinMatlab.m

```
1   % Load the last frame content into a 2D matrix MatrixframeFA
2   MatrixframeFA=imread([path,'\',filenameFAmask],
       NbframesFA); figure(1), imshow(MatrixframeFA,[]);
```

code/Step2_ComputeFlowinMatlab.m

8.4.1 Workflow

We start by describing the full workflow, and propose an exercise focusing on a couple of frames and one block at the end of the section.

Execute the following workflow, described in "Step2_ComputeFlowinMatlab.m", to one of the movies 2 (e.g. movie `3_2`). The purpose of this script is to compute the flow by cross-correlation, but only on the regions of interest that are the focal adhesions in the last frame of the exported mask. (*Reminder*: Movies 2 were acquired sequentially after movies 1, so we want to analyze only the FAs of the last frame of movie `3_FA.tif`.)

- Read the images: Matlab does not support the native microscopy format. A Matlab version of the LOCI Bio-Formats reader is available, but for sake of simplicity, we have converted the actin movies as 8 bit TIF files with Fiji (see

5) Two image regions (image blocks) are compared. In time-lapse data, a small region in frame t is compared with a shifted region of same size in frame $t + 1$. If the two regions are highly similar, their shift is considered as optic flow.

6) Use of path can be misleading: here we assume the following directory tree: code directory contains your Matlab code and will then become the current path for Matlab, and all your data and results are saved under the directory data, at the same level as code.

the Batch Macro used above) and saved them in the "data" folder. Matlab can get information about the number of frames or the width or height of an image using the command `imfinfo`.

First we read the FA mask that we obtained in step 1:

```
1
2  filenameFAmask='3_FA.tif';
3  info=imfinfo([path,'\',filenameFAmask]); %
4  NbframesFA=length(info);
5  % Load the last frame content into a 2D matrix MatrixframeFA
6  MatrixframeFA=imread([path,'\',filenameFAmask],
      NbframesFA);
7  figure(1), imshow(MatrixframeFA,[]);
8  % Display it in figure 1, Note that MatrixframeFA content is 255
      for background, and 0 for focal
9  % adhesions
```

code/Step2_ComputeFlowinMatlab.m

Read the first frame of flow movies:

```
1  filenameFlow='3_2.tif'; % REMINDER: this file has been
      obtained through conversion of zvi _2 files to
      8 bits tiff from ImageJ.
2  info=imfinfo([path,'\',filenameFlow]); %
3  NbframesFlow=length(info);
4
5
6
7  %% Compute the flow above FA only
8  % Read the first image
9  % Flow will be compute between pair of images, i.e.
      (1 and 2, 2 and 3, ...)
10 Matrixframe2=imread([path,'\',filenameFlow],1);
```

code/Step2_ComputeFlowinMatlab.m

- We then loop through the whole time series, where we load two successive frames in memory and then compute simple normalized cross-correlation of subblocks of the image ONLY if
  - the area is not too uniform (i.e., block standard deviation >10% of the standard deviation of the full image): indeed in that case the correlation will not produce a clear peak;
  - the block contains at least one focal adhesion (which is checked using the last frame of movie `3_FA.tif`).
- The peak of the correlation map will indicate the lag of position leading to the maximum similarity; the position of the peak relative to map center will be used as the flow vector coordinates.

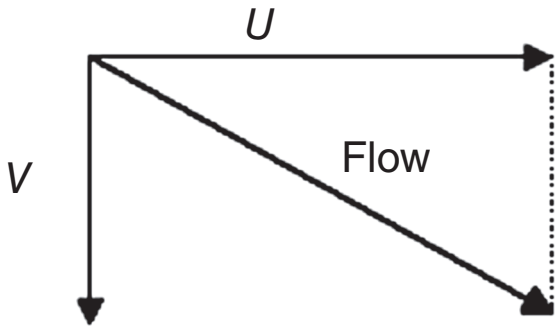


Figure 8.3 Output of PIV is the *x* and *y* components of the flow vector. Its norm can be computed as $\sqrt{\|U\|^2 + \|V\|^2}$.

- We compute the average flow over time for each block and save it with its position in a .mat file (which allows to save variable in the workspace).

Exercise: In order to better understand what is happening, we focus on one block only: use the Matlab script `Exercise_Step2_ComputeFlowinMatlab.m`. Several parameters need to be tuned for estimating the flow by cross-correlation (see Figures 8.3 and 8.4):

```
1 path='../data'; % where all data and results have been saved
2
```

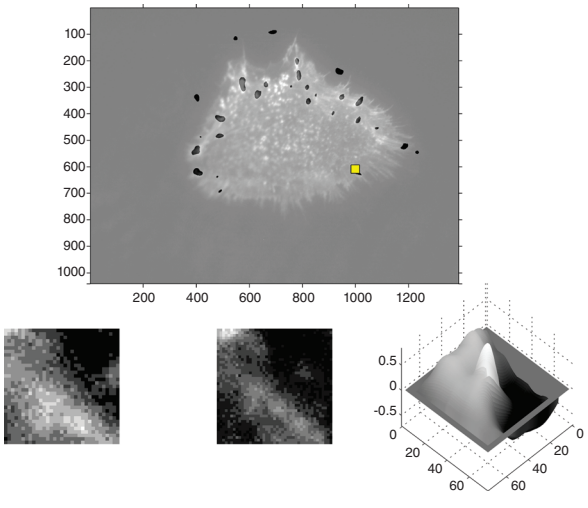


Figure 8.4 Example of FAs identified in movie 1, superimposed on a frame of movie 2. The processed block for correlation analysis is indicated by a yellow rectangle. Below you see a zoomed view of this region for frames 1 and 2 (blocks 1 and 2), and the corresponding correlation map. The correlation map is generated by shifting the two blocks against each other and computing the correlation for each shift. The flow vector is computed by finding the position of this peak (i.e., the shift where the correlation of blocks 1 and 2 is maximal).

```
%% Set the parameters for the PIV (normalized
   correlation based)
    Correlation_windowsize=16; % block size for normalized
       correlation
    Correlation_overlap=8; % the correlation will be computed
       every Correlation_overlap pixels
    Correlation_Max_search_area=5; % To add if the expected
       displacement is bigger than the block size
    Correlation_threshold=0.3; % only score above this value will
       be kept to measure the flow
    %(max theoretical value =1.0 for two identical images)
    maxspeed=5; % Maximum velocity allowed (in pixel
       per frame): this parameter will allow to crop
    %the correlation map between two blocks to avoid false peaks.
```

code/Step2_ComputeFlowinMatlab.m

Try different parameters for the window size (8, 16, 32, and 64): comment on both accuracy of results and speed of computation.

8.4.2 Summary of Tools Used in Step 2 and Alternative Tools

`quiver` is a native Matlab function to smooth and plot vector data as arrows (see example in Figure 8.5). `norxcorr2` is a native Matlab function allowing to compute correlation (similarity) map. The position of its maximum can be found using the `max` Matlab function returning both the position and the value of the maximum.

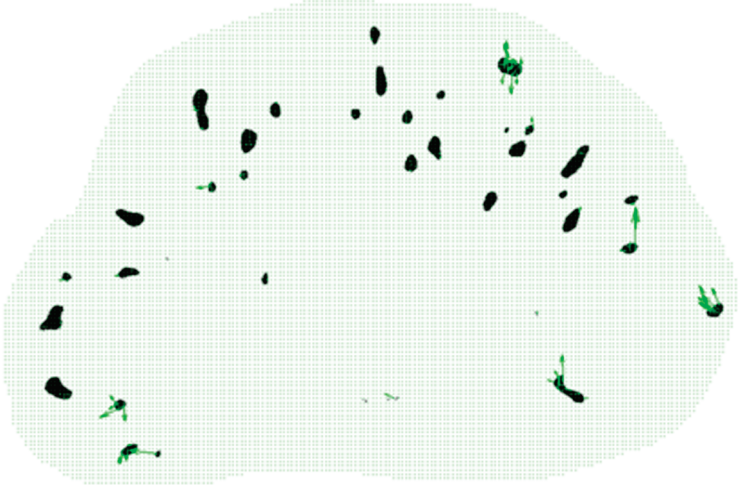


Figure 8.5 Output of the quiver visualization of flow, with scale parameter at 10 (norm multiplied by 10).

`save` allows to back up a selection of variables from the workspace (here used to save the position of the flow computed). To read back the data, the corresponding function is `load`.

Alternative tools: Several plug ins for optical flow estimation are bundled with Fiji.

- *FlowJ* (`Analyze > Optic Flow > FlowJ`): It is more complete since different methods are implemented, including the one based on intensity conservation equation. However, it is more demanding to use since you have to know the meaning of the several parameters to tweak (http://webscreen.ophth.uiowa.edu/bij/flowj.htm).
- *PIV analysis* (accessible from `Analyze > Optic Flow > PIV analysis`). This command implements the block matching correlation technique that we have done in Matlab in this step. This is one of the simplest existing methods of optical flow estimation. This plug-in has the advantage of a simple user interface and convenient output but needs long computation time.
- Mpicbg plug-ins implemented by Stephan Saalfeld: `Plugins > Optic Flow`.
  - *PMCC block flow* looks for the block that gives the maximum correlation score, as in PIV analysis, but with a different implementation.
  - *MSE block flow* looks for the matching most similar pixel in a search radius based on the minimal sum difference (not anymore correlation) between two blocks.
  - *MSE Gaussian flow* looks for the matching most similar pixel in search radius based on the minimal sum difference between a Gaussian neighborhood: block is not square anymore, it has a Gaussian shape.

8.5 Step 3: Calculation of Focal Adhesion Features

Now we want to calculate the following FA features and associate with each FA: area, growth rate, and actin flow speed. We will compute these features for each FA, and create a Matlab "structure array" to store it, that is, a structure for each FA containing these features, organized as an array of FA features. The solution processing all the movies is provided in `Solution_Step3_GetFeaturesbyFocalAdhesions.m`.

8.5.1 Workflow

8.5.1.1 Import of Focal Adhesion Masks to Matlab

First, we build a for-loop to import the mask images slice by slice with the `imread` function (see Figure 8.6).

We convert the gray value intensities (format: uint8 integer) into Booleans with a logical operation:

```
m3b=(m3==0).
```

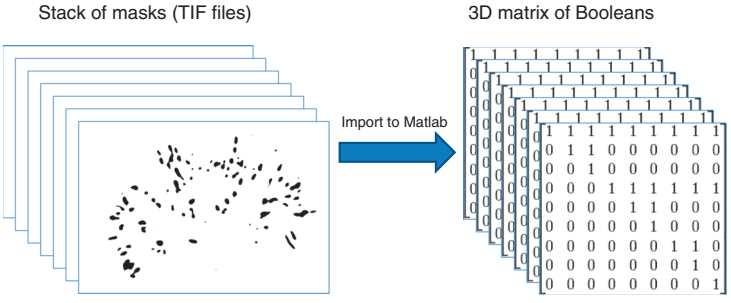


Figure 8.6 Import of mask images to Matlab.

Now, each FA pixel is marked with 1 (TRUE) and each background pixel is marked with 0 (FALSE).

8.5.1.2 Identify Objects

At the moment, the mask matrix just gives us the information regarding which pixel belongs to foreground (FA region, value 1) and which pixel belongs to background region (value 0). In the next step, we assign all foreground pixels to a certain FA object with a simple rule: All foreground pixels touching each other in space (*xy*-direction) and time (*z*-direction) belong to the same FA.

You can use the function `bwlabeln` as illustrated in Figure 8.7. `L = bwlabeln (m3b)` returns a label matrix, *L*, containing labels for the connected components in m3b (see Figure 8.7). The input image can have any dimension; *L* is the same size as m3b. The elements of *L* are integer values greater than or equal to 0. The pixels labeled 0 are the background. The pixels labeled 1 make up one object; the pixels labeled 2 make up a second object; and so on.

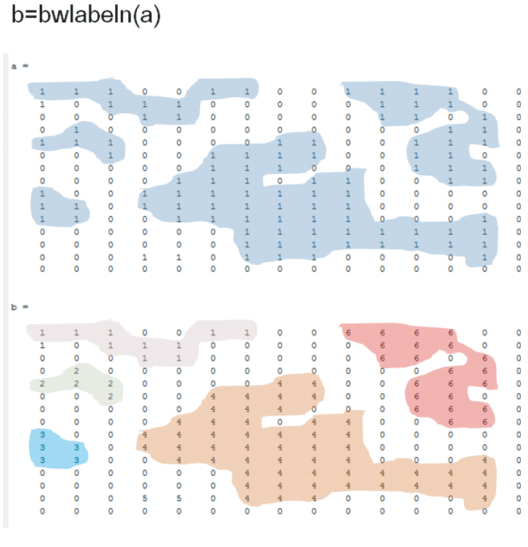


Figure 8.7 Identification of connected components with the Matlab function bwlabeln.

Now we have identified individual FAs. Each FA object is defined by its unique label number. Before we continue with calculating features of our objects, we have to define our FAs of interest: We only want to analyze FAs that are present in the last time frame. Why? In our experiment, actin flow was measured after collecting the FA time series. Thus, actin flow information is only available for FAs that are present in the last time frame. But why did we segment not only the last FA image, but the whole time series? Because we need information from the other frames of the time series to calculate the growth rate for our FAs of interest (this will be done later, see Section 8.5.1.3).

To identify our FAs of interest, you can apply the `unique` function to the last frame. Study the Matlab help to get more information about `unique`.

As a result, you should have an array of numbers (named `selected`) containing the labels of all FAs that are present in the last frame.

8.5.1.3 Calculate Object Features

The next step is a bit tricky. We have our 3D label matrix (`L`), the actin flow velocity image (`flow`), and the array with our FAs of interest (`selected`). Based on these data, we want first to calculate the area in the last frame for each FA of interest, and also assign a unique number to each FA as a feature (label).

The area feature calculation is done with the `regionprops` function. We feed `regionprops` with the last frame of the label matrix. The calculation of object area is straightforward (read the Matlab help for further information and have a look on the code snippet below).

If implemented correctly, regionprops should give an array of structs (named here `stat`) with fields `Area`.

```
stat=
43x1 struct array with fields:
    Area
```

To get rid of all FAs that are not present in the last frame, we use the information from the selected array and delete respective elements in the stat array:

```
stat=stat(selected)
```

Finally, we add second and third features, the FA label number and the movie number, to the `stat` array. Import flow data saved in step 2 with the `load` function.

This process of feature calculation as described above can be implemented in a few lines of Matlab code:

```
1     for i=1:length(stat)
2         stat(i).label=selectedFA(i);%add FA label
              number (tag) as feature
3         stat(i).datnumber=dat;% image number as feature
```

code/Solution_Step3_GetFeaturesbyFocalAdhesions.m

Next we want to compute the FA growth rate. Since the growth rate is the change of area over time, we first have to calculate FA areas for a certain time window. In the example below, a vector named `listareas` will collect the FA

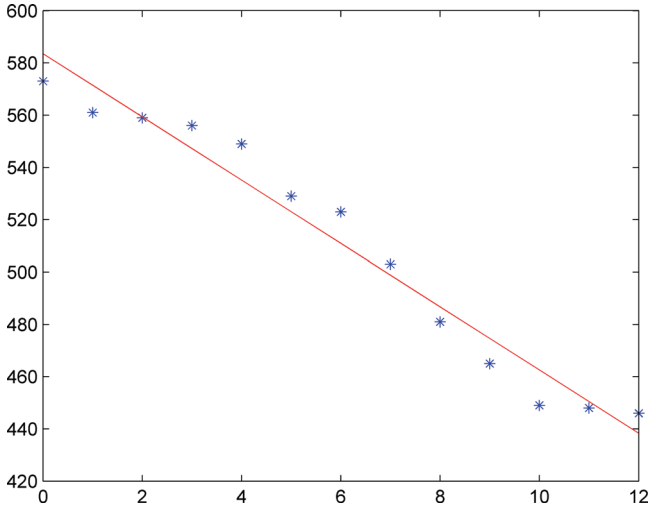


Figure 8.8 Example of plot of area against time for one particular focal adhesion, in this case shrinking.

areas of the last 13 frames (d*t*). Note that the last value is identical to the feature `Area` (both are the FA area in the last frame):

```
>> stat(1)
ans=
Area: 282 label: 1 datnumber: 2
>>listareas
listareas =
338 290  280  261  274  269  254  262  271  265  268  282  282
```

For Matlab experts: Try to calculate the `listareas` vector on your own. Solution is given in lines 64–68 of the step 3 solution.

