Eötvös Loránd University
Faculty of Informatics

Department of Algorithms
and Their Applications

Graph Digitizer

Supervisor:

Krisztián Tichler PhD
Senior Lecturer

Author:

László Kocsis
Computer Science BSc Student

Budapest, 2018

This page should be the original Thesis Topic Declaration.

Contents

Chapter 1

Introduction

Solving research problems often requires the analysis of existing data sets. Computerized analysis requires digital input, which means that the data must be properly digitized. An important special case is the digitization of graphs of continuous functions.

For example, lots of interesting continuous functions are plotted in figures of scientific publications. The source data of these charts could be utilized to check the calculations of the authors; to perform further data analysis; to validate novel analysis methods; or in computer simulations, as input, or for testing the output. Digitization is needed, if the plotted data cannot be acquired from the authors. Although more and more scientific journals require authors to make all raw data available in a public repository, this is often not the case.

Analog signal recordings are another example. Modern laboratories prefer digital signals, but there are exceptions, like the paper-based ECG recordings used in medicine. There is also a huge amount of existing analog traces, recorded during the analog era of science. Without digitization, these records are lost.

Graph digitization starts with obtaining a digital image of the graph. Noise, grid lines, intersecting curves, artificial discontinuities must be eliminated. The curve of the graph is usually multiple pixels thick, but only the pixels of its center line are needed. To convert the center line to a function, its vertical pixel strings must be replaced by their middle point. Finally, there may be axes or scale bars in the image, which may be used to calibrate the extracted points.

The purpose of the present work is to create a software for scientists, which helps them digitizing graphs of continuous functions, by providing tools to perform these steps in an automatic way.

Chapter 2

User Documentation

2.1 Overview

The Graph Digitizer application can be used to extract the points of a continuous function from an image of its graph. After loading the image, the curve of the graph is automatically thinned to its one-pixel-wide skeleton [1]. The (x, y) points of the function are calculated from the coordinates of the skeleton's pixels. If the graph contains an axis or scale bar in the X and/or Y direction, then the extracted function can be calibrated, after entering the real values corresponding to two different pixel positions in the same direction.

2.2 Installation and uninstallation

Graph Digitizer is a desktop application for Microsoft Windows. After installing the prerequisites, it can be deployed by simply copying the executable to the target computer.

Prerequisites:

- Microsoft Visual C++ 2017 Redistributable (x86)
- Microsoft .NET Framework 4.7.1

Supported operating systems (both 32- and 64-bit systems are supported):

- Microsoft Windows 7 Professional
- Microsoft Windows 8.1 Professional

- Microsoft Windows 10 Professional

Since only one image is processed at a time, and there is no reason for its size to be extremely large, the hardware requirements are low. Any personal computer with a 2-core, 2 GHz CPU, 2 GB RAM, and at least 1 GB of free storage space should be good enough. No special video card is needed.

Graph Digitizer can be removed by deleting the executable. The prerequisites need not be removed.

2.3 Usage

Graph Digitizer guides the user through the steps of the graph digitization process. Before starting this process, the user has to acquire a digital image of the graph, using a digital camera or an image scanner. If the image is part of an already digitized or originally digital document, like a PDF file of a scientific article or e-book, then taking a screenshot of the graph is sufficient. Note that the resolution of screenshots is usually low, so it is useful to zoom in on the graph beforehand.

Figure 2.1 shows the test image that will be used in the subsequent chapters to demonstrate the usage of the application. It is a screenshot of a graph from a PDF file of a scientific paper ([2], Figure 4), taken in Acrobat Reader, at 300% magnification, on a 96 DPI display.

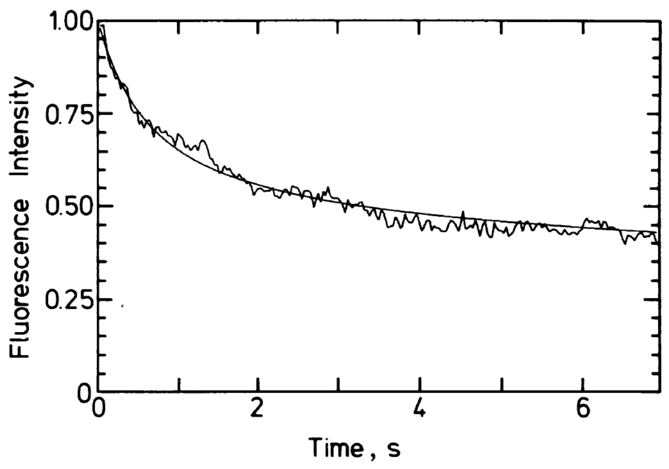


Figure 2.1: The test image [2]

2.3.1 Step #0: Preprocessing

Graph Digitizer can only process curves which are separated from other parts of the image, like other curves, gridlines, axes, labels etc. Any raster image editor can be used to do this.

After the curve is separated, its thickness has to be examined. If it is too thin, then the extracted graph will have too much digitization noise. If it is too thick, then its sharper peaks will be blunted. The ideal thickness is usually approximately 5-7 pixels, which can be achieved by resampling the image, using any image editor which is capable of smooth pixel interpolation.

Our test image (Figure 2.1) contains a measured photobleaching curve (noisy), and a fitted theoretical curve (smooth). The aim is to extract the measured curve, so the pixels of the theoretical curve must be deleted. This was a few minutes' work in Microsoft Paint. Then the image was resampled in IrfanView to double the resolution. The result is shown in Figure 2.2.

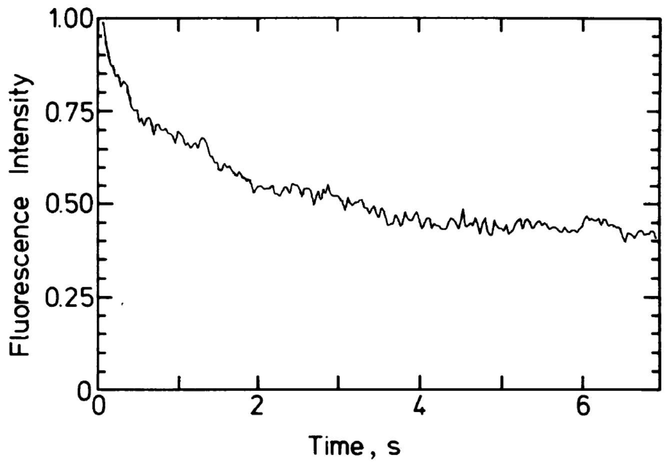
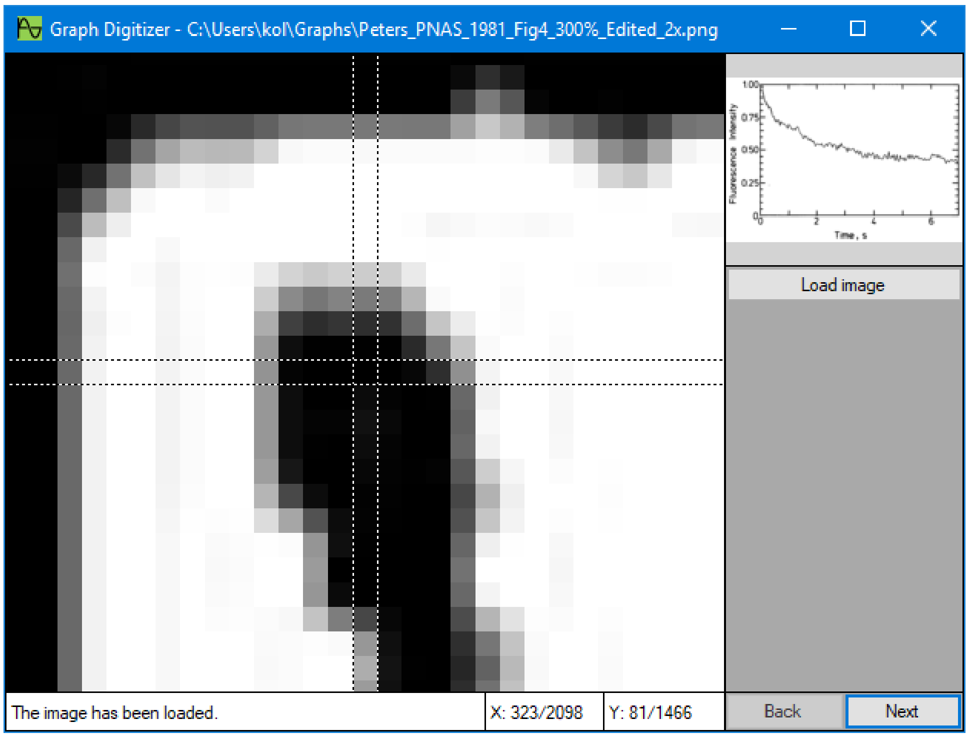


Figure 2.2: The test image after preprocessing

2.3.2 Step #1: Loading the image

The start screen of Graph Digitizer has a "Load image" button. Only 24-bit RGB images can be loaded, which are stored in BMP, JPG, TIF or PNG format. After selecting the image, it will be loaded and displayed in the user interface. In the upper-right corner, the whole image is displayed in a minimap. The main part of the interface is occupied by the image editor, which shows a strongly magnified portion of the image. The image editor has crosshairs, which target a single pixel, making pixel-precise image manipulations possible. The status bar under the image editor shows the location of the crosshairs in pixel coordinates.

Right after the image is loaded, the crosshairs target the upper-left corner pixel of the image. The crosshairs can be moved around using the arrow keys. Holding down the Shift key while pressing the arrow keys increases the movement speed tenfold. Larger, targeted jumps can also be done, by clicking on the minimap. Figure 2.3 shows the loaded image, with the crosshairs targeting the left end of the curve to be extracted.


Graph Digitizer - C:\Users\kol\Graphs\Peters_PNAS_1981_Fig4_300%_Edited_2x.png
Load image
Back
Next
Fluorescence intensity
Time, s

The image has been loaded.

X: 323/2098

Y: 81/1466

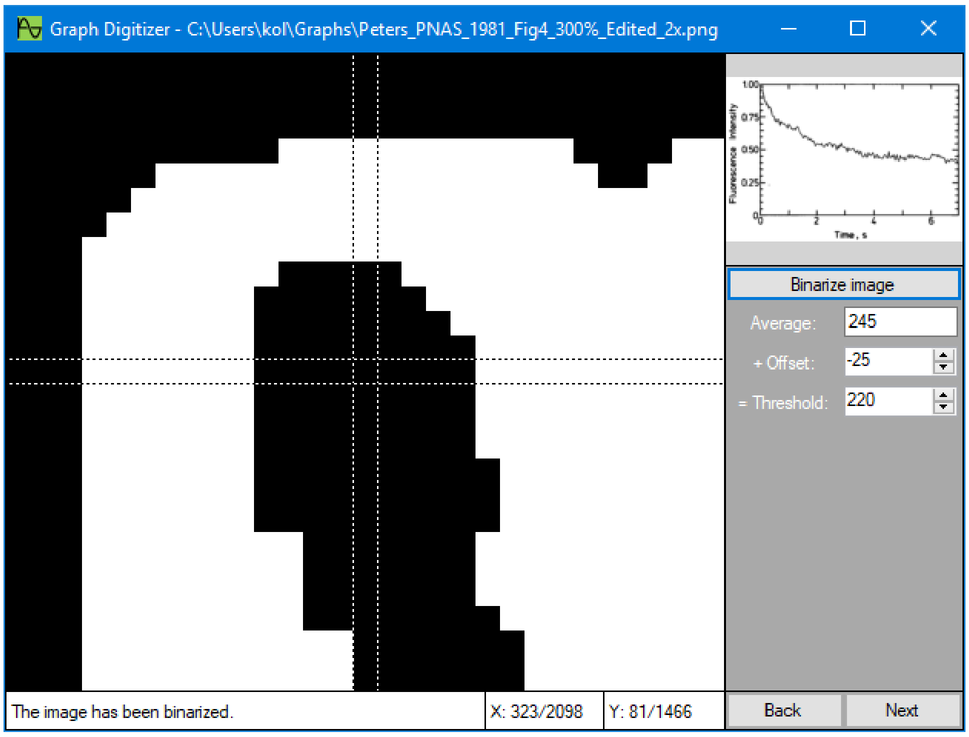
Figure 2.3: The test image loaded into Graph Digitizer

The "Next" button takes us to the next screen.

2.3.3 Step #2: Binarization

Clicking the "Binarize image" button converts the image to binary. Our test image is grayscale, but the algorithm works for color images as well, because those are converted to grayscale first. If the result of the automatic binarization is not satisfactory, then the binarization threshold can be fine-tuned by modifying the threshold, or the threshold offset, which is the difference of the threshold from the average pixel intensity. It is worth investigating the whole image before deciding if the selected threshold is optimal.

Figure 2.4 shows the test image after binarization, using the default threshold offset.


Graph Digitizer - C:\Users\kol\Graphs\Peters_PNAS_1981_Fig4_300%_Edited_2x.png

Average: 245

+ Offset: -25

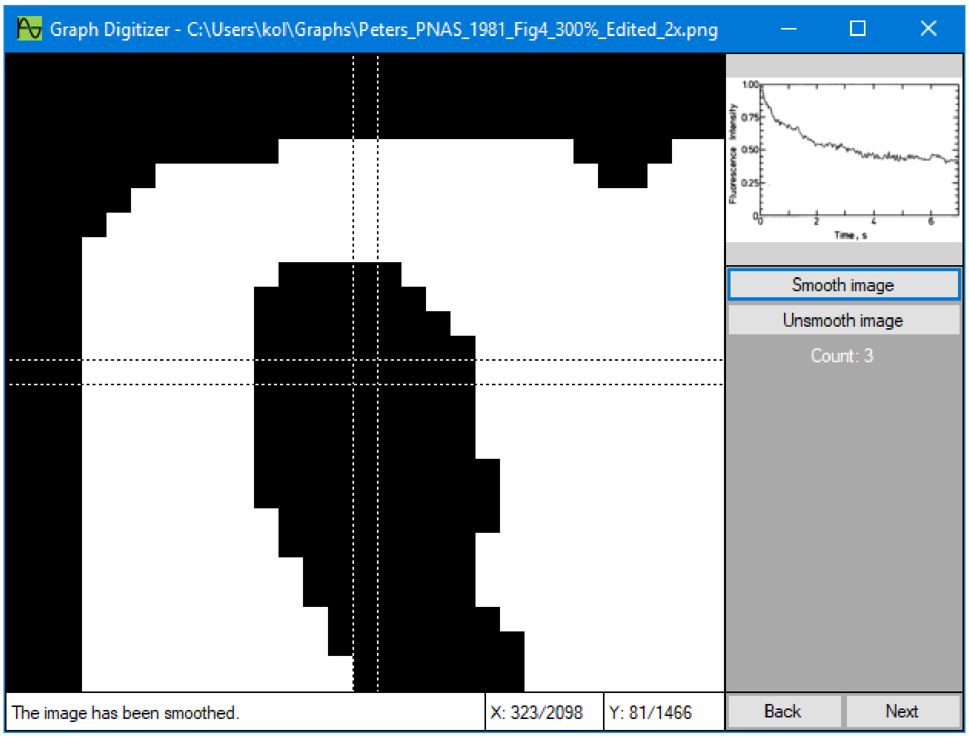
= Threshold: 220

The image has been binarized.

X: 323/2098

Y: 81/1466

Figure 2.4: The test image converted to binary


Graph Digitizer - C:\Users\kol\Graphs\Peters_PNAS_1981_Fig4_300%_Edited_2x.png

Count: 3

The image has been smoothed.

X: 323/2098

Y: 81/1466

Figure 2.5: The test image after smoothing

2.3.4 Step #3: Smoothing

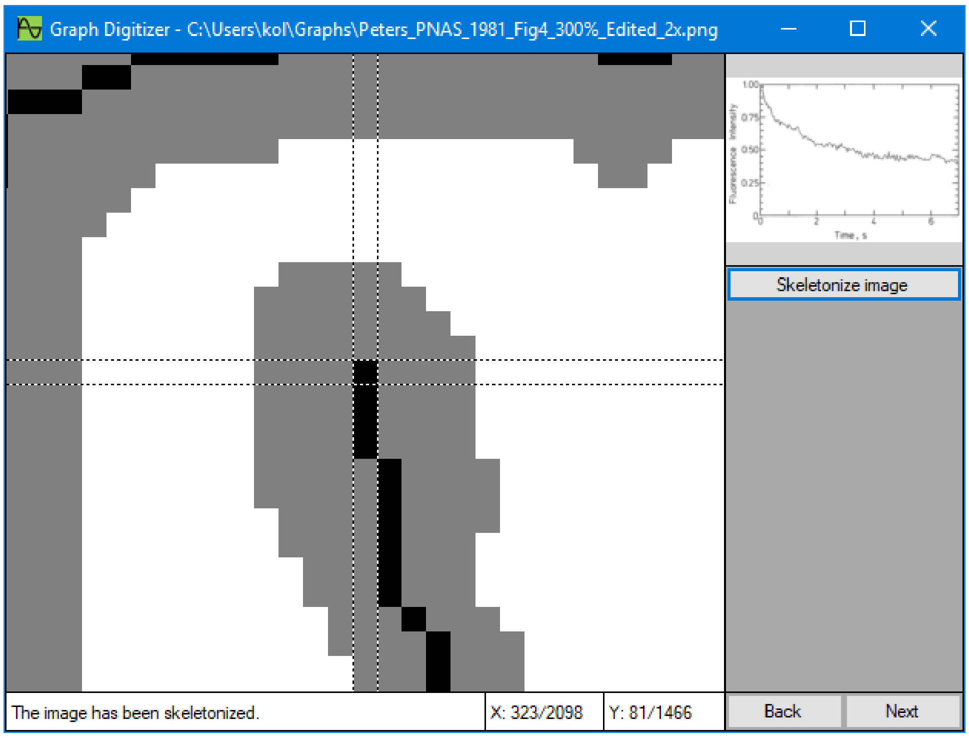
If the binarized image is noisy, that is, it has "furry" or jagged edges, then it needs to be smoothed. The smoothing algorithm simply makes those black pixels white which have 3 or less black neighbors, and similarly, makes those white pixels black which have 3 or less white neighbors. Smoothing is optional, but can be repeated a couple of times if needed.

Figure 2.5 shows the smoothed test image. Note that the left edge of the starting section of the curve is less jagged than it was in Figure 2.4.

2.3.5 Step #4: Skeletonization

Skeletonization "peels" black areas until only their 1-pixel-wide central line remains. Graph Digitizer displays the removed black pixels in gray, to help assessing the result of skeletonization. The algorithm is fixed, no parametrization is possible.

Figure 2.6 shows the skeletonized test image. The crosshairs target the starting point of the curve.