Now at each iteration of the loop, we have area profiles over time for each FA. From the area profiles, you can immediately see whether an FA is growing (increase of area over time) or shrinking (decrease of area over time). The area profile of Figure 8.8 shows a shrinking FA.

The growth rate is the change of area over time. This can be calculated, for example, by linear regression of the area profile. You can easily calculate the growth rate (= slope of the area profile) with the following lines:

```
1      for i=1:length(stat)
2          stat(i).label=selectedFA(i);%add FA label
               number (tag) as feature
3          stat(i).datnumber=dat;% image number as feature
4          %collect vector of areas for dt time points before
               the last frames
5          %as a feature, say 13 for example
```

```
6          %the growth rate is calculated by linear
               regression of area values over
7          %time
8         dt=13;
9          for j=1:dt
10             isregion=labelsFA(:,:,j+(end-dt))==stat(i).label;
11             listareas(j)= sum(isregion(:));
12
13         end
14         listareas(listareas==0)= nan; % FAs may be appearing
               and then have
15         %no area (area 0) for the first time points. We
               do not want to fit
16         %it so we put its values to nan so it will be ignored.
17         x=(0:length(listareas)-1); % create a reference abscissa
               for the fit
18         x=[ones(size(x));x];
19         % to compute the linear regression in this way,
               we need an additional line at 1
20         p=listareas/x;
21        figure(1) % for visual check
22        plot(x(2,:),listareas,'*'); hold on;
23         plot(x(2,:),p(1)+p(2)*x(2,:),'-r'); hold off;
24         % p is a vector such that p(1)+p(2)*x=listareas
25         stat(i).growthrate=p(2);
26     end
```

code/Solution_Step3_GetFeaturesbyFocalAdhesions.m

Finally, we have all our features inside the struct array:

```
stat=
    49x1 struct array with fields:
    Area label datnumber growthrate
    stat(1)
    ans=
    Area: 503 label: 3  datnumber: 1 growthrate: 14.0659
```

The last feature we want to add to each FA object is the average flow over time and for the whole FA. In step 2, we have saved in a .mat file per movie the following vector variables: `x_pos` and `y-pos`, which are coordinates on one frame, and `flow`, which is the average flow over time for this position. We start by loading these variables in the workspace:

```
1      %% Import actin flow data
2      % Calculation of mean actin flow above FA objects to add it as a
3      % feature
4      filenameFlowVar=[datatoprocess,'_flow.mat'];
```

```
% this will load the variables flow, x_pos and y_pos for this
    dataset
% datatoprocess (prefix from 1 to 4)
load([path,'/',filenameFlowVar]);
```

code/Solution_Step3_GetFeaturesbyFocalAdhesions.m

Then for each FA, we add the flow information by computing the average of flow when the pixel in the last frame of the matrix labelsFA at position x_pos and y_pos returns the label number of the current FA processed. Some of the values of flow may have the value `nan`, because the flow was not computed (too uniform or correlation score of the peak above the correlation threshold allowed in step 2). We use the Matlab function `nanmean`, which will ignore nan values for computing the mean. (*Note*: Some nanmean function is provided in the code directory in case you do not have access to the Statistics toolbox.)

```
for i=1:length(stat)

    for p=1:length(flow)

      % x_pos, y_pos and flow comes from step2, and are vector with the
      % flow magnitude for sparse points on FAs.
      % get the list of flow measurement for one particular FA:
      %collect flow value for this FA:
      f=1;
      if labelsFA(y_pos(p),x_pos(p),end)==stat(i).label;
        flowlist(f)=flow(p);
      end
    end
    stat(i).averageflow=nanmean(flowlist); % nanmean STATISTICS
       TOOLBOX here a version is provided in case Statistics
       toolbox is not available.

end
```

code/Solution_Step3_GetFeaturesbyFocalAdhesions.m

We save this array of structure:

```
save([path,'/data_',num2str(dat)],'stat')
```

code/Solution_Step3_GetFeaturesbyFocalAdhesions.m

8.5.2 Summary of Tools Used in Step 3

Matlab proposes different ways to perform linear regression, or linear fitting. We used the one that was explained in module 2 (`c=A/b`). Matlab functions in other

toolboxes (statistics or curve fitting) will also do the job, such as `polyfit`, `regress`, or `robustfit`.

8.6 Step 4: Statistical Analysis

Next, we want to find out whether there are interesting relationships between the features. In particular, the questions we had were the following:

- What is the relationship between growth rate, actin flow, and size?
- Do growing FAs (positive growth rate) exhibit a stronger or weaker actin coupling compared with disassembling FAs (negative growth rate)?

We first collect the data for all movies in one big structure array:

```
path='../data';
allstat=[];% initialisation in order to append all processed
   files in one
for i=1:4
load([path,'/data_',num2str(i)])
length(stat)
allstat=[allstat;stat];
end
```

code/Solution_Step4_StatisticalAnalysis.m

We create three vectors (area, flow, and growth rate) that contain in line *i* the feature for FA *i* and will be easier to manipulate.

```
for i=1:length(allstat)

  area(i)=allstat(i).Area;
  flow(i)=allstat(i).averageflow;
  growthrate(i)=allstat(i).growthrate;
end
```

code/Solution_Step4_StatisticalAnalysis.m

We can then plot for each FA its position in area, flow, and growth rate space:

```
 plot3(growthrate,area,flow,'o'); hold on;

xlabel('Area growth (slope)'),
ylabel('Area in last time point of movie 1');
zlabel('Average flow computed on movie 2');
```

code/Solution_Step4_StatisticalAnalysis.m

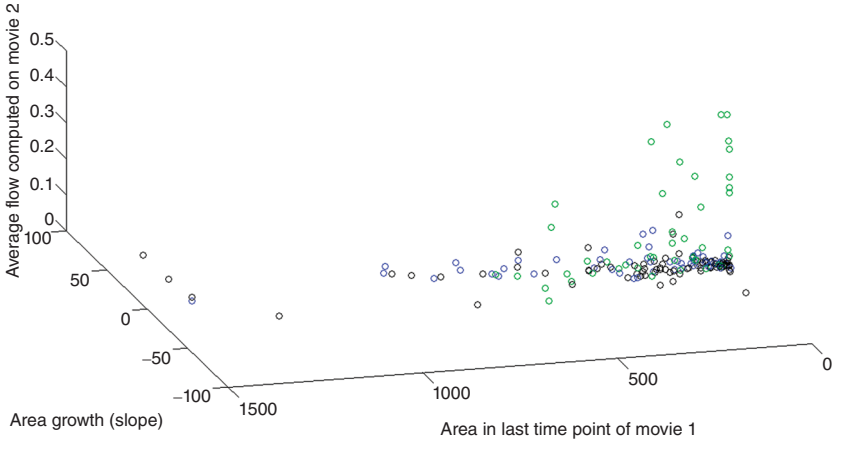


Figure 8.9 Plot of area versus area growth versus average speed. Each color corresponds to one cell.

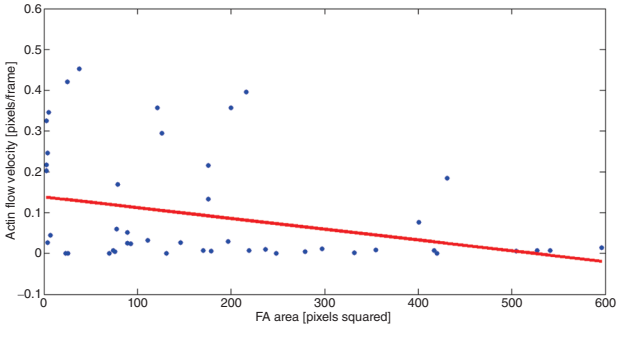


Figure 8.10 Plot of flow average speed in FA versus area for one cell. In red, its linear regression, with a score of 0.12 (not high). More data may be needed to see a relationship.

Figure 8.9 shows the result of a plot of the three main criteria we want to study. From a visual check, it appears that indeed it seems to have an inverse relationship between area and average flow. To check this, we can perform a linear regression on these two measurements (area and flow), as we did previously in step 3 to obtain the growth rate (Figure 8.10). This time, we want to check whether there is a linear relationship between these two features.

```
1 area=area(~isnan(flow));% we create a subset of area vector
     which will
2 %contain only area of FAs for which the flow was a number
     (isnan  will return
```

```
%1 for each line containing a nan, 0 otherwise. ~isnan actually do the
%contrary ~means NOT in matlab scripting language.
%growthrate=growthrate((~isnan(flow)));
flow=flow(~isnan(flow));
area=area(flow>0);
%growthrate=growthrate(flow>0);
flow=flow(flow>0);


A=flow;
B=[ones(size(area));area];
p=A/B;

flowfit=p(1)+p(2)*area;
figure,

 plot(area,flow,'.')
   xlabel('FA area [pixels squared]')
   ylabel('actin flow velocity [pixels/frame]')
hold on,
plot(area,flowfit,'r-');
flowresid=flow - flowfit;
%SSresid is the sum
%of the squared residuals from the regression.
SSresid=sum(flowresid.^2);
%SStotal is
%the sum of the squared differences from the mean of the dependent
%variable (flow) (total sum of squares).
SStotal=sum((flow-mean(flow)).^2);
%This statistic indicates how closely values you obtain
%from fitting a model match the dependent variable the model is intended
%to predict. (should be 1 for perfect prediction)
rsq=1 - SSresid/SStotal;
text(1000,3,['R=', num2str(rsq)]);
```

code/Solution_Step4_StatisticalAnalysis.m

R^2 will give the score of the prediction of flow by area using a simple linear regression. The prediction power of flow by area is very low, but would encourage to get more data since some inverse relationship coherent with the biology behind may be seen.

Exercise: Now we want to separate assembling and disassembling FAs in this analysis. Perform the same fitting by adding two lines before the previous code in order to analyze growing FAs (i.e., growth rate > 0). Is the relationship stronger or weaker?

8.7 Solutions

Full macros and Matlab code are available in the code directory:

- *Step1_FA_segmentation.ijm*: Focal adhesion segmentation (ImageJ Macro).
- *Step2_ComputeFlowinMatlab.ijm*: Actin flow computation (Matlab)
- *Step3_GetFeaturesbyFocalAdhesions.m*: Calculation of focal adhesion features (Matlab).
- *Step4_StatisticalAnalysis.m*: Statistical analysis (Matlab).

References

1 Hu, K., Ji, L., Applegate, K.T., Danuser, G., and Waterman-Storer, C.M. (2007) Differential transmission of actin motion within focal adhesions. *Science*, **315** (5808), 111–115.

2 Möhl, C., Kirchgessner, N., Schäfer, C., Hoffmann, B., and Merkel, R. (2012) Quantitative mapping of averaged focal adhesion dynamics in migrating cells by shape normalization. *J. Cell Sci.*, **125** (1), 155–165.

9
Tumor Blood Vessels: 3D Tubular Network Analysis

Christian Tischer[1] *and Sébastien Tosi*[2]

[1]*Advanced Light Microscopy Facility, EMBL Heidelberg, Meyerhofstr.1, D-69117, Heidelberg, Germany*
[2]*Institute for Research in Biomedicine - IRB Barcelona, Advanced Digital Microscopy, Baldiri Reixac, 10, E-08028, Barcelona, Spain*

9.1
Overview

9.1.1
Aim

In this module, we will implement a simple ImageJ macro to segment and analyze the blood vessel network of a subcutaneous tumor (see Figure 9.1). The analysis is fully performed in 3D, and possible strategies to extract statistics of the network geometry and interactively visualize the results are also discussed and implemented.

9.1.2
Introduction

Segmenting and extracting the geometry of the blood vessel network inside specific subregions of a tumor is a powerful investigation tool: The density of the vascularization and vessel branching points and the thickness of the vessels are for instance crucial age indicators to understand how the structure developed and possibly necrosed. With the help of a simple ImageJ macro these statistics can be extracted and the network 3D rendered with judicious color/transparency to provide insights on its organization.

9.1.3
Data Sets

The blood vessel data sets were acquired by a custom made (IRB Barcelona) macroSPIM allowing to image large (up to 1 cm), fixed, and optically cleared samples

Bioimage Data Analysis, First Edition. Edited by Kota Miura.

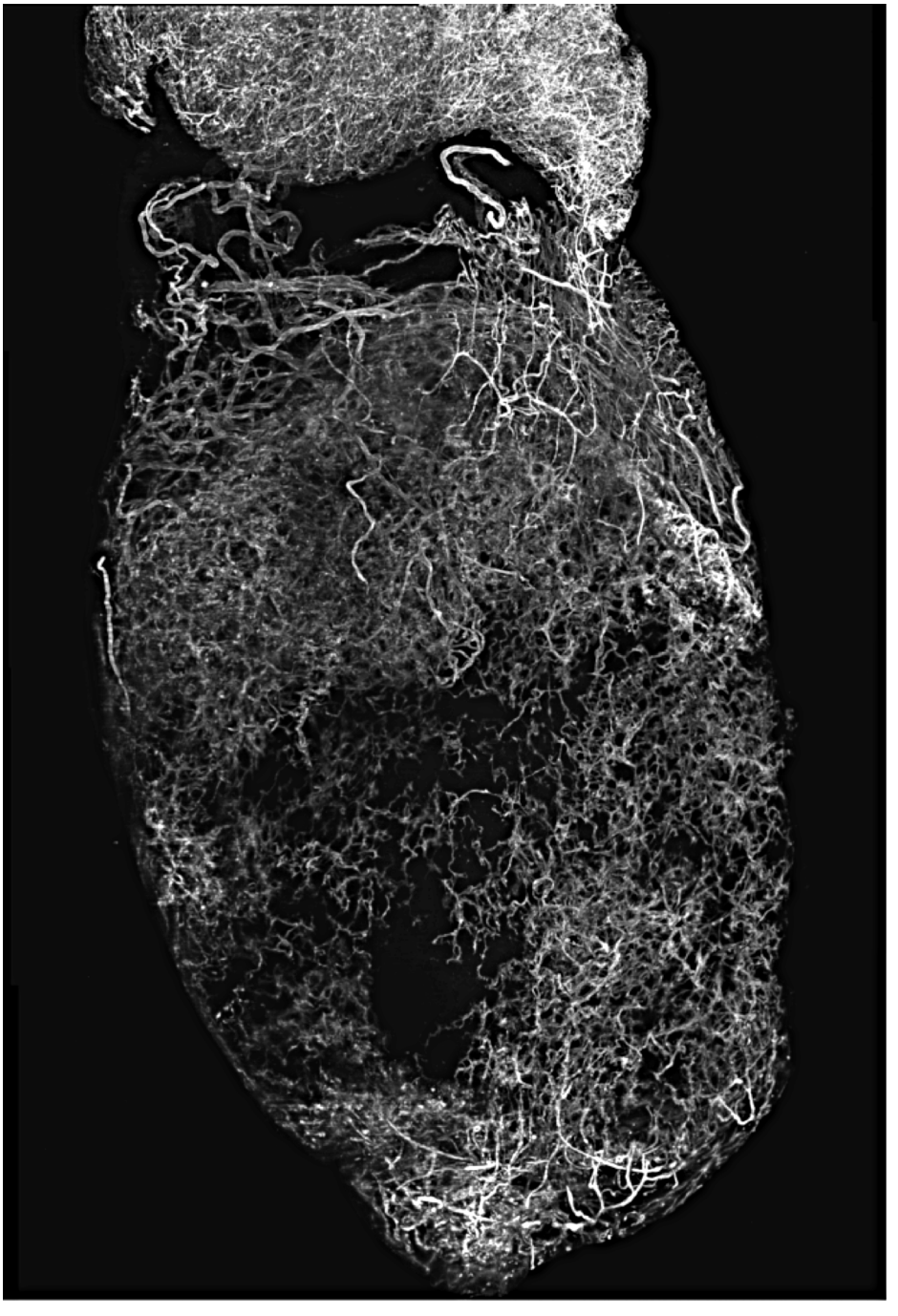


Figure 9.1 Maximum intensity projection of the original data set.