Graph Digitizer - C:\Users\kol\Graphs\Peters_PNAS_1981_Fig4_300%_Edited_2x.png
Skeletonize image
Back
Next

The image has been skeletonized.
X: 323/2098
Y: 81/1466

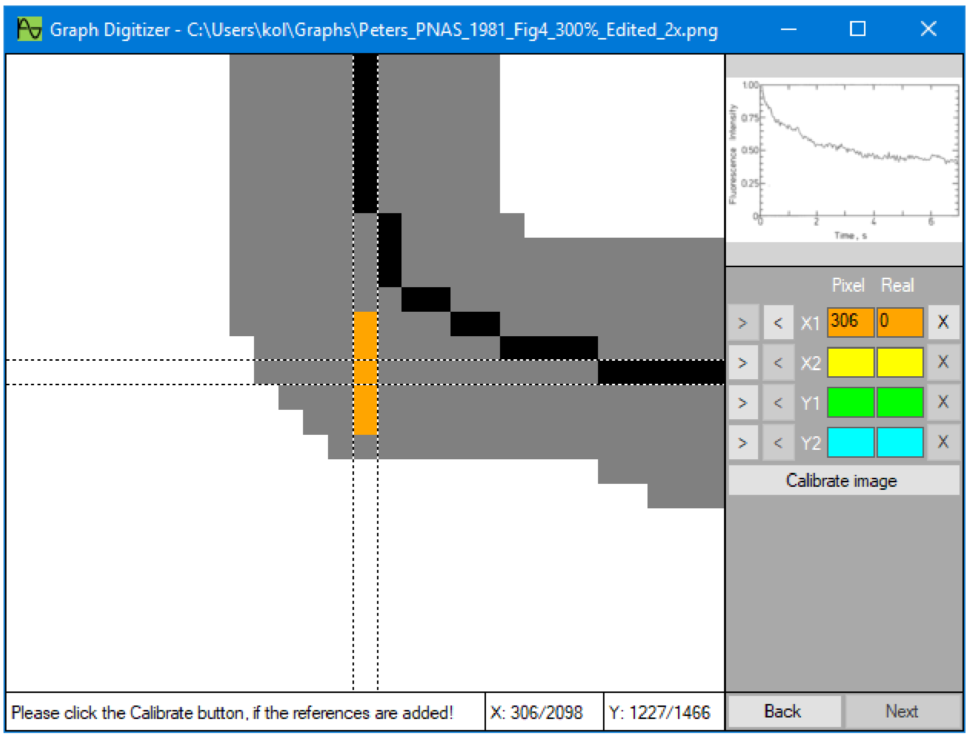
Figure 2.6: The test image after skeletonization

2.3.6 Step #5: Calibration

The pixels of the skeleton define the points of the function in pixel coordinates. If the image contains axes or scale bars, then these determine the distance of pixels in real, physical units. In other words, the image can be calibrated. There are two directions in the image: X (left-to-right) and Y (top-to-down). Each direction can be calibrated independently from the other, by selecting 2 different pixel positions (references) in this direction, and defining their value in real units. If there is an axis, then a pair of its tick marks can be chosen as references. Their real value is given by the corresponding tick labels. Note that the farther are the references from each other, the better is the precision. If there is only a scale bar, then its end points are the references. One of them has value 0, and the other has a value equal to the length of the scale bar. If calibration is not possible, or extracting pixel coordinates is enough, then calibration can be skipped in any or both directions.

Graph Digitizer provides a simple user interface for calibration. To put a reference tick into the image, the crosshairs should be moved to the desired position, and the ">" button should be pushed. The pixel value of the reference is filled automatically, but the real value (in proper physical units) should be typed into the table. The "<" button can be used to jump to a reference. The "X" button removes the reference from the image. If the required references are in place, the "Calibrate image" button must be clicked to validate the calibration parameters. If everything is OK, the "Next" button becomes active, otherwise a warning message is shown. Note that the "Next" button is also active when there are no references.

Figures 2.7, 2.8, 2.9 and 2.10 demonstrate the calibration process in case of our test image.


Graph Digitizer - C:\Users\kol\Graphs\Peters_PNAS_1981_Fig4_300%_Edited_2x.png

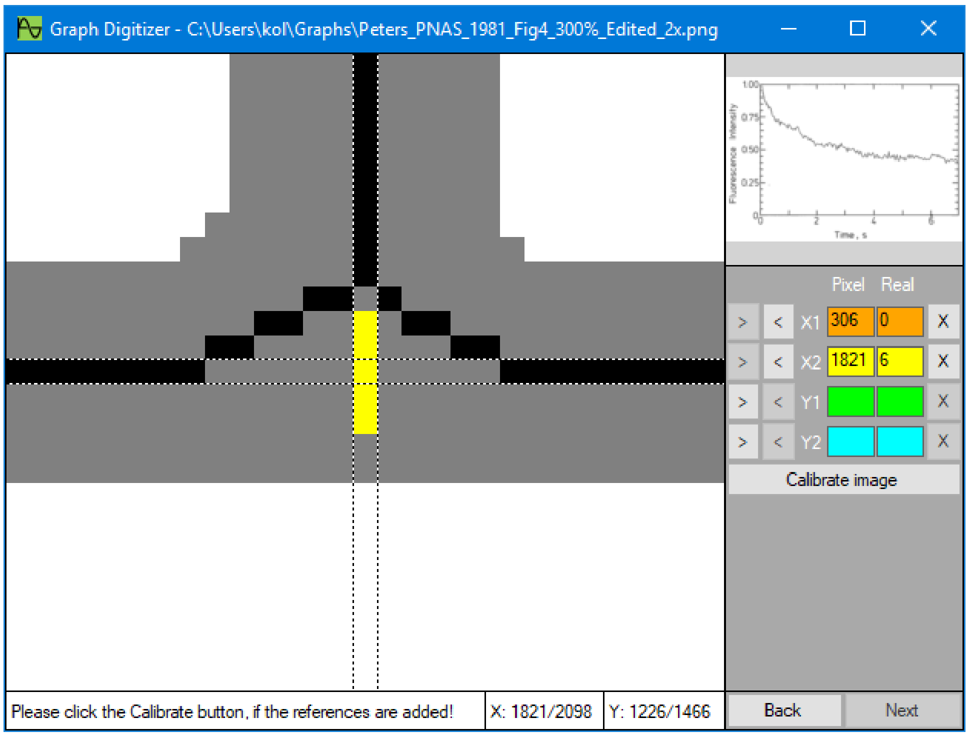
			Pixel	Real	
>	<	X1	306	0	X
>	<	X2			X
>	<	Y1			X
>	<	Y2			X

Please click the Calibrate button, if the references are added!

X: 306/2098

Y: 1227/1466

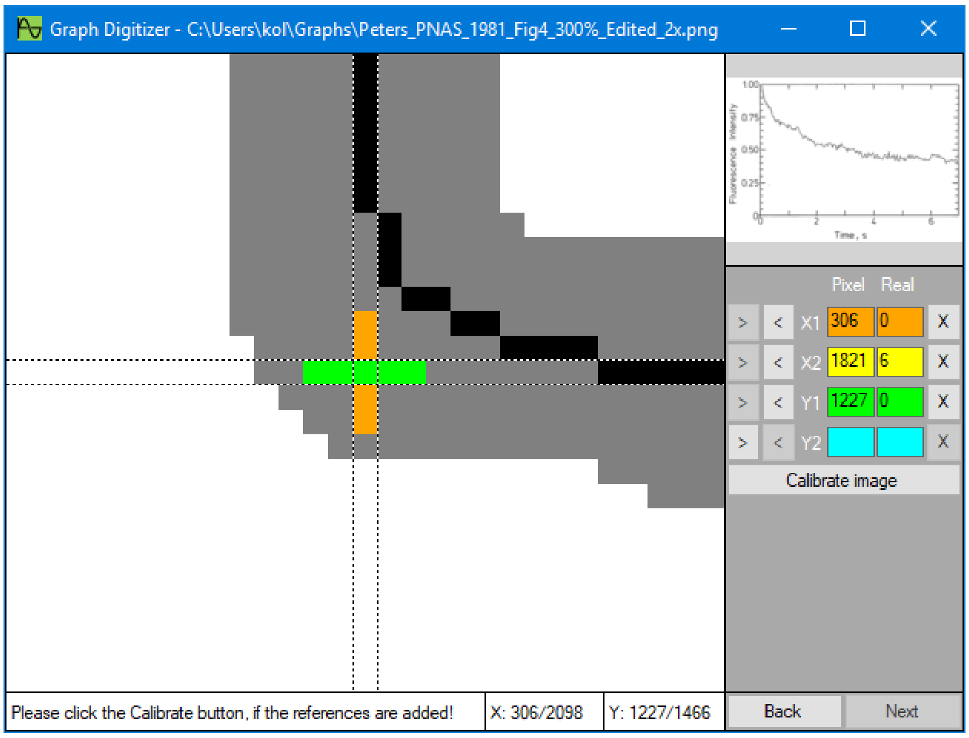
Figure 2.7: Test image calibration: Reference X_1 @ time = 0 s


Graph Digitizer - C:\Users\kol\Graphs\Peters_PNAS_1981_Fig4_300%_Edited_2x.png
Pixel
Real
X1
X2
Y1
Y2
Calibrate image
Back
Next

	Pixel	Real
X1	306	0
X2	1821	6
Y1		
Y2		

Please click the Calibrate button, if the references are added! X: 1821/2098 Y: 1226/1466

Figure 2.8: Test image calibration: Reference X_2 @ time = 6 s

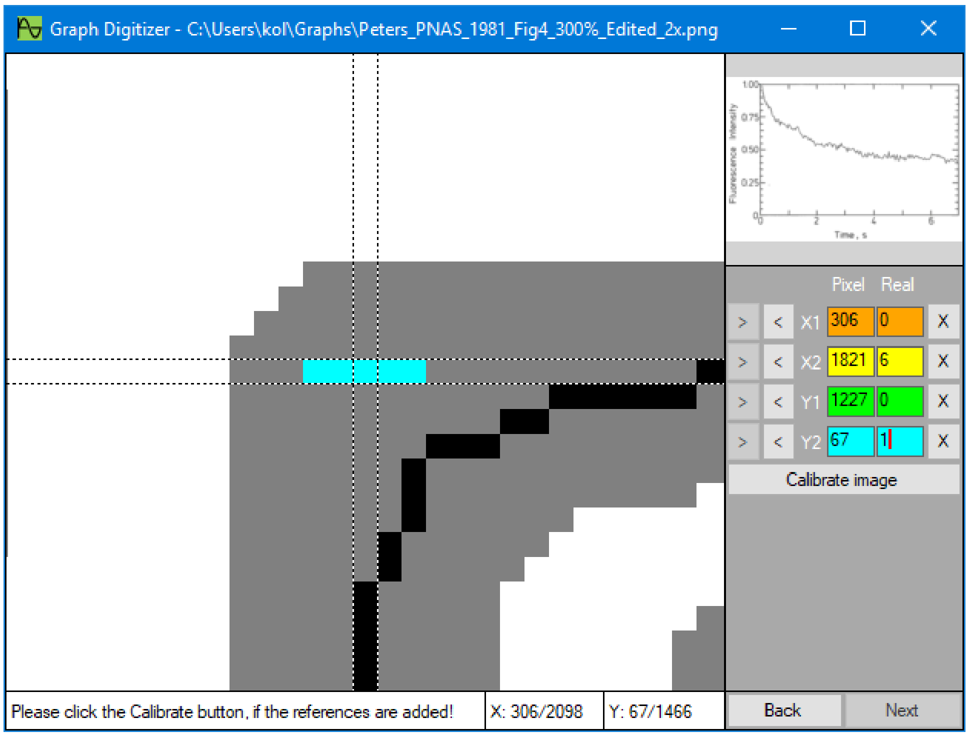

Graph Digitizer - C:\Users\kol\Graphs\Peters_PNAS_1981_Fig4_300%_Edited_2x.png
Pixel
Real
X1
X2
Y1
Y2
Calibrate image
Back
Next

	Pixel	Real
X1	306	0
X2	1821	6
Y1	1227	0
Y2		

Please click the Calibrate button, if the references are added! X: 306/2098 Y: 1227/1466

Figure 2.9: Test image calibration: Reference Y_1 @ fluorescence = 0

These images also show how the crosshairs can be used to find intersection points precisely, even if the skeleton of the axes is curved at corners and junctions.


Graph Digitizer - C:\Users\kol\Graphs\Peters_PNAS_1981_Fig4_300%_Edited_2x.png

	Pixel	Real
X1	306	0
X2	1821	6
Y1	1227	0
Y2	67	1

Please click the Calibrate button, if the references are added!

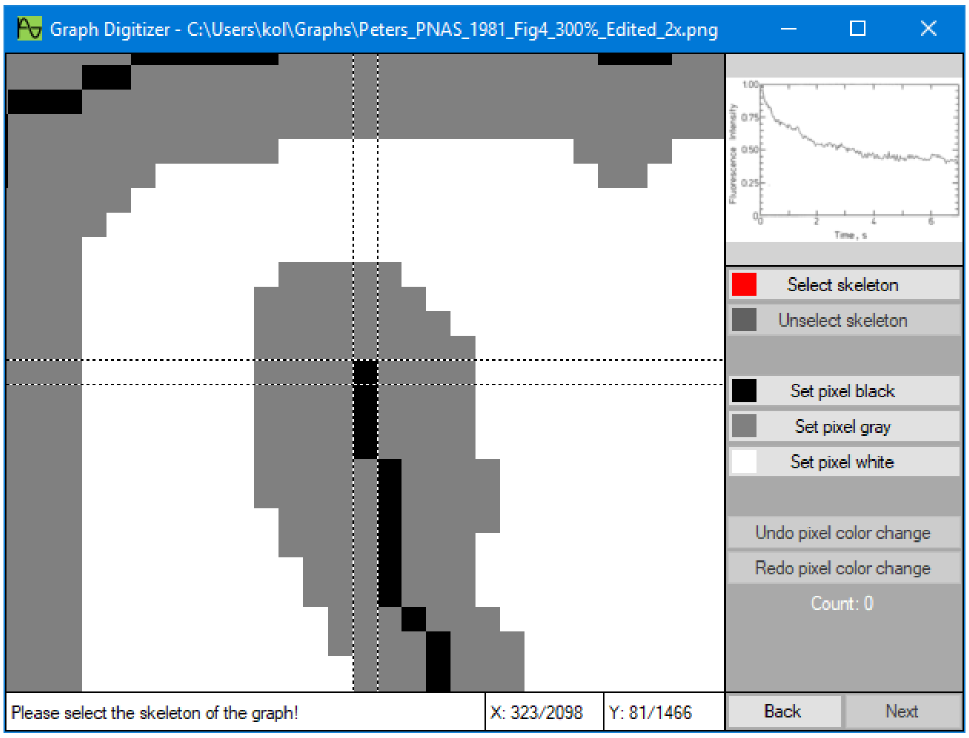
X: 306/2098

Y: 67/1466

Figure 2.10: Test image calibration: Reference Y_2 @ fluorescence = 1

2.3.7 Step #6: Selecting the skeleton

Before clicking the ”Select skeleton” button, the crosshairs must be positioned over the leftmost end of the curve (Figure 2.11).


Graph Digitizer - C:\Users\kol\Graphs\Peters_PNAS_1981_Fig4_300%_Edited_2x.png

Count: 0

Please select the skeleton of the graph!

X: 323/2098

Y: 81/1466

Figure 2.11: Selecting skeleton: Crosshairs at the starting point

The selection algorithm moves to the right from pixel to pixel, by scanning the neighboring black pixels in the order shown in Figure 2.12. If a black pixel is found, then it is regarded as the next skeleton pixel, and its color is set to red. The scanning continues until no more neighbors are found.

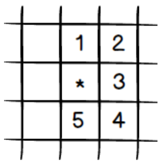
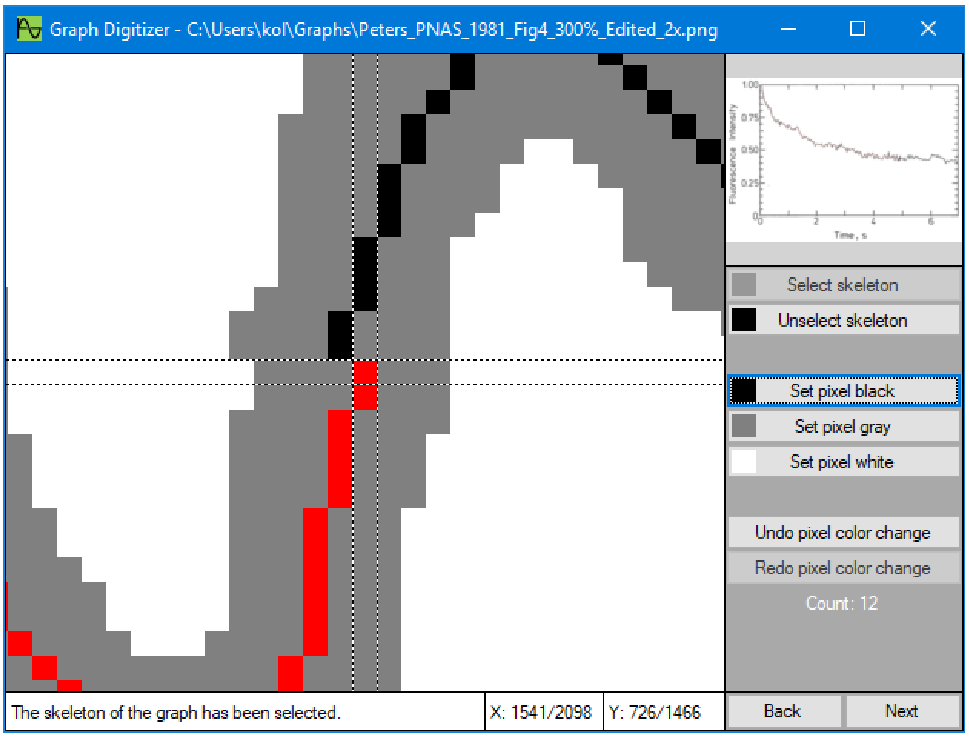


Figure 2.12: Pixel scanning order during skeleton selection

Since the skeletonization cannot be perfect, the selection process might get stuck somewhere in the middle of the curve. To handle these cases, pixels can be added to or removed from the image.

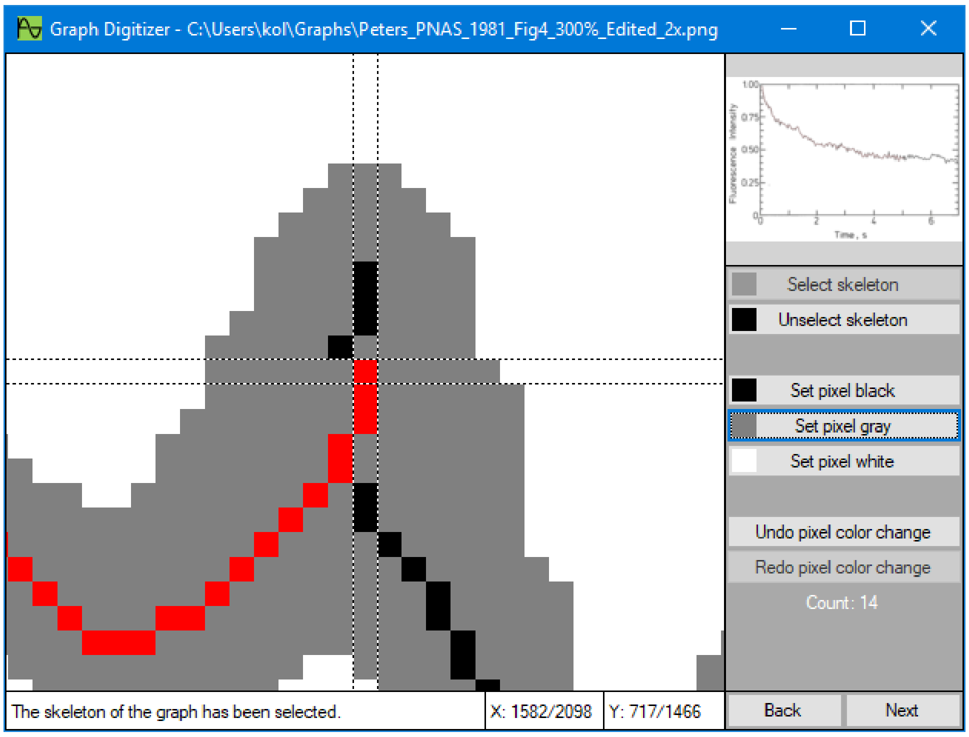
One typical problem is the meandering skeleton. This phenomenon puts some pixels out of the scanning range, which stops the selection process. The solution is building a "bridge" out of black pixels. In Figure 2.13, two black pixels should be added, right above the red end point. After these are added, the selection process runs forward automatically.