(pieces of organs, tumors, whole organisms . . .). The preparation protocol and the imaging are similar to Ref. [1]. For this project, mice developing some specific tumors are injected a rhodamine–lectin construct to stain their blood vessels before sacrificing. *Important note:* Two stacks cropped from the original data set are provided, namely "BloodVessels_small.tif" and "BloodVessels_med.tif." It is *highly recommended* to first work on the smaller stack as processing time is not negligible. You may test the final ImageJ macro on the larger stack.

9.1.4 Prerequisites

- 3D ImageJ Suite: To install the plug-in, select `Help` > `Update..`, click "Manage Update Sites," check "3D ImageJ Suite," click "Close," and then click "Apply changes." For a description of the plug-in, see imagejdocu.tudor.lu/doku.php?id=plugin:stacks:3d_ij_suite:start. We will use the multithreaded 3D filters that can now be found in `Plugins` > `3D` > `3D Fast Filters` (you may have to restart ImageJ).
- Lookup table (LUT) with random colors: Please copy the file "Random.lut" into the "luts" folder of your ImageJ installation. We will use this LUT to visualize segmented objects in label images.
- Restart ImageJ to make above actions come into effect.

9.2 Morphological Closing of Tubular Structures

9.2.1 Introduction

As the overall aim of the project is to trace the blood vessel network, we are not interested in the tubes' hollow structure. In fact, for segmentation it would be easier if the tubes were plain, because then we would not have to deal with dark "non-tube-voxels" inside bright "tube-voxels." Our first task is thus to try to "fill" the tubes, using grayscale morphological closing [2].

9.2.2 Workflow

Data Examination

Open and view the data in Fiji using the following commands:

Open file: `File` > `Open...` "../bloodvessels_small.tif"

View in 3D: `Plugins` > `3D Viewer`

Change brightness (in 3D viewer menu): `[Edit` > `Transfer Function` > `RGBA]`

Perform 3D Morphological Closing

As the resolution of the data set can be assumed reasonably isotropic and since the tubes can have any orientation, we will use a spherical structuring element for the morphological closing. Select the CloseGray filter in `Plugins > 3D > 3D Fast Filters` with same kernel radius in all dimensions. A closing radius of 6–8 µm (3–4 voxels) is a sensible value for the data set. This will not completely close the largest vessels; however, increasing the closing radius might merge the closest small vessels, so you have to go for a compromise here. If you have time, it is very instructive to also perform this grayscale closing operation by manually performing first a maximum filter followed by minimum filter.

9.2.3 Generate an ImageJ Macro

Implement a macro performing above operations.

Note: In the dialog box of the filters, it is possible to input the radius in physical units or in pixels but only the radius in pixel shows up in the macro recorder. As it is convenient to input a radius in physical units, you could write code to convert from micrometers to pixel units before calling the filter. For this, you will require the macro function **getPixelSize**. In general, to combine numbers with the text strings as ImageJ plug-in arguments you need **d2s(m,n)**, which converts a number m to a string keeping n decimals.

```
1  ////////////////////
2  // Initialization //
3  ////////////////////
4
5  run("Options...", "iterations=1 count=1 edm=Overwrite");
6  OriginalTitle = getTitle();
7
8  ////////////////////////////////////////////////////
9  // Morphological closing of tubular structures (1) //
10 ////////////////////////////////////////////////////
11
12 // Work on the small image "BloodVessels_small.tif";
13
14 // Parameters
15 ClosingRadius = 8 // units: micrometer
16 Nthreads = 8 // units: count
17
18 // Filtering: closing to fill hollow tubes
19 getPixelSize(unit, px, py, pz);
20 run("3D Fast Filters","filter=CloseGray radius_x_pix="+
      d2s(ClosingRadius/px,2)+" radius_y_pix="+d2s
```

```
    (ClosingRadius/py,2)+" radius_z_pix="+d2s
    (ClosingRadius/pz,2)+" Nb_cpus="+d2s(Nthreads,0));
21
22 rename("Closed.tif"); // we add suffix ".tif" because
    otherwise ImageJ adds it upon saving resulting in
    inconsistent image names
```

code/TubeAnalyst-1.ijm

9.3 Prefiltering to Enhance Filamentous Voxels

9.3.1 Introduction

The data sets used here exhibit a high contrast so that a simple intensity-based thresholding is almost sufficient to distinguish tube from background voxels. However, in case of higher noise and/or uneven sample staining, one may need to filter the data prior to thresholding. A good criterion to follow for this operation is to notice that a voxel is part of a filament if there is one direction along which the intensity is quite constant (along the filament) and two perpendicular directions along which the intensities quickly drop (perpendicular to the filament). The ImageJ command `Plugins > Analyze > Tubeness` computes a metric reflecting to what extent a voxel and its local neighborhood fulfill this criterion. The implemented algorithm is based on Ref. [3].

9.3.2 Workflow

Select the output image of above section ("Closed.tif").

Enhance Filamentous Voxels

Use the `Plugins > Analyze > Tubeness` command on the data (after the morphological closing) and check the result for different "Sigma," which controls the size of a Gaussian filter that is applied before the actual "Tubeness" computation. This Gaussian prefiltering indirectly determines the size of the neighborhood taken into account for computation of the local intensity distribution.

Sensible values are in the range of 6–8 µm, but you can experiment with different values. It is in fact usually almost impossible to find a value that is optimal for both the smallest and the largest vessels.

You will notice that the contrast is greatly enhanced after the filtering but voxels close to vessel branch points might be forced to zero as their neighborhood does not strictly follow the definition of being filamentous. If this problem is too pronounced, it is possible to perform another pass of morphological closing after the prefiltering to "repair" these gaps in the network.

9.3.3 Generate an ImageJ Macro Script

Implement a macro performing above operations.

```
1  ////////////////////////////////////////////////
2  // Prefiltering to enhance filamentous voxels (2)  //
3  ////////////////////////////////////////////////
4
5  // Work on the closed image
6  selectImage("Closed.tif");
7
8  // Parameters
9  VesselRadius = 8 // units: micrometer
10
11 run("Tubeness", "sigma="+d2s(VesselRadius,2)+" use");
12 rename("Tubeness.tif");
```

code/TubeAnalyst-2.ijm

9.4 Segmentation of Tubular Structures

9.4.1 Introduction

In order to analyze the tubular network, we need to decide whether a voxel is a part of a tube or part of the background. To do so, we will threshold the data, that is, assign voxels below or above a certain gray value as background or object. Typically, such thresholding also yields spurious isolated voxels that are not part of the tubular network. We will clean up such voxels based on the criterion that they are rather isolated and not connected to many other voxels.

9.4.2 Workflow

Convert Previous ("Tubeness") Image to 8-bit

This step is optional but it somewhat simplifies the following thresholding operation. You should be careful not to clip the intensities during the conversion: The easiest way is to find the voxel with minimum and maximum intensities in the stack by inspecting the stack histogram and setting the minimum and maximum intensity values of the display accordingly using `[Image > Adjust > Brightness/Contrast]` before the conversion using `[Image > Type > 8-bit]`.

Generate a Binary Image

Threshold the previous ("Tubeness") image using `Image > Adjust > Threshold` (manual, global thresholding). We recommend to convert the image to 8-bit

before thresholding as the Adjust Threshold interface works better with 8-bit than with 32-bit format. The aim is to adjust the lower bound of the threshold so that most of the vessels are thresholded without getting merged (if you followed the previous 8-bit conversion step a lower bound intensity around 8 gray values should work fine for both data sets).

Optional: Automated thresholding methods.

If you have time, you may explore some automated thresholding methods such as

- `Image > Adjust > Auto Threshold`
- `Image > Adjust > Auto Local Threshold`

Clean up Small Objects

Clean up object voxels that are isolated, that is, not connected to a minimum number of neighboring object voxels. This can be done by using the "minimum volume" option of `Analyze > 3D Objects Counter`. You can compare the initial segmentation mask and the resulting label mask after running "3D Objects Counter" in order to see what objects have been discarded. A practical value for the minimum volume filter is around 1000 voxels but you can experiment with different values.

Note: The voxels of the output have an intensity corresponding to the index of the connected object they are part of (label mask). The indexing starts at 1 and, depending on the active lookup table (LUT), some objects can thus appear very faint. To better visualize the results, it is handy to use the "Random.lut" (see Section 9.0.4). Just select it from `Image > Lookup Tables` or call **run("Random")** from your macro script.

9.4.3 Generate an ImageJ Macro Script

Write a macro performing above operations. The lower bound of the threshold should be stored in a variable **VesselThreshold** and the minimum volume for each connected component should be called **VesselVolumeThreshold**.

Note: For a proper 8-bit conversion of the Tubeness image, you need to set the display to the minimum and maximum gray value of the image stack. In a macro, the minimum and maximum value of the stack can be retrieved using **Stack.getStatistics(voxelCount, min, max, mean, std)**, and the current bounds of the display can be set by **setMinAndMax(min, max)**.

```
1 ////////////////////////////////////////////
2 // Segmentation of tubular structures (3) //
3 ////////////////////////////////////////////
4
5 // Work on the image after tubeness filtering
6 selectImage("Tubeness.tif");
7
```

```
// Parameters
VesselThreshold = 8 // units: gray values
VesselVolumeThreshold = 1000 // units: voxels

// Convert the 32-bit Tubeness image to 8-bit for thresholding
Stack.getStatistics(voxelCount, mean, min, max);
setMinAndMax(min,max);
run("8-bit");

// Threshold
setThreshold(VesselThreshold,255);
run("Convert to Mask", "method=Default background=Dark");
setSlice(nSlices/2); // move to central slices (only for nice
   viewing)

// Find connected components, remove too small objects and apply
   Random LUT
run("3D OC Options", "volume nb_of_obj._voxels dots_size=5
   font_size=10 redirect_to=none"); //  to ensure that Results
     Table is named "Results", i.e. uncheck the "macro friendly" naming
run("3D Objects Counter", "threshold=128 min.="+d2s
   (VesselVolumeThreshold,2)+" max.="+d2s(nSlices*getWidth()
*getHeight(),0)+" objects statistics summary");
run("Random"); // change LUT
```

code/TubeAnalyst-3.ijm

9.5 Skeletonization and Analysis of the Tubular Network

9.5.1 Introduction

In order to measure the network length and to count its branch points, we will reduce the segmented tubes (which have a certain thickness) to their one voxel wide centerlines. This process is called "Skeletonization." To identify the branch and end points of the skeletonized network, one can use the following observations:

- End-point voxels have less than two neighbors.
- Junction voxels have more than two neighbors.
- Slab voxels (remaining voxels) have exactly two neighbors.

To perform these tasks, we will use `Plugins > Skeleton > Skeletonize (2D/3D)` and `Plugins > Skeleton > Analyze Skeleton (2D/3D)` [4]. Skeletonization is based on a specific connectivity. For 3D images, ImageJ uses 26 neighbor per voxel by default.

9.5.2 Workflow

Skeletonization

The label mask first needs to be binarized (all nonzero voxels are objects) using `Image > Adjust > Threshold` with a lower threshold of 1 gray value. After this, you can skeletonize the binary image using `Plugins > Skeleton > Skeletonize (2D/3D)`. In order to visually check the skeletonization, you may overlay the binary mask (or the original data) with the skeleton using for instance the command `Image > Color > Merge Channels...`. The original image might be assigned the gray channel and the skeleton might be assigned the red channel.

Skeleton Analysis

Use `Plugins > Skeleton > Analyze Skeleton (2D/3D)` to analyze the skeleton (for the moment leave all pruning options unchecked). Examine the image output, which has the following color-coding:

- *End-point voxels:* Gray value of 30, appearing blue.
- *Junction voxels:* Gray value 70, appearing purple.
- *Slab voxels:* Gray value 127, appearing red.

Examine the output table, which not only contains the number of voxels falling into the three different classes but also the total length of the skeleton as well as their total number of end points and junctions. If there are several disconnected skeletons in the image, the statistics are reported for each of them. Observe that the number of junctions is smaller than the number of junction voxels, because at each junction there may be more than one voxel with more than two neighbors.

Skeleton 3D Visualization

Visualize the analyzed skeleton in the 3D viewer. You may realize that it is not looking very nice, because it is only one voxel thick. Also the difference between the slab, end point, and branch voxels is not easy to see.

Exercise: Figure out a way to alter the skeleton for 3D visualization purposes.

Hint: Change the value of the voxels by applying `Plugins > Process > Replace Value`, find an adequate combination of values and LUT, and finally thicken the skeleton by dilating it in 3D. Be very careful, when choosing the new values assigned to junction and end points as these voxels might be overwritten by close by slab voxels after dilation (local maximum operation).

9.5.3 Generate an ImageJ Macro Script

Implement a macro performing above operations.

```
////////////////////////////////////////////////////
// Skeletonization/analysis of tubular network (4) //
////////////////////////////////////////////////////

// Work on the label mask
selectImage("LabelMask.tif");

// Parameters
VisualisationDilation = 2 // units: pixels
PruneEnds = false; // true or false
Nthreads = 8;

// Make binary image
run("Duplicate...", "title=BinarizedTubes.tif duplicate"); //
   work on duplicate
run("8-bit");
setThreshold(1,255);
run("Convert to Mask", "method=Default background=Dark");

// Skeletonize
run("Duplicate...", "title=Skeleton duplicate"); // work on
   duplicate
run("Skeletonize (2D/3D)");
rename("Skeleton.tif");

// Remove end-point branches by pruning
if(PruneEnds) run("Analyze Skeleton (2D/3D)", "prune=none
   prune"); // no circular pruning, but end-point pruning
else run("Analyze Skeleton (2D/3D)", "prune=none"); // no
   circular pruning, no end-point pruning
IJ.renameResults("Results","Skeleton Results"); // rename
   results table for preventing it from being overwritten
   by other "Results"

// Beautify the visualisation:
// 1. Change analyzed skeleton colors
run("Macro...", "code=v=(v==127)*127+(v==30)*192+(v==70)*255
   stack");
// 2. Thicken analyzed skeleton
run("3D Fast Filters","filter=Maximum radius_x_pix="+d2s
  (VisualisationDilation,0)+" radius_y_pix="+d2s
  (VisualisationDilation,0)+" radius_z_pix="+d2s
  (VisualisationDilation,0)+" Nb_cpus="+d2s(Nthreads,0));
// 3. Change the LUT
run("Fire");
rename("ThickSkeleton.tif");
```

code/TubeAnalyst-4.ijm

9.6 Skeleton Pruning and Holes Closing (Optional)

9.6.1 Introduction

Depending on the roughness and thickness of the tubes in the raw data, the computed skeleton may contain false positive short branches and/or false positive small loops. These can eventually be removed by a process called "pruning". We will test the different pruning algorithms implemented in `Plugins > Skeleton > Analyze Skeleton (2D/3D)`.