The skeleton of the graph has been selected.

Figure 2.13: Selecting skeleton: Meandering skeleton

Another typical problem is caused by side branches. The solution is cutting off the side branch at its base, by replacing a black pixel with a gray one. Figure 2.14 shows an example of this phenomenon.

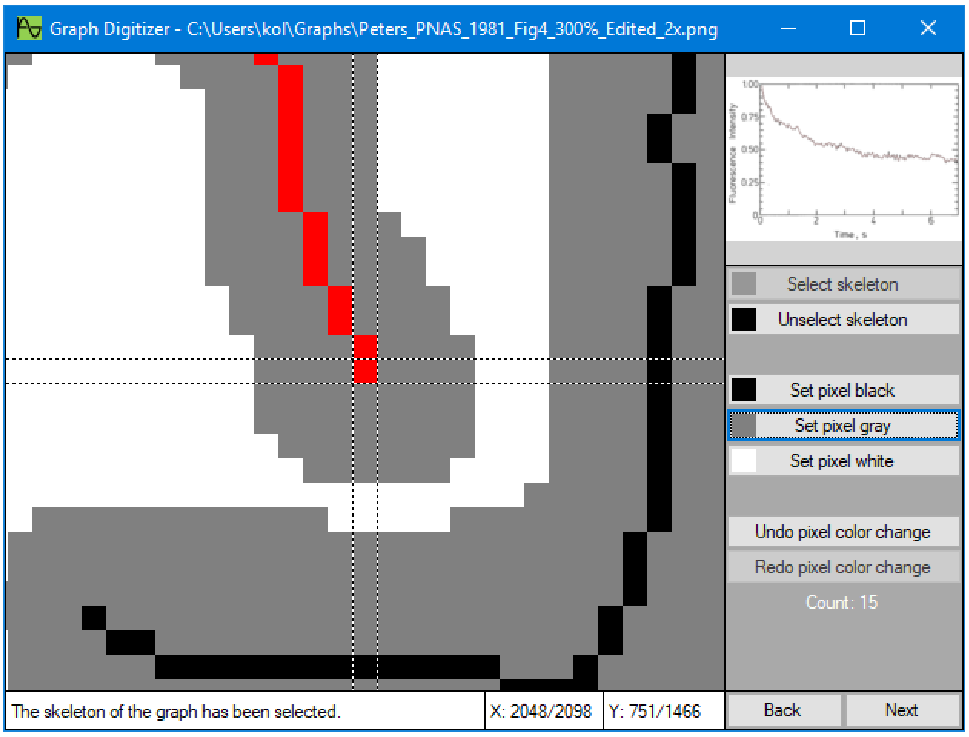


The skeleton of the graph has been selected.

X: 1582/2098 Y: 717/1466

Figure 2.14: Selecting skeleton: Side branch

Figure 2.15 shows that in case of the test image, 15 pixels had to be modified for the selection algorithm to be able to reach the end point of the curve.



The skeleton of the graph has been selected.

X: 2048/2098 Y: 751/1466

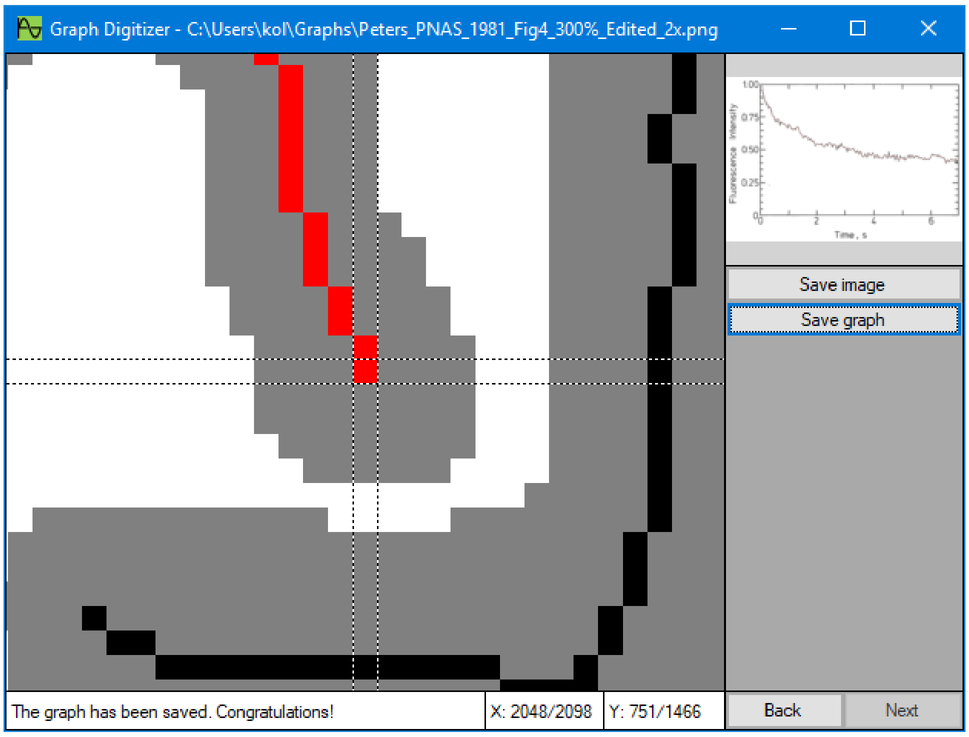
Figure 2.15: Selecting skeleton: Crosshairs at the end point

2.3.8 Step #7: Saving the result

After the skeleton of the graph is selected, its points can be saved. This involves 2 automatic steps:

- The first step is the "functionization" of the skeleton: replacing pixel columns with a single point, positioned at the middle of the column. This is important, because a function maps every element of its domain to only one element of its codomain.
- The second step is the calibration of the skeleton: replacing pixel values with real values, given in physical units. But this is done only if the image has been calibrated previously.

Figure 2.16 shows the last screen of the process, after the extracted graph has been saved. It is possible to save the processed image as well.


Graph Digitizer - C:\Users\kol\Graphs\Peters_PNAS_1981_Fig4_300%_Edited_2x.png

The graph has been saved. Congratulations!

X: 2048/2098

Y: 751/1466

Figure 2.16: Saving the graph extracted from the test image

The extracted graph is saved into a TXT file, as a TAB-separated pair of number columns. This file can be easily loaded into numerical software, like Microsoft Excel or Matlab, for further processing. Figure 2.17 is a plot of the extracted graph, created using Matlab.

With some practice, the processing of a single graph takes only a few minutes.

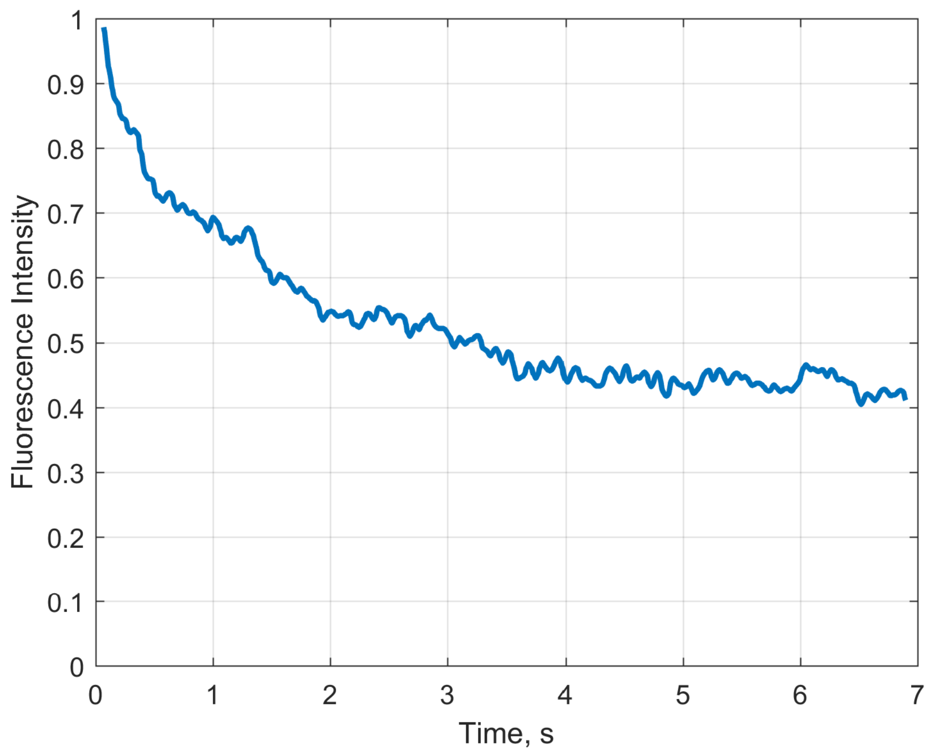


Figure 2.17: The extracted graph plotted in Matlab

2.3.9 Special case: Multiple graphs

Sometimes more than one curve has to be extracted from the same image. Figure 2.18 shows an example for this ([3], Figure 4).

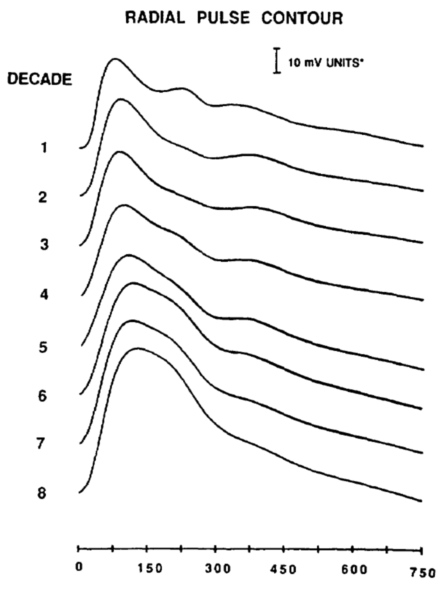


Figure 2.18: Test image with multiple graphs [3]

In such cases, graph digitization is performed the same way as above, but the last two steps have to be repeated to select and save every curve. This is why image calibration has to happen before skeleton selection: to be able to use the same calibration data for every graph.