9.6.2 Workflow

End-Point Pruning

Remove all branches containing exactly one end point by checking the "Prune ends" option in `Plugins > Skeleton > Analyze Skeleton (2D/3D)`. Carefully examine the image and check whether the pruning is always working as you would expect (see also Figure 9.2).

Circular Pruning

Sometimes, especially when the cross-section of the tubes is large the skeletonization can lead to (small) false circular skeleton parts. If this is the case in your data, try the "Prune cycle method" options of `Plugins > Skeleton > Analyze Skeleton (2D/3D)` and check if it helped removing the false cycles.

Note: One may consider additional algorithms; for instance to only remove branches up to a specified minimum length, however this is currently not implemented in `Analyze Skeleton (2D/3D)`. You should be very cautious with the pruning as important features of the network such as real loops and end-point segments may also be removed. Using it or not boils down to a trade-off between removing spurious branches and removing real network branches. It is always better to try to obtain a good segmentation mask in the first place but, as

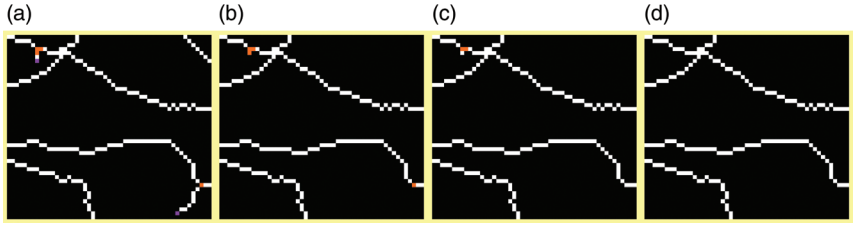


Figure 9.2 Skeleton annotation and pruning. Slab voxels are white, junction voxels are red, and end-point voxels are blue. Images are projections of 3D data and were subject to different processing steps: (a) Skeletonization => analysis. (b) Skeletonization => analysis with end-pruning. (c) Skeletonization => analysis with end-pruning => analysis. (d) Skeletonization => analysis with end-pruning => skeletonization => analysis.

you will notice, it is not easy to properly segment small and large vessels with such a simple image processing pipeline.

Fill Holes

The cycles in the large vessels usually originate from holes inside the segmented vessels, the problem can hence be mitigated by filling these holes in the binary mask before the skeletonization. This can be performed either in the 3D domain with `Plugins > 3D > 3D Fill Holes` or in 2D with `Process > Binary > Fill Holes`. In this last case, we must specify in the command call that the operation should be applied to the whole stack (slice by slice).

Note: More pixels will always be filled when the operation is performed in 2D, as a 2D hole appearing in a particular slice (e.g., a disk inside a cylinder) is not necessarily part of a 3D hole (the converse being true). In turn, 2D hole filling can generate some artifacts if the vessels form closed loops.

Morphological Closing

Sometimes, the large vessels of the binary mask are not only hollow but also a hole is pierced in their outside. These defects can lead to spurious small branches in the skeleton (as we saw before). If the holes are not too large, they can be filled in by morphological closing of the binary mask. If you have time, you can try this out.

Note: The simple workflow proposed in this practical is working reasonably well on the data sets we acquired, but, as we saw, it is pretty limited when it comes to segment a mixture of thin and thick vessels in the same stack. It cannot compete with some high-accuracy filament tracing methods, some of which are reviewed in Ref. [5]. More specifically, a very clever method is described in Ref. [6].

9.7 Extraction of Biologically Relevant Parameters

9.7.1 Introduction

As was previously motivated in Section 9.1.2, the density of the vascularization and branching points and the thickness of the vessels are crucial age indicators to understand how a tumor developed and possibly necrosed. We will now estimate these parameters on the segmented data.

9.7.2 Workflow

Generate an ImageJ Macro Script

The computation of the biologically relevant parameters cannot be achieved via the ImageJ menu, but you have to write an ImageJ macro.

Vessel Length and Number of Branch Points

To compute the total vessel length, you have to loop through the entries of the results table that you got from `Plugins > Skeleton > Analyze Skeleton (2D/3D)` and add up and/or multiply the respective entries in the respective columns ("# Branches", "Average Branch Length"). In addition to for-looping through the rows of the results table, you will need the **getResult(Column, row)** macro function in order to extract the values from the table.

Vessel Volume and Total Imaged Volume

To compute the total vessel volume, you should use the information obtained from the ImageJ macro function **Stack.getStatistics**. The volume can be expressed in voxel units or in physical unit. For the conversion, the calibration can be retrieved with **getPixelSize(width, height, depth, unit)**.

Density of Vascularization

To derive the density of vascularization, one needs to compute the fraction of space occupied by the vessels. This can be done by dividing the volume previously computed by the total imaged volume, which you can compute using a combination of the following functions:

- getWidth()
- getHeight()
- nSlices

Vessel Width

The average vessel width can readily be computed from the parameters we previously extracted . . . can you figure out how?

9.7.3 Generate an ImageJ Macro Script

Implement a macro perfroming the above operations.

```
1  ////////////////////////////////////////////////////////
2  // Extraction of biologically relevant parameters (6) //
3  ////////////////////////////////////////////////////////
4
5  // Needs the following tables and images:
6  // Table: Skeleton Results
7  // Image: BinarizedTubes.tif
8
9  getPixelSize(unit, px, py, pz);
10
```

```
// Total vessel length
IJ.renameResults("Skeleton Results", "Results"); // make table
   accessible
totalLength = 0;
nBranches = 0;
for(i=0; i<nResults; i++) {
  totalLength = totalLength + getResult("# Branches", i)
   *getResult("Average Branch Length", i);
  nBranches = nBranches + getResult("# Branches", i);
}
IJ.renameResults("Results","Skeleton Results"); // prevent
   table from being overwritten

// Total imaged and blood vessels volumes
selectWindow("BinarizedTubes.tif"); // image containing the
   segmented vessels
Stack.getStatistics(voxelCount, mean, min, max, stdDev);
totalImagedVolume = voxelCount*px*py*pz;
totalVolume = voxelCount*mean/255*px*py*pz;

// Mean vessel x-section and diameter
meanCrosssection = totalVolume / totalLength;
meanDiameter = 2*sqrt(meanCrosssection/PI);

print("");
print("");
print("Results");
print("----------");
print("Pruning = " + d2s(PruneEnds,0));
print("Total length = " + d2s(totalLength,2) + " " + unit);
print("Number of branches = " + d2s(nBranches,0));
print("Average branch length = " + d2s(totalLength/nBranches,2)
   + " " + unit);
print("Mean vessel cross-section = " + d2s(meanCrosssection,2)
   + " " + unit + "^2");
print("Mean vessel diameter = " + d2s(meanDiameter,2) + " " +
   unit);
print("Total vessel volume = " + d2s(totalVolume,2) + " "
  + unit + "^3");
print("Total imaged volume = " + d2s(totalImagedVolume,2) + " "
  + unit + "^3");
print("Volume fraction occupied by vessels = " + d2s
   (totalVolume/totalImagedVolume,2));
```

code/TubeAnalyst-6.ijm

9.8
Graphical User Interface (GUI)

The pipeline of operations we came up with can now be assembled to a complete macro to process the original stacks. Here, we add a dialog box allowing the user to enter the macro parameters via a GUI. The GUI can be created using the following macro commands:

- Dialog.create
- Dialog.addNumber
- Dialog.show
- Dialog.getNumber

Add the dialog box to the beginning of your macro code and make sure that the names of the parameters (variables) that you retrieve from the GUI are the same as the parameters in your macro code. In addition, you have to remove (comment out) the explicit assignments of the input parameters from your macro code (otherwise these explicit assignments will overwrite the assignments from the GUI).

```
1  ////////////////////////////////
2  // Graphical user interface (7) //
3  ////////////////////////////////
4
5  getPixelSize(unit, px, py, pz);
6  Dialog.create("TubeAnalyst");
7  Dialog.addNumber("Tube radius ("+unit+")", 6);
8  Dialog.addNumber("Vessel radius ("+unit+")", 8);
9  Dialog.addNumber("Vessel threshold", 8); //-1: man. calibration
10 Dialog.addNumber("Minimum vessel volume (pixels)", 1000);
11 Dialog.addNumber("Dilate Skeleton for viewing by (pixels)", 2);
12 Dialog.addNumber("Number of threads", 8);
13 Dialog.show;
14
15 ClosingRadius = Dialog.getNumber();
16 VesselRadius = Dialog.getNumber();
17 VesselThreshold = Dialog.getNumber();
18 VesselVolumeThreshold = Dialog.getNumber();
19 VisualisationDilation = Dialog.getNumber();
```

code/TubeAnalyst-7.ijm

9.9
3D Results Visualization

At the end of the macro, we can automatically load, show, and even animate the data in the 3D viewer, using commands such as

- **r**un("3D Viewer");
- call("ij3d.ImageJ3DViewer.setCoordinateSystem," "false");

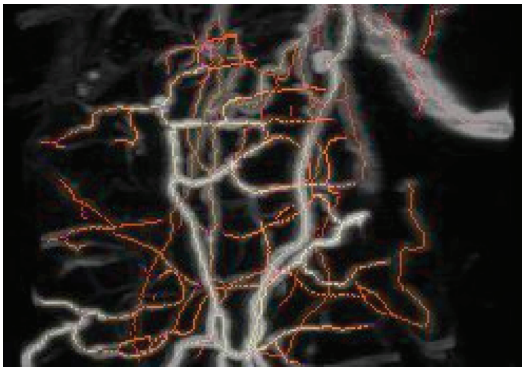


Figure 9.3 Maximum intensity projection of the original data set with overlaid skeleton.

- call("ij3d.ImageJ3DViewer.add," "ImageName," "None," "RefToImageName," "0," "true," "true," "true," "2," "0");
- call("ij3d.ImageJ3DViewer.startAnimate");
- wait(NumberOfMilliseconds); and
- call("ij3d.ImageJ3DViewer.stopAnimate").

The third item is used to add an image called "ImageName" to the viewer and label it "RefToImageName" in the 3D viewer `Edit > Select ..` menu entry. In case you did not have time to write up the complete macro during the practical a possible solution is provided in: ../code/**TubeAnalyst.ijm**. An example overlay of the output files is shown in Figure 9.3.

```
1  ////////////////////////////////////
2  // 3-D results visualisation (8) //
3  ////////////////////////////////////
4
5  // Merge original stack and skeleton
6  selectImage("Skeleton.tif");
7  run("Invert LUT");
8  run("Merge Channels...", "c1=*None* c2=Skeleton.tif c3="+
      OriginalTitle+" create keep");
9  run("Channels Tool...");
10
11 // Load original stack, label mask and analyzed skeleton in 3D
      viewer
12 run("3D Viewer");
13 call("ij3d.ImageJ3DViewer.setCoordinateSystem", "false");
14 call("ij3d.ImageJ3DViewer.add", OriginalTitle, "None",
      OriginalTitle, "0", "true", "true", "true", "2", "0");
```

```
call("ij3d.ImageJ3DViewer.add", "LabelMask.tif", "None",
   "LabelMask.tif", "0", "true", "true", "true", "2", "0");
call("ij3d.ImageJ3DViewer.add", "ThickSkeleton.tif", "None",
   "ThickSkeleton.tif", "0", "true", "true", "true", "2", "0");
call("ij3d.ImageJ3DViewer.startAnimate");
```

code/TubeAnalyst-8.ijm

9.10 Assignments

3D Viewer Automation

Add some more 3D viewer macro controls (in the last part of the complete macro). It is for instance possible to control the transparency of each object and many more features of the 3D viewer. You can experiment with it by recording these actions. Try to be creative!

Local Statistics

Write a macro to split the original stack in several subvolumes of user-defined size and estimate all the previous geometrical parameters in each of the subvolumes. Doing so, one can access to the local values of these biological parameters. For this assignment, you should rather use the larger stack (you might first need to slightly tune the parameters of your workflow to get a valid segmentation). It is possible to run the whole pipeline on each subvolume or, more efficiently, to run it once on the whole stack up to the segmentation and extract the information of interest in each subvolume afterward.

Acknowledgment

We would like to thank Alexandre Calon (IRB Barcelona) for preparing the mice and helping us set up the foundations of tumors quantitative analysis from MacroSPIM images.

References

1 Jährling, N., Becker, K., and Dodt, H.U. (2009) 3D-reconstruction of blood vessels by ultramicroscopy. *Organogenesis*, **5** (4), 227–230.

2 Vincent, L. (1993) Morphological grayscale reconstruction in image analysis: applications and efficient algorithms. *IEEE Trans. Image Process*, **2** (2), 176–201.

3 Sato, Y., Nakajima, S., Shiraga, N., Atsumi, H., Yoshida, S., Koller, T., Gerig, G., and Kikinis, R. (1998) Three-dimensional multi-scale line filter for segmentation and visualization of curvilinear structures in medical images. *Med. Image Anal.*, **2**, 143–168.

4 Arganda-Carreras, I., Fernandez-Gonzalez, R., Munoz-Barrutia, A., and Ortiz-De-Solorzano,

C. (2010) 3d reconstruction of histological sections: application to mammary gland tissue. *Microsc. Res. Tech.*, **73** (11), 1019–1029.

5 Lesage, D., Angelini, E.D., Bloch, I., and Funka-Lea, G. (2009) A review of 3d vessel lumen segmentation techniques: models, features and extraction schemes. *Med. Image Anal.*, **13** (6), 819–845.

6 Li, H. and Yezzi, A. (2006) Vessels as 4d curves: global minimal 4d paths to extract 3d tubular surfaces. in IEEE Conference on Computer Vision and Pattern Recognition Workshop, 2006 (CVPRW'06), pp. 82–82.

10
3D Quantitative Colocalization Analysis

Chong Zhang[1] *and Fabrice P. Cordelières*[2]

[1]*Universitat Pompeu Fabra, Carrer Tànger 122–140, 08018 Barcelona, Spain*
[2]*Bordeaux Imaging Center, UMS 3420 CNRS – Université de Bordeaux – US4 INSERM, Pôle d'imagerie photonique, Institut François Magendie, 146, Rue Léo-Saignat, 33077 Bordeaux, France*

10.1
Overview

10.1.1
Aim

In this project, we will implement two ImageJ macros to analyze both intensity- and object-based colocalization in 3D, and to visualize the colocalization results.

10.1.2
Introduction

Subcellular structures interact in numerous ways, which depend on spatial proximity or spatial correlations between the interacting structures. Colocalization analysis aims at finding such correlations, providing hints of potential interactions. If the structures have only simple spatial overlap with one another, it is called *cooccurrence*; if they not only overlap but also codistribute in proportion, it is then *correlation*. Colocalization may be evaluated visually, quantitatively, and statistically:

- It may be identified by superimposing two images and inspecting the appearance of the combined color. For example, colocalization of red and green structures can appear yellow. However, this intuitive method can work only when the intensity levels of the two images are similar (see a detailed example in [1]). Scatter plot of pixel intensities from the two images also qualitatively indicates colocalization, for example, the points form a straight line if the two structures correlate. But visual evaluation does not tell the degree of colocalization, nor if it is true colocalization at all.

Bioimage Data Analysis, First Edition. Edited by Kota Miura.