Figure 2.19 displays a plot of all extracted graphs, created using Matlab.

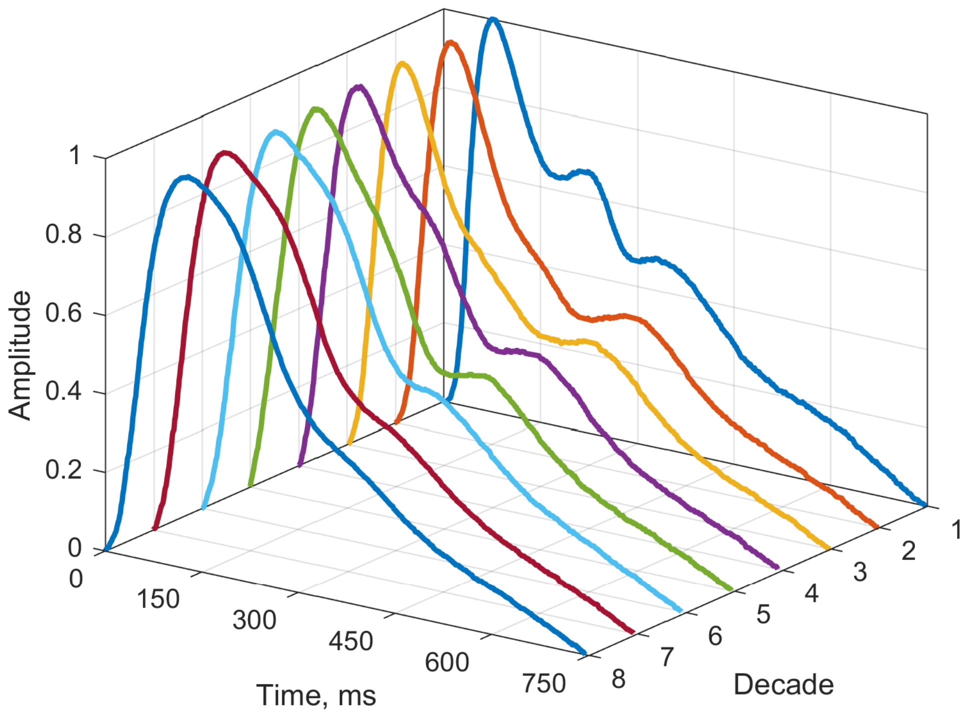


Figure 2.19: The extracted graphs plotted in Matlab

2.4 Troubleshooting

If the application is started without installing the prerequisites, an error message will appear:

- If the .NET Framework is not installed: "The application requires one of the following versions of the .NET Framework: ..."
- If the Visual C++ Redistributable is not installed: "The program can't start because VCRUNTIME140.dll is missing from your computer"

If the application encounters an unforeseen error, a message box will appear, and the program will stop. The message box shows the path of the error log file, which contains important details about the problem.

Chapter 3

Developer Documentation

3.1 Functional requirements

Graph Digitizer is a desktop application for scientists, to digitize graphs of continuous functions. The steps of the digitization process are the following:

The user loads the image of the graph.

The input of the program is a digital image, depicting the graph to be digitized. It is enough to support 24-bit RGB images in BMP, JPG, TIF and PNG format, and the curve of the graph can be expected to be isolated from other parts of the image. Lots of image editors are available to convert between different image types and to clean up the surroundings of the curve.

The user skeletonizes the image.

Graphs of continuous functions are lines, which must be automatically skeletonized, that is, thinned to get their one-pixel-wide center line. During the iteration of the software, multiple sub-steps were identified:

The user converts the image to binary.

Using a darkness threshold provided by the user, the pixels of the image are automatically turned into white background and black foreground pixels. It is enough to handle dark graphs on light background, because this is the usual case. In the reverse case, an image processor program can be used to invert the colors of the image.

The user makes the binary image smoother.

The edge of the graph, that is, the border between white and black areas, must be smooth enough to help the skeletonization algorithm find the center line. Smoothing is also an automatic process, which can be repeated, if needed.

The user runs the skeletonization algorithm.

The skeletonization process turns the black pixels of the binary image into gray, except for the pixels of the center line, which remain black.

The user calibrates the image.

Graphs usually contain axes or scale bars. After the skeletonization step, these will also be one pixel wide, so the pixel coordinates of ticks or length of scale bars can be precisely determined. If the user specifies both a pixel distance, and a corresponding value in physical units (seconds, milliVolts, etc.), the software must be able to convert pixel coordinates to physical units. The software must support both horizontal (X) and vertical (Y) calibration.

The user selects the skeleton of the graph.

The skeletonization algorithm finds the center line of everything on the image, so the skeleton belonging to the graph in question must be selected manually. The user must be able to mark the pixel at the left end of the graph, and instruct the software to select the whole graph automatically. The software must display the selected skeleton in a special color (say, red). If the selection is not complete because of a side branch or a discontinuity, the user must be able to fix the skeleton manually.

The user saves the result.

The main result of the digitization process is the selected skeleton, but saving the processed image itself can also be useful sometimes.

The user saves the extracted function points.

If the user is satisfied with the selected skeleton, they must be able to save the points of the function into a TAB-separated text file. The points of the function correspond to the pixels of the skeleton, except for two things: if the user specified calibration data, then the pixel coordinates must be converted to physical units (calibration); and vertical columns of pixels in the skeleton must be replaced by a single function point, positioned at the middle height of the column (”functionization”).

The user saves the skeletonized image.

This option is useful to debug image processing problems, if the user fails to digitize a graph.

It is very important that the user must be able to **undo** every step of the digitization process. The UML use case diagram for the software is shown in Figure 3.1.

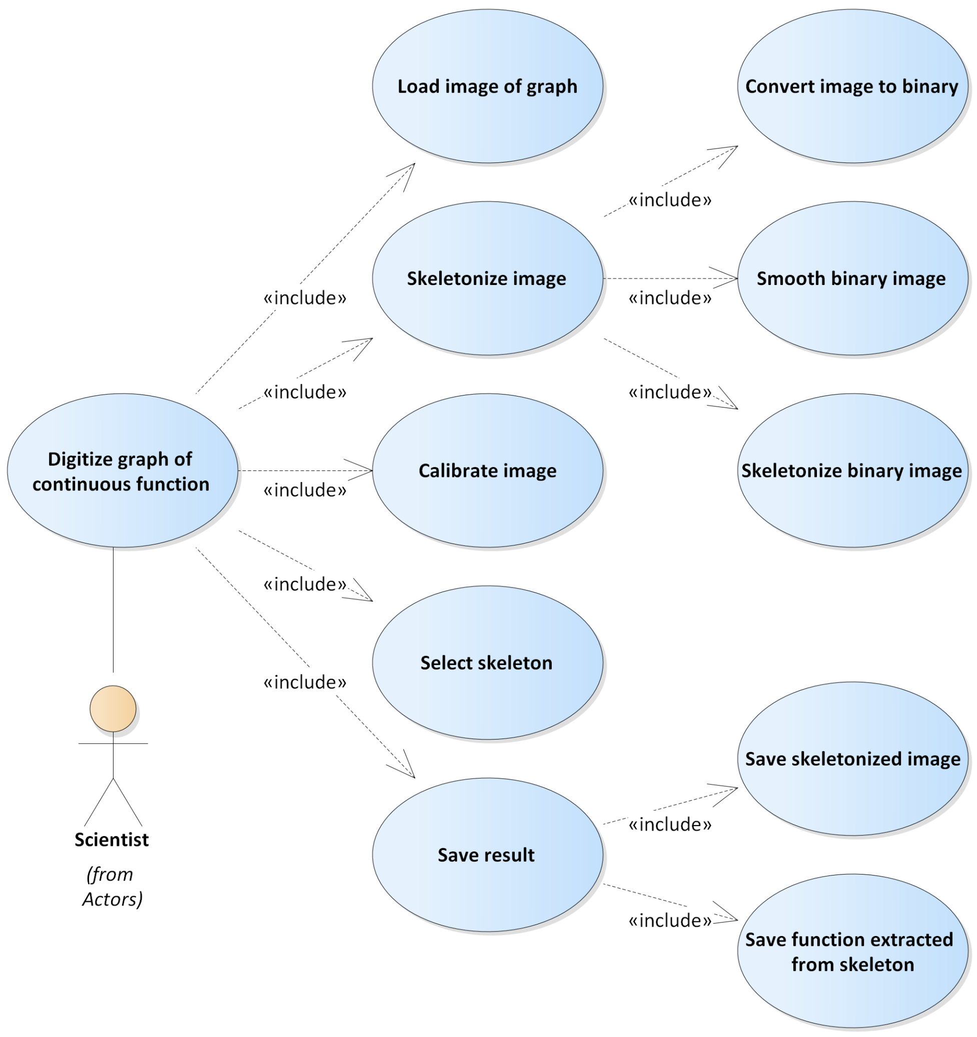


Figure 3.1: UML use case diagram

3.2 Technical requirements

Target environment: Microsoft Windows

Since Microsoft Windows is by far the most widespread desktop operating system, it is enough to develop the software for Windows. Windows 7 is still widely used, so this version must also be supported.

Programming language: C++

Scientific applications are usually performance-critical, and therefore frequently written in C++. Although the performance requirements for Graph Digitizer are not high, using C++ definitely does not hurt.