- In general, two categories of quantitative approaches to colocalization analysis can be found: intensity-based methods and object-based methods. Intensity-based methods compute global measures about colocalization, using the correlation information of intensities of two channels. Several review papers have been published during the last decade, where coefficients' meaning, interpretation, guide of use, and examples for colocalization are given [1–4]. Tools for quantifying these measures can be found in many image analysis open-source and commercial software packages, to name just a few: Fiji's JACoP plug-in and Coloc 2, CellProfiler, BioImageXD, Huygens, Imaris, Metamorph, and Volocity. Most object-based colocalization methods first segment and identify objects, and then account for objects' interdistances to analyze possible colocalization. Usually, two objects are considered colocalized, if the centroids of the objects are within certain distance [2,3,5], or if two objects with certain percentage of area/volume overlap [6,7]. We will implement some specific methods for both categories in Sections 10.2 and 10.3.
- Colocalization studies should generally perform some statistical analysis, in order to interpret whether the found cooccurrence or correlation is just a random coincidence or a true colocalization. A common method is Monte Carlo simulations [8] but it is computationally expensive. Recently, a new analytical statistics method based on Ripley's K function is proposed and included as an Icy plug-in, StatColoc [9].

10.1.3 Data Sets

10.1.3.1 HeLa Cells Data Set

Virologists often need to answer the questions of when and where the viral replication happens and the relevant virus–host interactions. The data set (see Figure 10.1a as an example) we are using in this module is HeLa cells imaged with a spinning disk microscope (imag courtesy of Dr. Ke Peng from Heidelberg University). Z serial images were acquired in three channels, from which two are used: channel 1 (C1, red) shows viral DNA in high contrast and channel 3 (C3, green) shows viral particles in high contrast (a viral structural protein). The high contrast signals either come from synthetic dyes or fluorescent protein. The goal is to identify the viral particles that have synthesized viral DNA indicating such structures represent replicating viral particles and potentially the viral replication sites. Thus, the identification can be achieved through a colocalization analysis between the objects in these two channels.

10.1.3.2 Synthetic Data Set

In order to help better understanding the steps during the object-based colocalization, we will first use a synthetic data set to test and build up the macro. And once our macro program is working, we will test it on the segmented HeLa cells data set. Figure 10.4 shows two 3D views of the synthetic data set with two channels, where channel 1 (*red*) has six objects and channel 2 (*blue*) has seven objects. Each object in channel 1 has different level of spatial overlap with one of

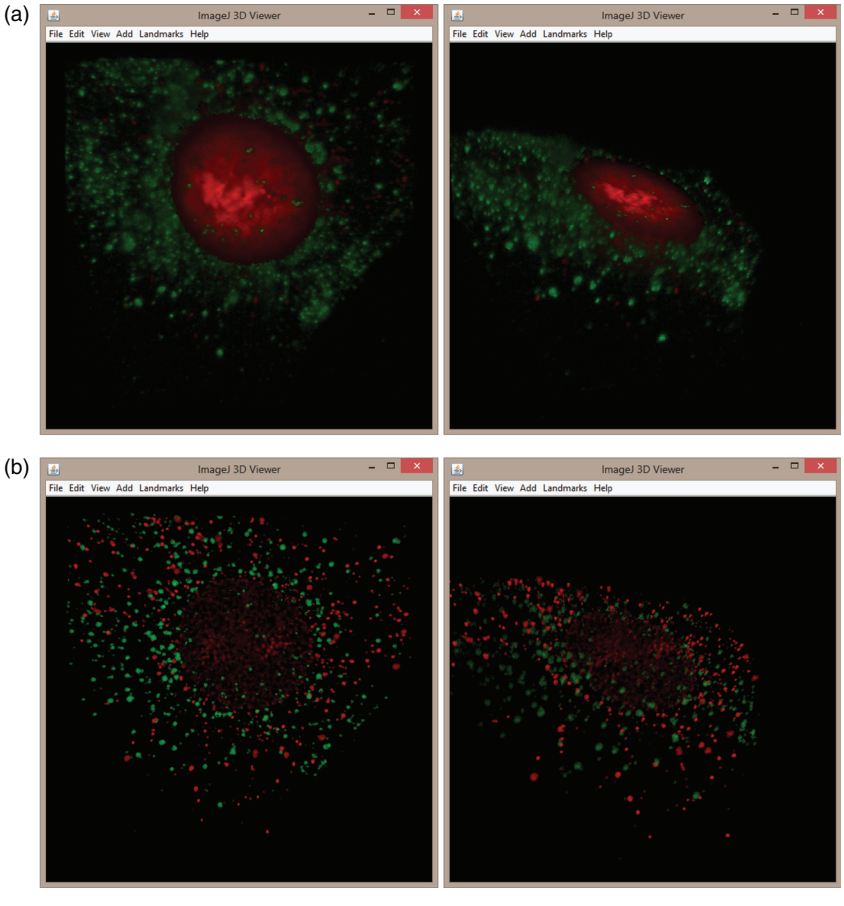


Figure 10.1 (a) The HeLa cells data set (with two channels) from two views. (b) The prediction from the trained classifiers using ilastik Pixel Classification workflow, also from two views. Higher intensity in the prediction images indicates higher probability of being the objects of interest.

the objects in channel 2. The synthetic data set can be found in this module's folder (C1_syn and C2_syn).

10.1.4 Image Preprocessing

Talking about colocalization, we often also think about deconvolution. Careful image restoration by deconvolution removes noise and increases contrast in images, improving the quality of colocalization analysis results. In Fiji, you can find several plugins for these tasks to try on your images, such as

- Parallel Iterative Deconvolution (`fiji.sc/Parallel_Iterative_Deconvolution`), where the point spread function (PSF) can be estimated using the

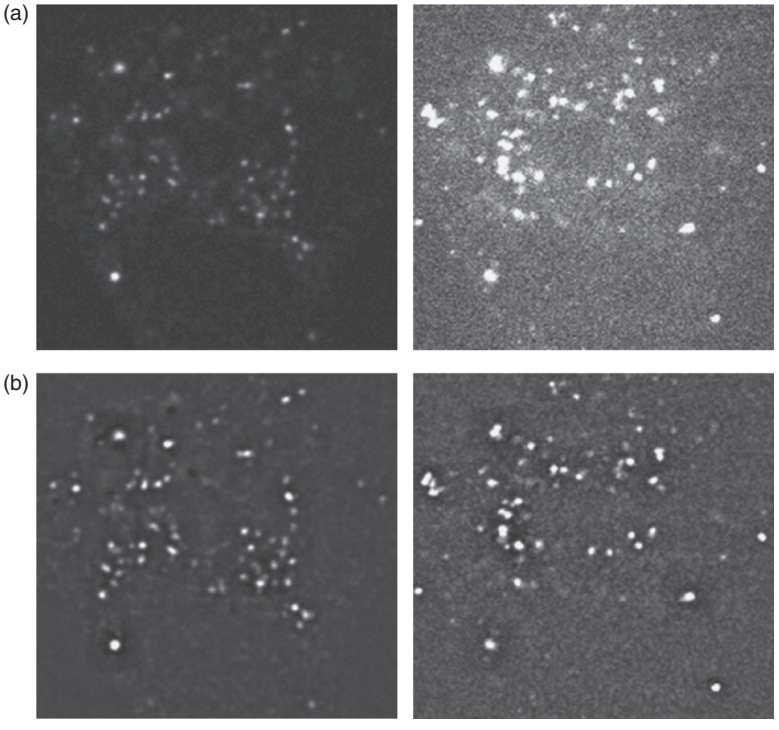


Figure 10.2 Example images before (a) and after (b) deconvolution.

Diffraction PSF 3D plug-in (`fiji.sc/Diffraction_PSF_3D`) (for our images, you can try to select "WPL" as Method, and use 20 or more iterations; PSF estimation needs media refractive index, numerical aperture, wavelength, and pixel spacing. You can also find "PSF_C1.tif" and "PSF_C3.tif" for the two channels, estimated from the Diffraction PSF 3D plugin.) An example can be found in Figure 10.2.

- To further remove background noise, you can try [`Process -> Subtract Background`] (for our images, the rolling ball radius can be set to 10 pixels).

However, deconvolution is not the focus of this module. Therefore, we would assume that the images to be processed during this module are either already deconvolved or are acquired with high image quality without the need of deconvolution. Other issues that may need to be dealt with during the preprocessing include illumination correction, noise removal, and background or artifacts disturbance. Since we have already been practicing techniques to handle these situations, here we would as well rather not to discuss them.

It is worth noting briefly here that before this step several points should be taken care of during the image acquisition and collection part.

- To have the imaging hardware setup appropriately. That is, to adjust the exposure time, detector gain, and offset so as to be able to detect the dimmest structures without saturating the brightest structures.

- To check for chromatic aberrations, one uses small beads that are fluorescent in many colors and thus should 100% colocalize with themselves. If they appear shifted, you have to realign your microscope or account for the shift during the analysis.
- To appropriately control bleed-through.

10.2 Intensity-Based Colocalization Methods

10.2.1 Overview

In this tutorial, we will implement a simple colocalization tool, in the form of an ImageJ macro. It will provide means to calculate most commonly used colocalization indicators and evaluators, namely Pearson's, Spearman's, and Manders coefficients.

10.2.2 Step 0. Creating the Framework: Generation of a User Interface and Initial Data Retrieval

Colocalization analysis is performed on a couple of images. The user has to choose which images to use as channel A and as channel B. In case multiple images are opened under ImageJ, it might be convenient to establish a list of all available images. We, therefore, create a first function, `getImageList()` that returns an array containing all images' titles. Basically, images are selected one after the other using the `selectImage(index)` function. Their titles are stored into an array that will be returned once the function is called:

```
1  //-------Retrieve images' list-----------
2  function getImageList(){
3    run("Tile");
4    out=newArray(nImages);
5    for(i=0; i<nImages; i++){
6      selectImage(i+1);
7      out[i]=getTitle();
8    }
9    return out;
10 }
```

The resulting array has to be stored into an array, `imgList`, to be recalled on demand. Next step is also prepared by creating two variables, `channel A` and `channel B`, which will accommodate the images' titles, initialized with empty

strings. A section containing all variables will be defined, on the top of the macro:

```
//-----------Variables---------------
var imgList=getImageList();
var channelA="";
var channelB="";
```

The graphical user interface (GUI) provides a convenient way to select the two images to analyze. It also allows restricting the analysis to only some evaluators as we plan to implement 3 of them. The initial GUI frame is setup by the `Dialog.create("Title")` command. Two drop-down lists are added, which display the images' list. Finally, the GUI is displayed using the `Dialog.show()` command. The user choices are retrieved through the `Dialog.getChoice` function that result is stored into a variable for later use. In this example, variables `channel A` and `channel B` will contain the titles of the two selected images:

```
//----------------GUI---------------
function GUI(){
  Dialog.create("Co-localisation tool");
  Dialog.addChoice("Channel A", imgList, imgList[0]);
  Dialog.addChoice("Channel B", imgList, imgList[1]);
  Dialog.show();

  channelA=Dialog.getChoice();
  channelB=Dialog.getChoice();
}
```

As these tools work on images' intensities, a routine has to be defined to get them from the images and store them into arrays. This step is performed by first retrieving the image's dimensions using the `getDimensions(width, height, channels, slices, frames)` function, then using a triple loop to go through all three dimensions of the image/stack. As this procedure has to be applied on two images, providing the image title as an input to the function avoids duplicating the block of commands:

```
//--------Retrieve image's intensities----
function getImageIntensities(title){
  selectWindow(title);
  getDimensions(width, height, channels, slices, frames);
  out=newArray(width*height*slices);
  index=0;

  for(z=1; z<=slices; z++){
    setSlice(z);
```

```
    for(y=0; y<height; y++){
      for(x=0; x<width; x++){
        out[index]=getPixel(x, y);
        index++;
      }
    }
  }

  return out;
}
```

Now is the time to put together all steps: three functions were created to 1-retrieve the list of opened images, 2-provide the user with a graphical interface, and 3-extract the images' intensities. Two variables have to be created, `A` and `B`, which will contain the intensities from both channels. The only missing part is a call to all three, as performed in the following listing:

```
//-----------Variables---------------
var imgList=getImageList();
var channelA="";
var channelB="";

var A=newArray(1);
var B=newArray(1);


//----------Processing steps-----------
GUI();
A=getImageIntensities(channelA);
B=getImageIntensities(channelB);

//-------Retrieve images' list-----------
function getImageList(){
  run("Tile");
  out=newArray(nImages);
  for(i=0; i<nImages; i++){
    selectImage(i+1);
    out[i]=getTitle();
  }
  return out;
}

//----------------GUI---------------
function GUI(){
  Dialog.create("Co-localisation tool");
  Dialog.addChoice("Channel A", imgList, imgList[0]);
```

```
  Dialog.addChoice("Channel B", imgList, imgList[1]);
  Dialog.show();

  channelA=Dialog.getChoice();
  channelB=Dialog.getChoice();
}

//--------Retrieve image's intensities----
function getImageIntensities(title){
  selectWindow(title);
  getDimensions(width, height, channels, slices, frames);
  out=newArray(width*height*slices);
  index=0;

  for(z=1; z<=slices; z++){
    setSlice(z);
    for(y=0; y<height; y++){
      for(x=0; x<width; x++){
        out[index]=getPixel(x, y);
        index++;
      }
    }
  }

  return out;
}
```

10.2.3 Step 1. First Look at Intensity-Based Colocalization: The Cytofluorogram

Colocalization studies generally look for a linear relationship between the concentrations of two molecules on structures of interest. If a unique stoichiometry of association exists, plotting the intensities of channel B against the ones from channel A for each pixel should result in a dots' cloud taking the shape of a single line. This plot is called cytofluorogram. This method has been borrowed from cytometry by the first confocal microscopists. Generating this representation is a starting point in colocalization studies. It allows evaluating the intensities' dependencies, and helps in data interpretation. Three main phenomena might impair analysis, that are visible on a cytofluorogram: background, noise, and bleedthrough. The latter occurs when the chromophore used for one channel is also detected in the second channel. As a consequence, monolabeled structures will appear in a large dynamic range over the first image, and a restricted range on the second image, both being highly correlated. The cytofluorogram will therefore display a linear dots' cloud, located near one of the axes of the graph. Another issue is the

background and noise, originating from various sources. It might occur in case the labeling was suboptimal, in case of a lack of specificity of the labeling reagents, or if the acquisition was done using improper parameters. As a consequence, the dots' cloud on the cytofluorogram will appear as more dispersed and correlation will be less obvious. Additionally, a large uncorrelated dots' cloud might appear close to the graph's origin. Finally, cytofluorogram may also display either multiple linear or single/multiple nonlinear dependency between intensities. Under those circumstances, it may help choosing the proper procedure. For single linear dependency, calculating the Pearson's coefficient will be adapted, while the use of Spearman's coefficient is required in nonlinear cases. Splitting the image into multiple regions of interest might also solve the multiple stoechiometries of association issue.

Implementing the cytofluorogram representation is quite straightforward using ImageJ macros. The instruction `Plot.create("Title", "X label", "Y label")` provides the framework to build axis. Another command exists, that takes X and Y data as input. However, this generates a plot where dots are connected by lines: as a dot cloud is required, use of the former command is recommended. In this case, the `Plot.setlimits(xMin, xMax, yMin, yMax)` instruction is required to set the limits of the plotting area. It will be fed with the output of the `Array.getStatistics(array, min, max, mean, stdDev)` command. Finally, the dots are added to the graphical frame using `Plot.add("dots", A, B)` and displayed through the call of `Plot.show()`. All instructions are packed into a new function as follows:

```
1 //--------Plot cytofluorogram---------
2 function cytofluorogram(){
3   Plot.create("Cytofluorogram", channelA, channelB);
4   Array.getStatistics(A, xMin, xMax, mean, stdDev);
5   Array.getStatistics(B, yMin, yMax, mean, stdDev);
6   Plot.setLimits(xMin, xMax, yMin, yMax);
7   Plot.add("dots", A, B);
8   Plot.show();
9 }
```

Now having a first available output for our macro, the GUI part should be modified. A checkbox is added, asking whether or not the cytofluorogram should be drawn. The user request is stored into a new boolean variable, `doCytofluorogram`, which is used in the processing part of the macro to call or not the relevant function. The final listing for Step 1 reads:

```
1 //-----------Variables---------------
2 var imgList=getImageList();
3 var channelA="";
4 var channelB="";
```

```
5
6  var A=newArray(1);
7  var B=newArray(1);
8
9  var doCytofluorogram=true;
10
11 //----------Processing steps-----------
12 GUI();
13
14 A=getImageIntensities(channelA);
15 B=getImageIntensities(channelB);
16
17 if(doCytofluorogram) cytofluorogram();
18
19 //-------Retrieve images' list-----------
20 function getImageList(){
21   run("Tile");
22   out=newArray(nImages);
23   for(i=0; i<nImages; i++){
24     selectImage(i+1);
25     out[i]=getTitle();
26   }
27   return out;
28 }
29
30 //----------------GUI---------------
31 function GUI(){
32   Dialog.create("Co-localisation tool");
33   Dialog.addChoice("Channel A", imgList, imgList[0]);
34   Dialog.addChoice("Channel B", imgList, imgList[1]);
35   Dialog.addCheckbox("Cytofluorogram", true);
36   Dialog.show();
37
38   channelA=Dialog.getChoice();
39   channelB=Dialog.getChoice();
40   doCytofluorogram=Dialog.getCheckbox();
41 }
42
43 //--------Retrieve image's intensities----
44 function getImageIntensities(title){
45   selectWindow(title);
46   getDimensions(width, height, channels, slices, frames);
47   out=newArray(width*height*slices);
48   index=0;
49
50   for(z=1; z<=slices; z++){
51     setSlice(z);
52     for(y=0; y<height; y++){
```

```
      for(x=0; x<width; x++){
        out[index]=getPixel(x, y);
        index++;
      }
    }
  }

  return out;
}

//--------Plot cytofluorogram---------
function cytofluorogram(){
  Plot.create("Cytofluorogram", channelA, channelB);
  Array.getStatistics(A,xMin, xMax, mean, stdDev);
  Array.getStatistics(B,yMin, yMax, mean, stdDev);
  Plot.setLimits(xMin, xMax, yMin, yMax);
  Plot.add("dots", A, B);
  Plot.show();
}
```

10.2.4 Step 2. Colocalization Seen Through Correlation Indicators: Pearson's and Spearman's Coefficients

Assuming a single stoichiometry of association for two markers of interest on structures, the intensities from channel B should linearly depend on the intensities from channel A for each single structures' pixel. This unique relationship might be described using a regular statistical descriptor: the correlation coefficient. When used for colocalization studies, this R^2 value is called Pearson's coefficient. It describes how well the dots' cloud on the cytofluorogram follows a line. However, it does not quantify the amount of colocalization and therefore belongs to the family of the colocalization indicators, as opposed to quantifiers. Its value is located within the [−1; 1] range, −1 being found in inverse correlation situations (the signals are mutually exclusive), 0 for absence of correlation (no relationship between signals), and 1 for a perfect correlation, only obtained when both images are exactly the same (Figure 10.3).

ImageJ embarks fitting capabilities. Using a user-defined equation, it performs the adjustment of the model over the experimental data points. Step 1 allowed to extract numerical values that will now be fitted with a linear model, taking the $y = ax + b$ form. The `Fit.doFit("y=a*x+b", A, B)` command is used to initialize the fitting process. A plot is retrieved, presenting the dots' cloud and a red line representing the average line approximating the dots' cloud (instruction: `Fit.plot`). Finally, Pearson's coefficient is retrieved (`Fit.rSquared`) and the fitting data are logged (slope and offset, `Fit.logResults`). The slope of the line is a

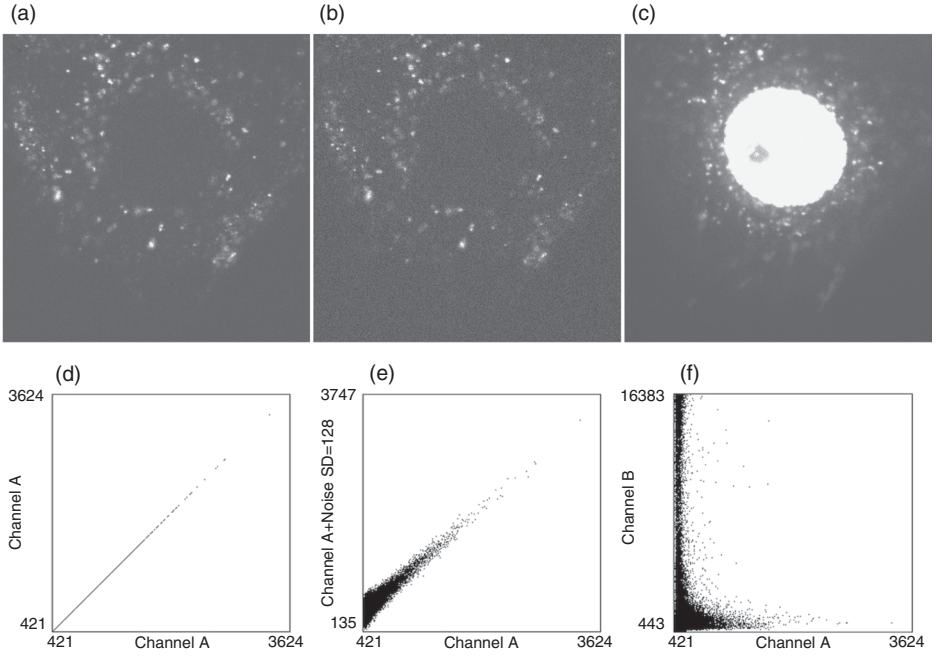


Figure 10.3 An example of cytofluorograms on the HeLa data set. (a) A single slice was extracted from the original data set (images (a) and (c)). Image (a) was submitted to the `Process/Add noise/Add Specified Noise` function to give image (b). Cytofluorograms (d), (e), (f) were obtained using couples of images (a) and (a), (a) and (b), and (a) and (c), respectively. (d) presents perfect colocalization: cytofluorogram displays a single line. (e) presents an initially perfect correlation, degraded by noise. (f) explicits what could be interpreted as exclusion of signals.

really rough estimator of the proteins' stoichiometry and modulo the efficiency of the fluorescence process, and assuming the chromophores' surrounding has a limited effect on the fluorescence efficiency. Packed into a function, it reads:

```
1  //--------Pearson coefficient----------
2  function pearson(){
3    Fit.doFit("y=a*x+b", A, B);
4    Fit.plot;
5    rename("Pearson, X: "+channelA+", Y: "+channelB);
6    print("------------------");
7    print("Pearson's coefficient: "+Fit.rSquared);
8    print("Fitting parameters:");
9    Fit.logResults;
10 }
```

Now that the first draft of the macro has been written, the reader might start testing it. Open two images, and run the macro. In the GUI, select the two images to analyze, check the boxes corresponding to the analysis to be

performed. Keep in mind that processing images pixel per pixel, using a macro, is not the optimal way to run such analysis: some time is required to get the results, aspecially if the images' dimensions are large.

Association between proteins of interest, especially in case of a direct interaction, might be seen as a saturable phenomenon. As a consequence, the domain where linear intensity correlation occurs might be limited, reaching a plateau for high-intensity values. Under this type of situation, Pearson's coefficient fails to give a faithful indication of intensities correlation. A second correlation coefficient might therefore be used, which instead applies not to values but their rank over the intensity range. First, the values are sorted by increasing order. Instead of taking the intensities, the rank of the value is taken to calculate a regular correlation coefficient: it then takes the name of Spearman's coefficient. This method has the benefit of linearizing the data, making the regular correlation coefficient usable.

Instructions exist under ImageJ to manipulate arrays of values. Among them, the `Array.rankPositions(array)` instruction returns an array where values are replaced by their rank within the range. A function can be written, taking the same basis as for Pearson's coefficient calculation, but taking the rank values instead of the raw values. The function's listing reads:

```
1  //--------Spearman coefficient--------
2  function spearman(){
3    rankedA=Array.rankPositions(A);
4    rankedB=Array.rankPositions(B);
5
6    Fit.doFit("y=a*x+b", rankedA, rankedB);
7    Fit.plot;
8    rename("Spearman, X: "+channelA+", Y: "+channelB);
9    print("-------------------");
10   print("Spearman's coefficient: "+Fit.rSquared);
11   print("Fitting parameters:");
12   Fit.logResults;
13 }
```

This new step within the macro creation ends by creating two boolean variables, `doPearson` and `doSpearman`, to store whether or not those parameters should be calculated, updating the GUI and the conditional processing part.

To test the macro, one can for instance take twice the same image. Casting one of the two images to 32-bits (`Image/32-bit`), squaring the values (`Process/Math/Square`) then casting back to 16-bits (`Image/16-bit`) will generate two images carrying a nonlinear dependency of intensities. In this case, use of Pearson's coefficient will lead to a value away from the ideal value of one, while the use of Spearman's coefficient will more appropriately give a perfect value of one. Another means to test the macro is to work on duplicated images, adding noise to one of the two (`Process/Noise/Add specific noise`). Comparing the values of Pearson's coefficient under noisy situation is a good means to appreciate its strength/lack of strength also shared with the Spearman's coefficient.

10.2.5 Step 3. Colocalization Seen Through Quantifiers: Manders' Coefficients

Correlation coefficients assume a highly degree of dependency between intensities. However, two proteins might be associated to structures in proportions that vary from one to another. In this case, looking for correlation won't lead to a faithful estimation of cooccurence. An alternative method consists in first partitioning the image into object/nonobject pixels, identifying zones of overlapping objects and finally calculating the percentage of signal involved in the overlap. This mode of quantification is known as Manders' coefficients, where two values are calculated, expressing the percentage of signal from channel A that finds a counterpart on channel B (coefficient M1) and the reverse way round (M2 coefficient).

Implementing the Manders' coefficients requires a user input: an intensity threshold should be set for both channels independently. The overlapping pixels are defined as carrying both an intensity A greater than the threshold defined on image A and an intensity B greater than the threshold defined on image B. Having already extracted the intensities from both channels, it is easy to sum all intensities from A fulfilling the former condition, same being true for intensities from B. Implementation is straightforward: as user input is required, the threshold box is first activated. Assuming fluorescence images, the background is considered as black (`setAutoThreshold("Default dark");`). The `waitForUser ("text")` is used to allow the user to set the parameters. Finally, the threshold values are retrieved (`getThreshold(lowerA, upperA)`). In the Manders' function, we will define four numerical variables accommodating the globally summed thresholded intensity of A (`sumA`), of B (`sumB`), the summed intensity of A finding a counterpart on B (`sumAColoc`) and the reverse way round (`sumBColoc`). The variables are fed using a loop through the intensities' arrays, final calculation of M1 and M2 performed, and logged. The core of the Manders' function reads:

```
1  //--------Manders' coefficients--------
2  function manders(){
3    selectWindow(channelA);
4    run("Threshold...");
5    setAutoThreshold("Default dark");
6    waitForUser("Set the appropriate threshold on "+
         channelA+"  then click on Ok");
7    getThreshold(lowerA, upperA);
8
9    selectWindow(channelB);
10   run("Threshold...");
11   setAutoThreshold("Default dark");
12   waitForUser("Set the appropriate threshold on "+
         channelB+" then click on Ok");
```

```
13    getThreshold(lowerB, upperB);
14
15    sumA=0;
16    sumAColoc=0;
17
18    sumB=0;
19    sumBColoc=0;
20
21    for(i=0; i<A.length; i++){
22      if(A[i]>lowerA) sumA+=A[i];
23      if(B[i]>lowerB) sumB+=B[i];
24
25      if(A[i]>lowerA && B[i]>lowerB){
26        sumAColoc+=A[i];
27        sumBColoc+=B[i];
28      }
29    }
30
31    print("-------------------");
32    print("Mander's coefficients: ");
33    print("Threshold for A: "+lowerA+"; Threshold for
          channel B: "+lowerB);
34    print("M1 (% intensity of A co-localising): "+
          (sumAColoc*100/sumA));
35    print("M2 (% intensity of B co-localising): "+
          (sumBColoc*100/sumB));
36  }
```

The major difficulty when working with Manders' coefficients is to define the intensity thresholds. Depending on their values, part of the structures might be excluded from the analysis. Many algorithms exist under ImageJ to automatically define the limit value. The experimenter might choose from the available list, or decide to set it based on visual inspection. By doing so, the user might impair the analysis. More advanced methods exist, as will be latter seen under the object-based section.