Application framework: Microsoft .NET

One of the best frameworks for writing desktop applications for Windows is the .NET Framework. The .NET Framework supports C++ development via an extended version of ISO C++, called C++/CLI. In addition to the comprehensive class library, C++/CLI programs can also utilize the memory management, type safety, garbage collection and exception handling features of .NET. It is especially important that exceptions contain call stack information, which helps debugging immensely. Although these features come with some performance overhead, C++/CLI programs may contain native C++ code as well (using the `#pragma unmanaged` directive).

Graphical User Interface

Direct image manipulation requires a Graphical User Interface (GUI). The .NET Framework provides two built-in GUI options: Windows Forms and Windows Presentation Foundation. The latter is more advanced, but for the purposes of Graph Digitizer, Windows Forms is acceptable.

Free and open source code

Every software tool created for scientists should be free and open source. To achieve this, the code of Graph Digitizer is licensed under GNU GPL 3, and can be freely downloaded from the following GitHub repository: `https://github.com/koldev/GraphDigitizer`

Development environment

Graph Digitizer is modeled and designed in Enterprise Architect 13.5 and Balsamiq Mockups 3.5.15. The code is developed using Microsoft Visual Studio Community 2017 15.7.1, for .NET Framework 4.7.1. Git 2.17.0 is used for version control. Minimal hardware requirements for development: any 2-core, 2 GHz CPU and 4 GB RAM.

Test environment

All testing can be done on Windows 10, except for the end-to-end tests, which have to be run on every supported Windows version (7, 8.1 and 10). From a testing point of view, virtual machines are as good as physical computers.

3.3 User interface

3.3.1 Idea #1: Image editor

The Graph Digitizer application can be regarded as a special kind of image editor. The simplest GUI for an image editor would contain a menu bar and a tool bar at the top, a status bar at the bottom, and an editor area in between. Here is a possible menu structure:

- File menu
  - New*: Clear the editor area. If there was an image open, ask the user about saving it first.
  - Open*: Open an image of a graph.
  - Save*: Save the edited image.
  - ——————————
  - Exit: Stop the application. If there was an image open, ask the user about saving it first.
- Edit menu
  - Undo*: Undo the last action.
  - Redo*: Redo the last action.
  - ——————————
  - Binarize*: Convert the image to binary.
  - Smooth*: Smooth the binary image.
  - Skeletonize*: Skeletonize the binary image.
  - ——————————
  - Settings: Open the Settings dialog box (display options, algorithm parameters, etc.).
- Skeleton menu
  - Select*: Select the skeleton of the graph by the next click.
  - Unselect*: Remove selection.

  - Calibrate*: Open the Calibration dialog.
  - ——————————————
  - Save*: Save the selected skeleton.
- View menu
  - Zoom In*: Zoom in on the edited image.
  - Zoom Out*: Zoom out of the edited image.
- Help menu
  - Help: Open the help file or website of the application.
  - About: Open the About box.

The menu items with the asterisk (*) are good candidates to have a corresponding tool bar button. After zooming in on the editor area, pixel colors could be changed by selecting the color on a toolbar and clicking on the image. After clicking the Select menu item, the first click on the image would mark the leftmost pixel of the skeleton. Starting from this pixel, the selection algorithm would move along the skeleton as far to the right as possible, marking the visited pixels by changing their color to red. Without going into more details, it is clear that this approach would work.

But there is a huge problem with implementing the Graph Digitizer as an image editor: it is not obvious for the user what steps should be taken to digitize the graph. The enabled/disabled state of menu items and tool bar buttons could be controlled by a state machine, which stops the user from performing invalid operations, like smoothing an already skeletonized image, but this would not make the user interface intuitive.

3.3.2 Idea #2: Wizard

The most important property of graph digitization is that it is a process, a series of well-defined steps. The usual user interface for guiding the user through a series of steps, is the so-called wizard. Wizard applications usually display a sequence of different dialog boxes, each having "Back" and "Next" buttons, except for the first one, which only has a "Next" button, and the last one, which usually has a "Finish" button, in addition to the "Back" button.

The wizard paradigm solves the nonlinearity problem of the image editor design, but by using separate dialog boxes, it loses the advantage of having a single, unchanging editor area.

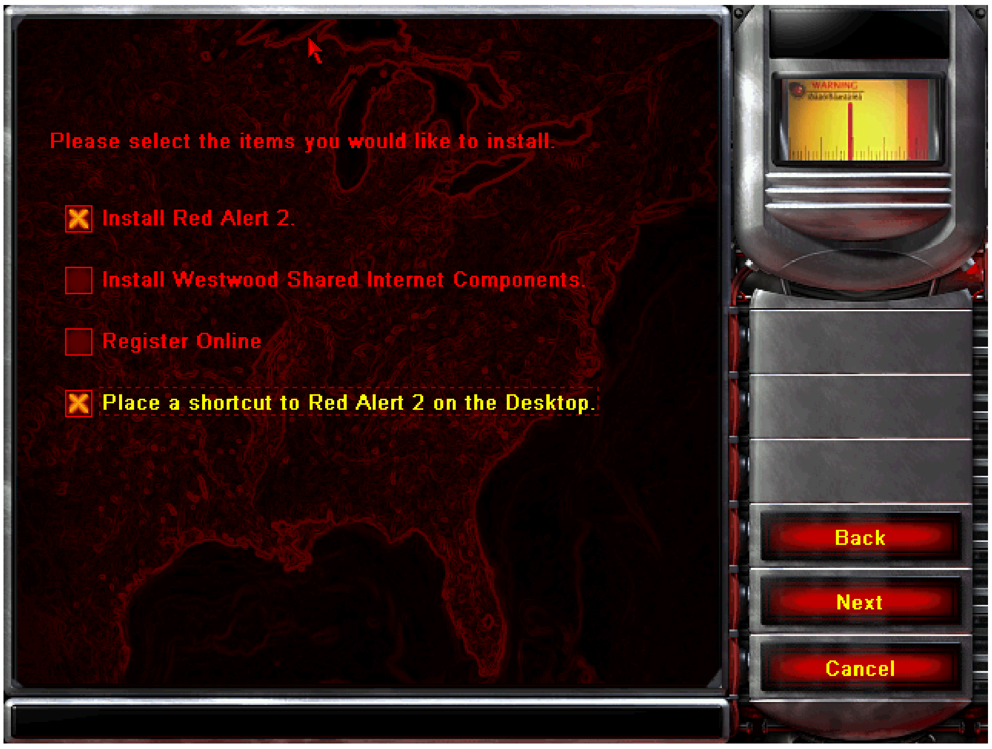
3.3.3 Idea #3: Red Alert 2

Command & Conquer: Red Alert 2 was a famous real-time strategy game of the early 2000's. It was famous not just for the fantastic gameplay, but also for its well designed user interface [4].

The Red Alert 2 GUI was based on some powerful ideas, which served as an inspiration for designing Graph Digitizer.

Installer screen: wizard

Figure 3.2 shows one step of the installer. The central panel was used to enter the information needed by the installer. The control panel on the right contained the usual "Back" and "Next" buttons. In addition to building the linear installer wizard, this structure was used for hierarchical menus as well: with input controls on the central panel, and menu buttons on the control panel. The user could dig down into the menu hierarchy, but there was always a "Back" button to go one level up.


Please select the items you would like to install.

[✓] Install Red Alert 2.

[ ] Install Westwood Shared Internet Components.

[ ] Register Online

[✓] Place a shortcut to Red Alert 2 on the Desktop.

Figure 3.2: Red Alert 2 installer screen

Game screen: "image editor" with minimap

Figure 3.3 shows the game screen. The central panel was used to manipulate units (troops, vehicles and buildings) with the mouse. The control panel provided lots of different tools for this. This is conceptually similar to an image editor, where the user selects tools to manipulate pixels and shapes on the image displayed in the central area. The central area displayed only a small, magnified portion of the battlefield. There were a small map of the whole battlefield in the upper-right corner, above the control panel. Clicking on it moved the portion visible in the central area (the viewport) to the clicked position. Such minimaps are often used in games, and also in text editors, even in Enterprise Architect. In all these cases the user works on a small part of a large object, and the minimap helps them navigating within the object.

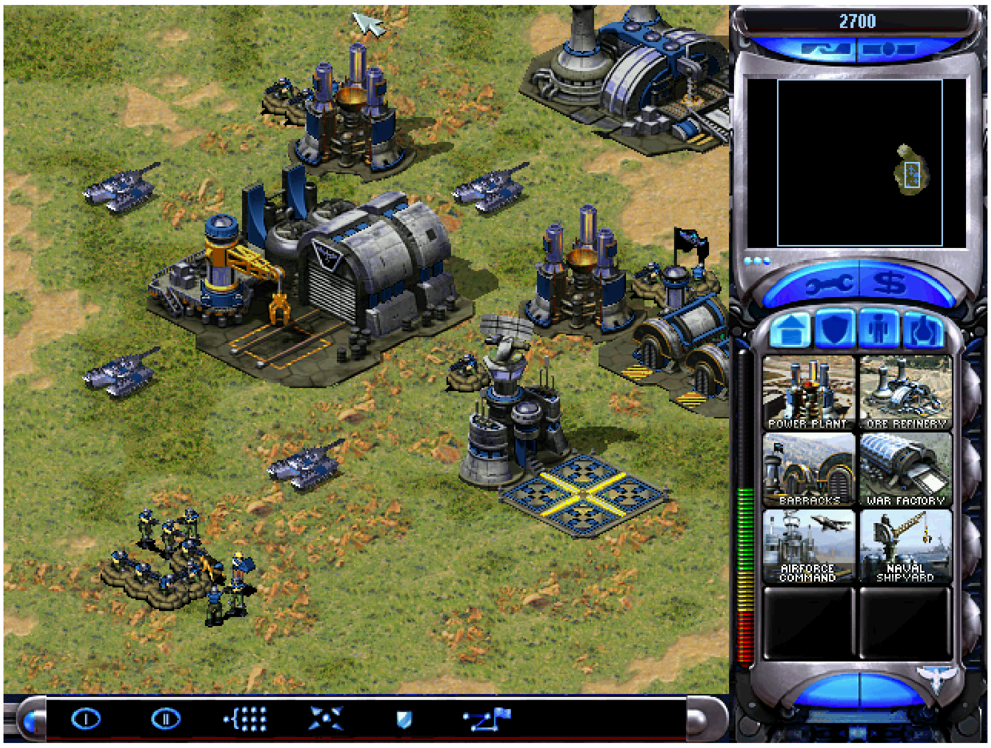


Figure 3.3: Red Alert 2 game screen

The Graph Digitizer GUI was designed to be a combination of these ideas: it is a wizard, with an image editor area, and a minimap.