10.2.6 Step 4. The Final Macro

The tools developed in this chapter is mainly of demonstration purpose. While giving the expected results, it lacks strength to compute parameters on large data sets. Assuming it will be used by nonmacro familiar persons, it is of good practice to handle exceptions. For instance, what happens when two images of different dimensions are selected? ImageJ will return an error that can be personalized. In the following listing, care has been taken on this kind of potential issues:

```
1  //-----------Variables---------------
2  var imgList=getImageList();
3  var channelA="";
4  var channelB="";
5
6  var A=newArray(1);
7  var B=newArray(1);
8
9  var doCytofluorogram=true;
10 var doPearson=true;
11 var doSpearman=true;
12 var doManders=true;
13
14 //----------Processing steps-----------
15 GUI();
16
17 A=getImageIntensities(channelA);
18 B=getImageIntensities(channelB);
19
20 if(A.length!=B.length) exit("The two images should have
       the same dimensions");
21
22
23 if(doCytofluorogram) cytofluorogram();
24 if(doPearson) pearson();
25 if(doSpearman) spearman();
26 if(doManders) manders();
27
28
29 //-------Retrieve images' list-----------
30 function getImageList(){
31   run("Tile");
32   out=newArray(nImages);
33   for(i=0; i<nImages; i++){
34     selectImage(i+1);
35     out[i]=getTitle();
36   }
37   return out;
38 }
39
40 //----------------GUI---------------
41 function GUI(){
42   if(imgList.length<2) exit("This macros requires at least
         two images");
43
44   Dialog.create("Co-localisation tool");
45   Dialog.addChoice("Channel A", imgList, imgList[0]);
46   Dialog.addChoice("Channel B", imgList, imgList[1]);
```

```
47    Dialog.addCheckbox("Cytofluorogram", true);
48    Dialog.addCheckbox("Pearson's coefficient", true);
49    Dialog.addCheckbox("Spearman's coefficient", true);
50    Dialog.addCheckbox("Manders' coefficients", true);
51    Dialog.show();
52
53    channelA=Dialog.getChoice();
54    channelB=Dialog.getChoice();
55    doCytofluorogram=Dialog.getCheckbox();
56    doPearson=Dialog.getCheckbox();
57    doSpearman=Dialog.getCheckbox();
58    doManders=Dialog.getCheckbox();
59  }
60
61  //--------Retrieve image's intensities----
62  function getImageIntensities(title){
63    selectWindow(title);
64    getDimensions(width, height, channels, slices, frames);
65    out=newArray(width*height*slices);
66    index=0;
67
68    for(z=1; z<=slices; z++){
69      setSlice(z);
70      for(y=0; y<height; y++){
71        for(x=0; x<width; x++){
72          out[index]=getPixel(x, y);
73          index++;
74        }
75      }
76    }
77
78    return out;
79  }
80
81  //--------Plot cytofluorogram---------
82  function cytofluorogram(){
83    Plot.create("Cytofluorogram", channelA, channelB);
84    Array.getStatistics(A, xMin, xMax, mean, stdDev);
85    Array.getStatistics(B, yMin, yMax, mean, stdDev);
86    Plot.setLimits(xMin, xMax, yMin, yMax);
87    Plot.add("dots", A, B);
88    Plot.show();
89  }
90
91  //--------Pearson coefficient----------
92  function pearson(){
93    Fit.doFit("y=a*x+b", A, B);
94    Fit.plot;
```

```
95    rename("Pearson, X: "+channelA+", Y: "+channelB);
96    print("------------------");
97    print("Pearson's coefficient: "+Fit.rSquared);
98    print("Fitting parameters:");
99    Fit.logResults;
100 }
101
102 //--------Spearman coefficient--------
103 function spearman(){
104   rankedA=Array.rankPositions(A);
105   rankedB=Array.rankPositions(B);
106
107   Fit.doFit("y=a*x+b", rankedA, rankedB);
108   Fit.plot;
109   rename("Spearman, X: "+channelA+", Y: "+channelB);
110   print("------------------");
111   print("Spearman's coefficient: "+Fit.rSquared);
112   print("Fitting parameters:");
113   Fit.logResults;
114 }
115
116 //--------Manders' coefficients--------
117 function manders(){
118   selectWindow(channelA);
119   run("Threshold...");
120   setAutoThreshold("Default dark");
121   waitForUser("Set the appropriate threshold on
          "+channelA+" then click on Ok");
122   getThreshold(lowerA, upperA);
123
124   selectWindow(channelB);
125   run("Threshold...");
126   setAutoThreshold("Default dark");
127   waitForUser("Set the appropriate threshold on
          "+channelB+" then click on Ok");
128   getThreshold(lowerB, upperB);
129
130   sumA=0;
131   sumAColoc=0;
132
133   sumB=0;
134   sumBColoc=0;
135
136   for(i=0; i<A.length; i++){
137     if(A[i]>lowerA) sumA+=A[i];
138     if(B[i]>lowerB) sumB+=B[i];
139
140     if(A[i]>lowerA && B[i]>lowerB){
```

```
141         sumAColoc+=A[i];
142         sumBColoc+=B[i];
143       }
144     }
145
146     print("------------------");
147     print("Mander's coefficients: ");
148     print("Threshold for A: "+lowerA+"; Threshold for channel B:
            "+lowerB);
149     print("M1 (% intensity of A co-localising):
            "+(sumAColoc*100/sumA));
150     print("M2 (% intensity of B co-localising):
            "+(sumBColoc*100/sumB));
151 }
```

10.3 Object-Based Colocalization Methods

10.3.1 Overview

Cautions must be taken when using intensity-based measures. For example, the number of colocalized objects cannot be quantified. Often, including or not extracellular regions when no signals for colocalization are present changes values of these measures. Yet, many of such measures is only suitable for situations that both signals/objects cooccur in fix proportion to one another, that is, they are colocalized in a linear relationship. Furthermore, correlation-based measures are sensitive to image contrast and noise. Given these issues, in this section we will write our own object-based colocalization macro based on the percentage of area/volume overlap between objects. We could establish our own specific protocols and control settings, and at the same time practice more with ImageJ macro scripting.

10.3.2 Step 1. Object Segmentation

There are multiple tools for detecting spot-like structures, for example, spot detection method in Module 3, Fiji particle tracker in Module 4, Icy Spot detector, and so on. In particular, machine-learning-based methods allow for training a classifier that can "pick" objects of interest out of other signals in the image. For data sets like in Figure 10.1a, we are only interested in highly contrast spot-like structures, which coexist with others such as nucleus and cloud-like objects. Therefore, it might be especially suitable to train a spot classifier using tools like ilastik Pixel Classification workflow [10].

We will start by the already segmented image, using any means you find that gives the best segmented spots (Figure 10.4).

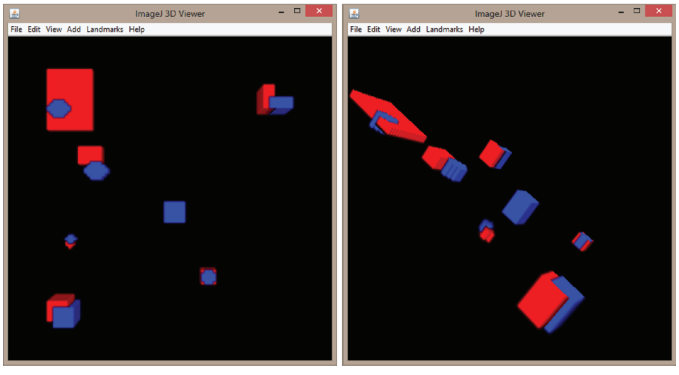


Figure 10.4 Synthetic 3D data set from two views.

10.3.3 Step 2. Filtering Objects by Size

Often, the segmentation contains objects that are not interesting for us such as noise or other structures. Since object-based methods concern individual objects, then we should apply some filtering criteria in order to discard them for further analysis. Such criteria could be, for example:

- (3D/2D/1D) size range of the object-of-interest (in each channel)
- object shape, for example, circularity,[1] compactness[2]
- object location

It should be mentioned that this step greatly influences the colocalization measurements. We will discuss only size-related filtering here. Our strategy consists of two steps: filtering in 3D, and then in 2D.

10.3.3.1 Filtering by 3D Sizes

`3D Objects Counter` is able to do this. We can open the macro file "`Step2_filtering.ijm`" and add steps in. After running it, we will be asked to specify the directories where the images from two channels are, and also a directory to store results. After that, we will see a window with many parameters to setup. Let's ignore them for now and just click "Ok."

Then let's select the image of channel 1 (folder synthetic/C1_syn), and then run [`Analyze > 3D Object Counter`]. In the popup window, there are two parameters of our interest, Min and Max in the `Size filter` field. Let's suppose

1) Circularity measures how round, or circular-shape like, the object is. In Fiji, the range of this parameter is between 0 and 1. The more roundish the object, the closer to 1 the circularity.
2) Compactness is a property that measures how bounded all points in the object are, for example, within some fixed distance of each other, surface-area to volume ratio. In Fiji, we can find such measurements options from the downloadable plug-in in `Plugins > 3D > 3D Manager Options`.

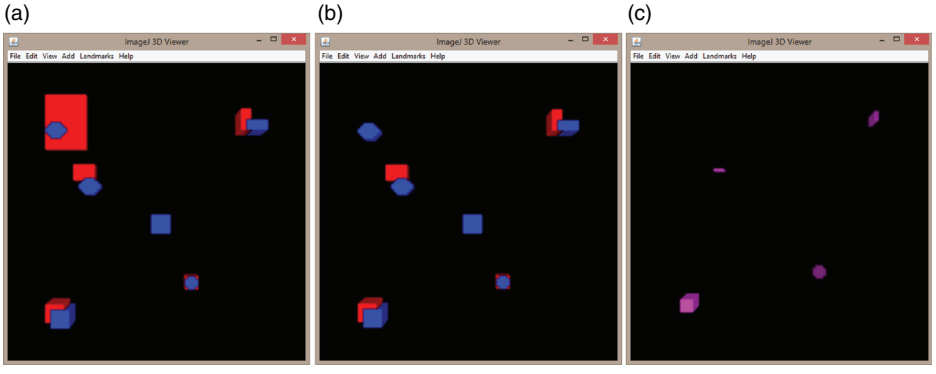


Figure 10.5 Synthetic 3D data set after first filtering in 3D (a), then also in 2D (b), and the overlapping regions after the filtering steps (c).

that the object of interest should have a size of minimum 3 voxels and maximum 10 voxels in each of the three dimensions, resulting in object volume sizes (in voxels): `Min` $= 3^3 = 27$ and `Max` $= 10^3 = 1000$. This filtering step removes the smallest object from both channels. Although they may seem to overlap with objects in the other channel, they are likely to be for example, noise and their spatial cooccurrence could be coming from randomly distributed particles/noises that are close to each other by chance.

Since in this module we may have to produce many intermediate images, so it might be a good practice to `rename` these intermediate images. And if they will not be used any further, they might as well just be closed by running [`File > Close`].

10.3.3.2 Filtering by 2D Sizes

You may have noticed that this filtering criteria is not sufficient to remove the largest object in channel 1 (Figure 10.5a). This is because, for example, the object is very thin in one or two axes and large in other(s), thus the total 3D size may be within the size range of the object of interest. In this case, it makes sense to have another filtering but in 2D this time.

Note that this section could be specific to the application in this particular module. For those who would not encounter this problem, you can skip this section.

Solution 1

One possibility is to set a size range in the *XY* plane, and if needed also a maximum spanning size in the *Z* axis. In order to do this, we will use [`Analyze > Analyze Particles`] and its `Size` parameter setting. Note that the `Analyze Particles` function works on 2D binary images only, then we first need to convert the label image by the 3D Object Counter, which is a label image. A simple thresholding should work, since the label starts at 1. Normally, the binary image after thresholding has value 0 and 255. Sometimes, a binary image of value 0 and

1 makes further analysis easier. To do so, we could divide every pixel by this image's nonzero value, that is, 255, using [Process > Math > Divide].

So, similarly we could set the value to be, for example, 9–100 (i.e., $3^2 - 10^2$). Also, distinct size ranges can be employed for each channel if needed. Additionally, for some applications, the parameter Circularity could also be used. Since we would like to obtain the filtered segmentation image, thus Masks will be chosen to Show.

What should we do if the 2D filtering is to be done in the *XZ* plane? If the axes are swapped, so as the original *XZ* plane will be the new *XY* plane, then we could apply the same procedure on the axes-swapped image. This can be done with *TransformJ Rotate* in [Plugins > Transform > TransformJ]. It applies to the image a rotation transform around the three main axes. Please note that if the image to be processed is large, this will not be the optimal solution as it will be both time and computation expensive to apply a transform and interpolate the whole image. A faster option could be *TransformJ Turn* in [Plugins > Transform > TransformJ]. The advantage of using this plug-in rather than the more generally applicable *TransformJ Rotate* is that it does not involve interpolation of image values but merely a reordering of the image elements. As a result, it generally requires much less computation time, and does not affect image quality.

Solution 2

Another possibility is an anisotropic erosion + dilation operations in 3D. In this case, we would only want to erode and then dilate in one dimension – the one where object-to-be-removed is thin. We will use [Process > Filters > Minimum 3D] and then [Process > Filters > Maximum 3D] with the same radius settings for each dimension. We will set both X and Y radii to 0 and Z radius to 3.0, because the largest object in Channel 1 is thin in *Z* axis.

In general, it should work – provided the filter radius is not smaller than the object radius along its smallest dimension, then the object should disappear and not return. This also gives the possibility to filter anisotropically considering the fact that in many microscopy images the Z spacing is larger than those in *X* and *Y* axes. Note that the erode/dilate alternative in [Plugins > Process] will not work, because the filter size is not adjustable. On the other hand, please keep in mind that the erode and dilate operations may modify object shape, especially when objects are not roundish blob-like.

Depending on specific applications, more filtering steps can be applied before or after the previous size filtering, such as shape or location. In this module, we will not discuss in details and assume the size filtering is sufficient for our task (Figure 10.5b).

10.3.4 Step 3. Finding Spatial Overlapping Objects

In order to find colocalized objects, we will first find the overlapping (or shared) parts of the two filtered channels, and then identify the corresponding objects in

each channel that contain these overlapping regions. To achieve this, what do we need to do? There are multiple ways, all of which involve:

- in each channel, label the objects so as to identify them;
- in each channel, calculate the volume (in voxels) of each object;
- compute the objects' overlapping regions of the two channels;
- calculate the volume of each overlapping region; and
- find the labels of objects in each channel that overlap with some object in the other channel.

These tasks could be done with [Analyze > 3D Object Counter] and [Plugins > 3D > 3D Manager].

10.3.4.1 Find in Each Channel Those Overlapping Objects

We could see that in both channels, the overlapping regions have values higher than zero; while the rest of the two images have either one or both channels with zero background. Therefore, if we multiply the two images, only the overlapping regions will show values higher than zero. So, we can run [Process > Image Calculator], set the object maps of from the two channels as Image 1 and Image 2, and set Multiply as Operation. Let's call the obtained multiplicative image as "SharedMask." Figure 10.5c shows the 3D view of the overlapping regions after the filtering steps. We can see that the two overlapping but with too large and too small objects (see Figure 10.4) are now excluded, remaining four overlapping regions of the two channels.

The images of both channels that contain objects filtered by size are binary images of values 0 and 1, thus the "SharedMask" is also a binary image of values 0 and 1. Now, if we add (through [Process > Image Calculator] with Add as Operation) the "SharedMask" to the filtered binary images of each channel, the two resultant images would give background zero value, all the objects in each channel value 1, except where the overlaps are, that is, 2. Then when using "3D hysteresis thresholding" plug-in ([Plugins > 3D > 3D Hysteresis Thresholding]), only regions with value >= a low threshold (i.e., 1) that also contain value >= a high threshold (i.e., 2) are extracted, that is, the objects containing the overlaps. Therefore, we have obtained also one image per channel, which contains only objects that overlap with some object in the other channel.

10.3.4.2 Calculate the Volume of Each Overlapping Region

Since we define the object volume overlap ratio as the colocalization criteria, objects and their volume values in both channels and the SharedMask should be calculated. We need to make sure in [Analyze > 3D OC Options], to check the option of "Nb of Obj. voxels" (if we count voxels) or "Volume" (number of voxels multiplying voxel size). If the voxel size is not set, by default it is 1 for each dimension, thus these two parameters give the same value. After running the 3D Objects Counter, the resultant Object map gives each object a unique nonzero label and the background the zero label. Also, the measurements will be shown in a table. In 3D OC Options menu, if

"Store results within a table named after the image (macro friendly)" is not checked, the results are stored in the `Results` table. This way we could use the built-in macro functions `nResults` and `getResult` to access items in the table, with the following code:

```
1 ObjVolume_ch1 = newArray (nResults);
2 print("ObjVolume_ch1 ("+nResults+"): ");
3 for(i=0;i<nResults);i++){
4   ObjVolume_ch1[i] = getResult("Nb of obj. voxels", i);
5   print(ObjVolume_ch1[i]);
6 }
```

This code first creates a new array *ObjVolume_ch1* (line 1), so as to store all objects' volumes in the current channel. It is then filled with the values obtained from the `Results` table (line 4). For checking the code, we could also print the array (line 5). Similar arrays and object labels should be created for the other channel and the SharedMask. We will duplicate the above codes and just modify the array name when using the tables from the other images.

10.3.4.3 Identify Overlapping Object Pairs

So far we have obtained the objects' labels and volume values in each channel and the "SharedMask" image. The next task will be to identify the labels of each object pair that overlap, in order to further find out their corresponding overlap volume ratio. Before jumping into the next part, let's ask ourselves a question – does our problem involve analyzing individual colocalized objects, or rather a global measure that tells something about colocalized objects as a whole? If the answer to your problem is the later, then we could skip the remaining of this section and the following one. Instead, probably the overlap-related global measures such as Mander's Coefficients, Overlap Coefficient could suffice. Therefore, our macro in Section 10.2 or the ImageJ plug-in JACoP [2,3] could do the job.

If you are interested in studying individual colocalized objects and their further characteristics, such as their spatial distribution, shape, size, and so on, we will go on to the next task of identifying in each channel that objects overlap with objects in the other channel. Since we know the overlapping regions, then we just need to look at the same regions in each of the two channels to get these objects that have overlapping parts in the other channel. We will use the plug-in [`Plugins > 3D > 3D Manager`]. It plays a similar role as the `ROI Manager` but in 3D. So the first thing is to add the 3D overlapping regions into the `3D Manager` so that we could use them for further measurements, and also for inspecting the same regions on other images such as the label images. To do so, we will use the following code:[3)]

3) Note that the "`Ext`" is a built-in function added by plugins using the MacroExtension interface.

```
selectImage("Objects map of Shared Mask");
run("3D Manager");
Ext.Manager3D_AddImage();
Ext.Manager3D_SelectAll();
```

After adding the 3D overlapping regions into the 3D Manager, we will select the object label image of channel 2, so as to find out the corresponding label values for every overlapping region. Similar to what we have done before, we can measure the maximum intensity value of each region from the label image in channel 2. So, we will check `3D Manager Options` and select (and only select) the `Maximum gray value`. And then click the `Quantif_3D` in the `3D Manager` window. It will give a window "3D quantif," with a column "Max." This column stores the maximum gray values of each region in 3D Manager from the label image in channel 2.

Since it is not the usual "`Results`" table, we can't use the `nResults` and `getResult` built-in functions to access the contents of this table directly. One easy option, as also is shown in the tubular network module, is with a built-in function `IJ.renameResults` (if you have ImageJ version 1.46 or later) that changes the title of a results table from one to another. Thus, we can always change the name of any table to "Results" and then use the easier built-in functions `nResults` and `getResults` to get table contents.

Just for the general interest, another (more complicated) way to access information from a table with any name other than "`Results`" can be seen in the following code. It shows an example of how we can get one column's items from a table, that is, the column that obtains the objects in each channel that have these overlapping regions information from the table "`3D quantif`":

```
1 shareObjLabelInCh2 = newArray(numSharedObjects);
2 selectWindow("3D quantif");
3 tbl = getInfo("window.contents");
4 tblLines = split(tbl, "\n");
5 idx=0;
6 for (i=1; i<= numSharedObjects; i++){
7   lineA = split(strA[i], "\t");
8   shareObjLabelInCh2[idx]=lineA[2];
9 }
```

where (in lines 3–4) "*tbl*" stores everything in the "`3D quantif`" table as *string*, and we can get each item from the table by two "`split`" operations: first into lines through the line break "`\n`" (as in line 4) and then into separate items through the tab or column break "`\t`" (as in line 7). *shareObjLabelInCh2* is an array of size *numSharedObjects*, the number of overlapping regions, which stores the object labels that contain the corresponding overlap part in channel 2.

10.3.5 Step 4. Filtering the Colocalization Objects by Volume Overlap Percentage Criteria

We have finally arrived to the stage that we are ready to select "real" colocalized objects from the candidates. As we will use the volume overlap criteria, let's first calculate the volume overlap ratio. There may be several ways to define the ratio, we will use, for example, the ratio between the overlapping region and the volume of the smaller of the two objects. In order to not complicate the problem too much, we will assume that objects in one of the channels have smaller size. This could be a reasonable assumption in many biological applications. The following code realizes the process of determining the colocalization using such selection criteria, assuming the channel with smaller objects is channel 2. A new image stack, "SharedMask Filtered," is created and filled with the overlapping regions that have volume size larger than the specified percentage of the corresponding object in channel 2:

```
selectImage("Objects map of Shared Mask");
run("3D Manager");
Ext.Manager3D_AddImage();
newImage("SharedMask Filtered", "8-bit black", width,
    height, slices);
for (j=0; j<numSharedObjects; j++){
  objLabelInC2 = shareObjLabelInCh2[j];
  voRatio=ObjVolume_shared[j]/ObjVolume_ch2[objLabelInC2-1];
  print("volume overlap ratio of "+j+"th object in channel 2 is:
      "+voRatio+" = "+ObjVolume_shared[j]+"/"+ObjVolume_ch2
      [objLabelInC2-1]);

  //select the objects that have volume overlapping higher
      than a user specified ratio, "volOverlap"
  if (voRatio>volOverlap){
    numColocObjects=numColocObjects+1;
    Ext.Manager3D_Select(j);
    Ext.Manager3D_FillStack(1,1,1);
  }
}
```

where "numColocObjects" gives the total number of colocalization object pairs, "voRatio" computes the volume overlap ratio, and "volOverlap" is a user-specified threshold that discards objects with lower overlap ratio. `Manager3D_FillStack` fills the newly created image with the selected 3D regions. For each region, the values filled can be all the same, or a different one as label. The final colocalized objects in each channel can be obtained using again the `3D Hysteresis Thresholding`, as we have done previously, but with the newly created overlapping regions image, "SharedMask Filter." In the synthetic example, the four candidate objects have volume overlap ratio of: 0.11, 0.51, 0.25, and 1 (see

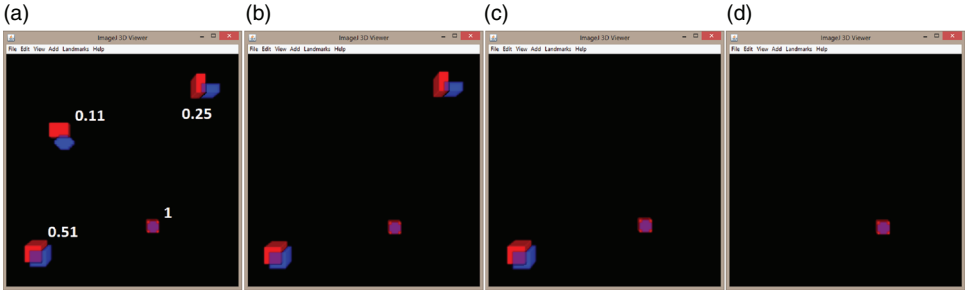


Figure 10.6 (a) Candidate colocalization objects from the two channels, the number next to each object pair shows the volume overlap ratio compared to the blue channel objects. (b–d) Determined colocalization object pairs using specified ratio threshold of 0.2, 0.3, and 0.52, respectively.

Figure 10.6a). And if we specify "volOverlap" to be, for example, 0.2, 0.3, 0.52, the final "real" colocalization results differ, as shown in the three images on the right side, respectively, in Figure 10.6.

10.3.6 Step 5. Visualizing Results

To better examine 3D data, Fiji offers [`Plugins > 3D Viewer`] to visualize rendered volumes, surfaces, or orthogonal slices. After opening the "3D Viewer" window, images can be loaded using either [`File > Open`] (for any image on disk) or [`Add > From Image`] (for images already loaded in Fiji). Multiple images can be loaded. For overlapping images, we could modify image's transparency by [`Edit > Change transparency`]. Image brightness can be changed using [`Edit > Transfer Function > RGBA`]. There are many more possibilities to control and edit the visualization properties, we will leave this as a homework for you to exploit further. Examples of visualizing 3D data using this viewer can be found in many figures in this module such as Figures 10.5–10.7.

10.3.7 Step 6. Testing the Macro on HeLa Cells

The pipeline of operations we came up with can now be assembled to a complete macro to process the original HeLa cells stacks. We only miss one step at the very beginning. Because the original HeLa cells images are first segmented by ilastik. And we exported the predictions of both channels, which are not binary image since they are the probabilities of each voxel being the object of interest. Thus, we would provide a threshold value to binarize the images. Again, here the threshold value can be set differently for the channels. In order to make things general to work for images with different intensity levels, if we consider a threshold always between 0 and 1, then it can be scaled according to the image

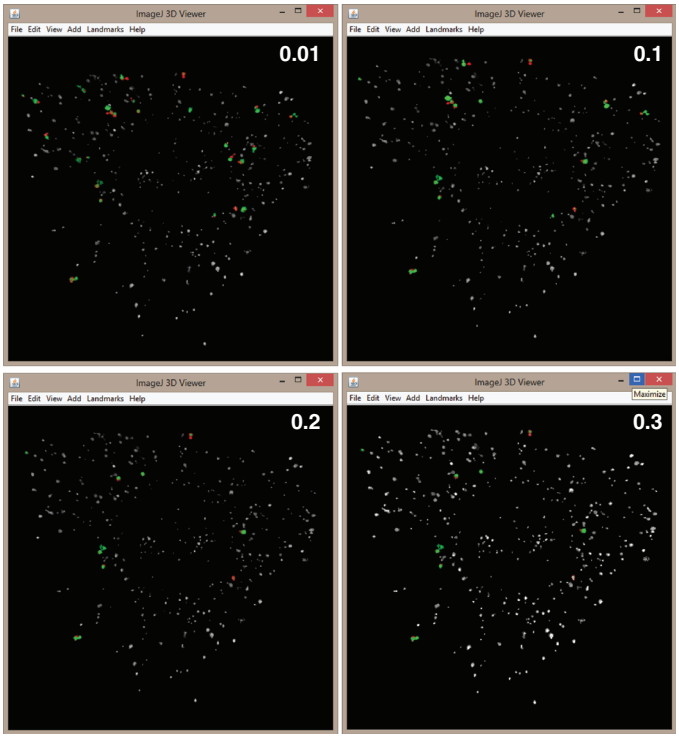


Figure 10.7 Colocalized objects in channel 1 (*red*) and channel 2 (*green*), together with all filtered objects in channel 1 (*white with transparency*) using the specified ratio thresholds: 0.01, 0.1, 0.2, and 0.3. Their corresponding number of determined colocalization objects are 25, 14, 9, and 9, respectively.

intensity range. To do this, the built-in function `getMinAndMax` can be used to get the original image intensity range, and then the threshold value can be calculated accordingly:

```
1 getMinAndMax(min,max);
2 setThreshold(min+(max-min+1)*thres,max);
3 run("Convert to Mask", "method=Default background=Dark black");
```

where "thres" is the threshold value between 0 and 1, and we assume the background has low intensity value in this case.

So, now when we try to analyze the HeLa cell images, the parameters used for the synthetic data set may not be applicable anymore. Of course, we could manually modify each value through the entire macro file we made. But this is not efficient. So, it is better to use variables for these parameters, and specify them, for example, in the beginning of the macro code.

```
Dialog.create("Dialog Window Title");
Dialog.addNumber("Probability map threshold: ", 0.5);
Dialog.addMessage("");
Dialog.addNumber("Minimum 3D object size (in voxels): ", 5);
Dialog.addNumber("Volume overlap for colocalization: ", 0.2);
Dialog.addMessage(" ");
Dialog.addCheckbox("Display in 3D Viewer?", false);
Dialog.show();

thres_pm_ch1 = Dialog.getNumber();
min3DObjSize=Dialog.getNumber();
volOverlap = Dialog.getNumber();
Display3D = Dialog.getCheckbox();
```

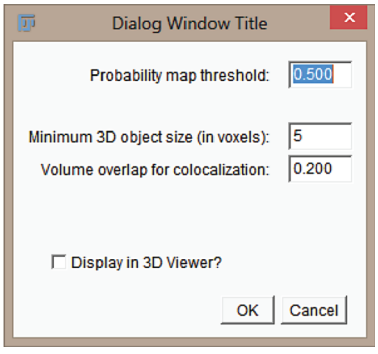


Figure 10.8 An example of creating a dialog window for parameters to be specified by users.

After the macro is "parameterized," we can specify values. For this data set, let's set the threshold to be 0.5 for both channels, and 3D object sizes to be [5,500], and maximum 2D object sizes in *XY* plane are 150 (for channel 1) and 50 (for channel 2). And volume overlap ratio threshold can be any value between 0 and 1. Figure 10.7 shows a few example results of the HeLa cell images.

Please note that for a complete colocalization analysis, further examination steps are needed such as accuracy evaluation, robustness evaluation, and reliability evaluation like comparing to random events. These are out of the scope of this module thus not discussed here.

10.3.8 Step 7. Setting Up a Parameter Interface

You may find modifying parameters inside the source code of the macro not an elegant usage. If you are willing, constructing a simple user interface window for parameters is made very easy in Fiji, as we have already seen in Section 10.2. We will refresh it and also create an interface for setting up the object-based parameters. The `Dialog.*` functions offer a practical dialog box/window with just a few lines of code. An example is shown in Figure 10.8, with the code on the left side and the generated dialog window the right side. Please note that the parameters that fetch the values from the dialog should be specified following the same top–down order as the dialog. Now, we can create customized dialog window for the parameters that we would like user to specify.

In case you do not have time to write up the complete macro during the module, a possible solution is provided in this module's folder (`coloc.ijm`).

10.4 Tips and Tools

- ImageJ Built-in Macro Functions: http://rsbweb.nih.gov/ij/developer/macro/functions.html

- 3D ImageJ Suite: download: http://imagejdocu.tudor.lu/lib/exe/fetch.php?media=plugin:stacks:3d_ij_suite:mcib3d-suite.zip and unzipping it in your plugins folder. Detailed description of the plug-in: imagejdocu.tudor.lu/doku.php?id=plugin:stacks:3d_ij_suite:start.
- JACoP (includes object-based method): imagejdocu.tudor.lu/doku.php?id=plugin:analysis:jacop_2.0:just_another_colocalization_plugin:start

References

1 Dunn, K.W., Kamocka, M.M., and McDonald, J.H. (2011) A practical guide to evaluating colocalization in biological microscopy. *Am. J. Physiol. Cell Physiol.*, **300**, C723–C742.

2 Bolte, S. and Cordelières, F.P. (2006) A guided tour into subcellular colocalization analysis in light microscopy. *J. Microsc.*, **224**, 213–232.

3 Cordelières, F.P. and Bolte, S. (2014) Experimenters' guide to colocalization studies: finding a way through indicators and quantifiers, in practice. *Methods Cell Biol.*, **123**, 395–408.

4 Zinchuk, V. and Zinchuk, O. (2008) Quantitative colocalization analysis of confocal fluorescence microscopy images. *Curr. Protoc. Cell Biol.*, **U4**, 19.

5 Obara, B., Jabeen, A., Fernandez, N., and Laissue, P.P. (2013) A novel method for quantified, superresolved, three-dimensional colocalisation of isotropic, fluorescent particles. *Histochem. Cell Biol.*, **139**(3), 391–402.

6 Rizk, A., Paul, G., Incardona, P., Bugarski, M., Mansouri, M., Niemann, A., Ziegler, U., Berger, P., and Sbalzarini, I.F. (2014) Segmentation and quantification of subcellular structures in fluorescence microscopy images using Squassh. *Nat. Protoc.*, **9**(3), 586–596.

7 Wörz, S., Sander, P., Pfannmöller, M., Rieker, R.J., Joos, S., Mechtersheimer, G., Boukamp, P., Lichter, P., and Rohr, K. (2010) 3D geometry-based quantification of colocalizations in multichannel 3D microscopy images of human soft tissue tumors. *IEEE Trans. Med. Imaging*, **29**(8), 1474–1484.

8 Fletcher, P.A., Scriven, D.R., Schulson, M.N., and Moore, D.W. (2010) Multi-image colocalization and its statistical significance. *Biophys. J.*, **99**, 1996–2005.

9 Lagache, T., Meas-Yedid, V., and Olivo-Marin, J.C. (2013) A statistical analysis of spatial colocalization using Ripley's K function, in IEEE International Symposium on Biomedical Imaging, pp. 896–899.

10 Sommer, C., Straehle, C., Koethe, U., and Hamprecht, F.A. (2011) ilastik: interactive learning and segmentation toolkit, in IEEE International Symposium on Biomedical Imaging.

Index

Note: (IJ) stands for ImageJ, (M) stands for Matlab and (R) stands for R

Bioimage Data Analysis, First Edition. Edited by Kota Miura.

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.