3.3.4 Wireframes

Figure 3.4 shows the wireframe of the application's main form. The title bar contains the application's name ("Graph Digitizer"), and the absolute path of the loaded image. The client area of the form is occupied by the main panel. The main panel provides the layout for five sub-panels:

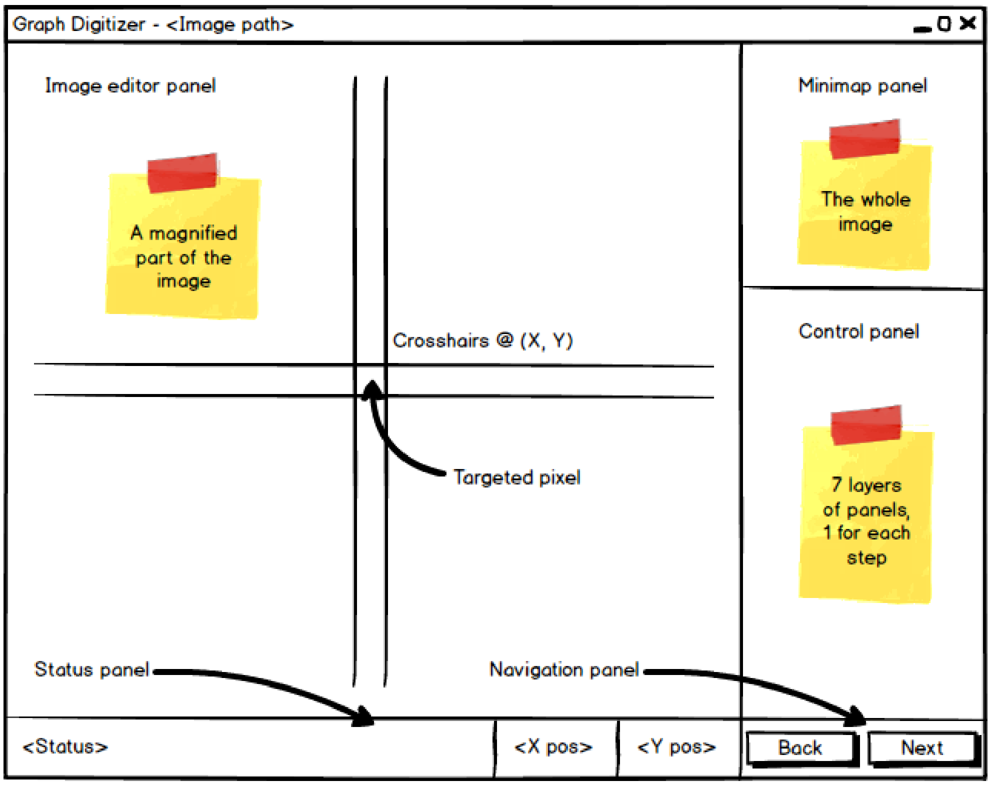


Figure 3.4: The main form

Image editor panel

The image editor panel displays a magnified portion of the loaded image, and the crosshairs, which target a single pixel.

Minimap panel

The minimap panel displays the whole image, and a semi-transparent rectangle, which shows the position and size of the magnified part, relative to the whole.

Status panel

The status panel is composed of three labels: the status label, and the two labels which display the crosshairs' X and Y position in pixel coordinates.

Control panel

The control panel holds a stack of seven sub-panels, one for each step of the graph digitization process (see below).

Navigation panel

The navigation panel contains the "Back" and "Next" buttons, which are needed for the wizard functionality.

Figure 3.5 shows the wireframe of the seven sub-panels of the control panel:

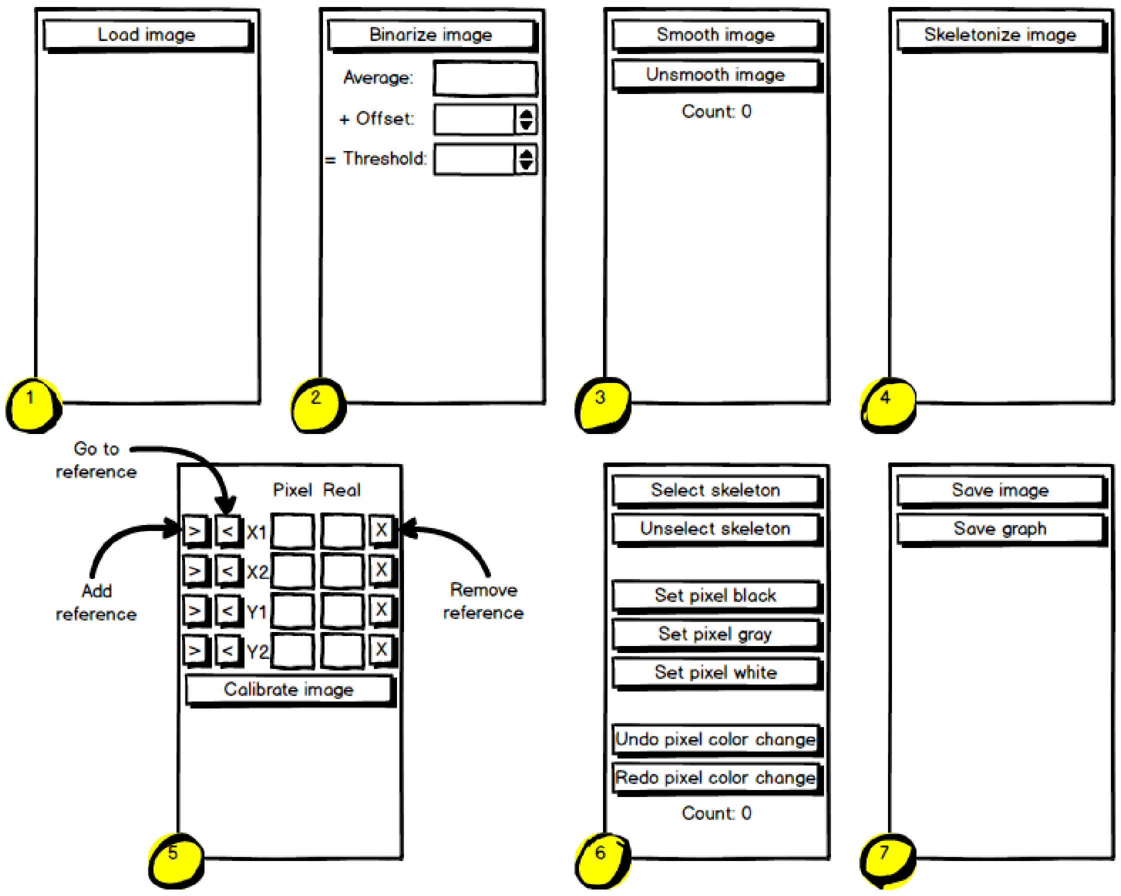


Figure 3.5: The sub-panels of the control panel

Panel #1: Image loading

The image loading panel has a single button to load an image of a graph.

Panel #2: Image binarization

The image binarization panel has a button to start converting the image to binary, and two up-down controls to fine-tune the binarization threshold, or its offset relative to the average intensity of the image.

Panel #3: Image smoothing

The image smoothing panel offers a button to smooth an image. The smoothing can be repeated by clicking this button again. If the result is too smooth, the number of smoothing rounds can be decreased by clicking the second button of this panel.

Panel #4: Image skeletonization

The image skeletonization has a single button to skeletonize the image.

Panel #5: Image calibration

The image calibration panel contains a table of controls. The first pair of rows can be used to calibrate the image in the X direction. Clicking the first two ">" buttons add the X_1 and X_2 references to the image. The "Pixel" text boxes are there to store their locations in pixel coordinates, and the "Real" text boxes can be used to enter their real value, in physical units. The references can be found easily by clicking the "<" button, and removed using the "X" button. The second pair of rows can be similarly used to define the Y_1 and Y_2 references, in order to calibrate the image in the Y direction.

Panel #6: Skeleton selection

The skeleton selection panel has a button at the top to start selecting the skeleton. If the selection algorithm gets blocked, then the lower five buttons can be used to fix the skeleton manually. The unselection option comes in handy when there are multiple curves to be extracted from the image.

Panel #7: Result saving

The result saving panel has buttons to save the extracted graph and the processed image.

3.4 Application behavior

Figure 3.6 shows the bird's eye view, the level-1 UML state machine diagram of Graph Digitizer.

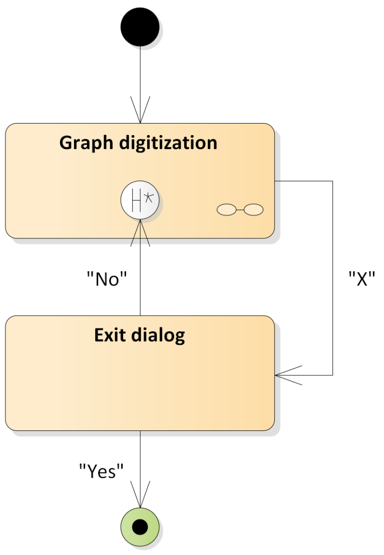


Figure 3.6: Level-1 UML state machine diagram

The main state of the application is the working state, when the user performs graph digitization. It can be left by clicking the "X" button of the main form, which displays the modal ”Exit” dialog box. This asks the user to choose between closing the application and resuming work.

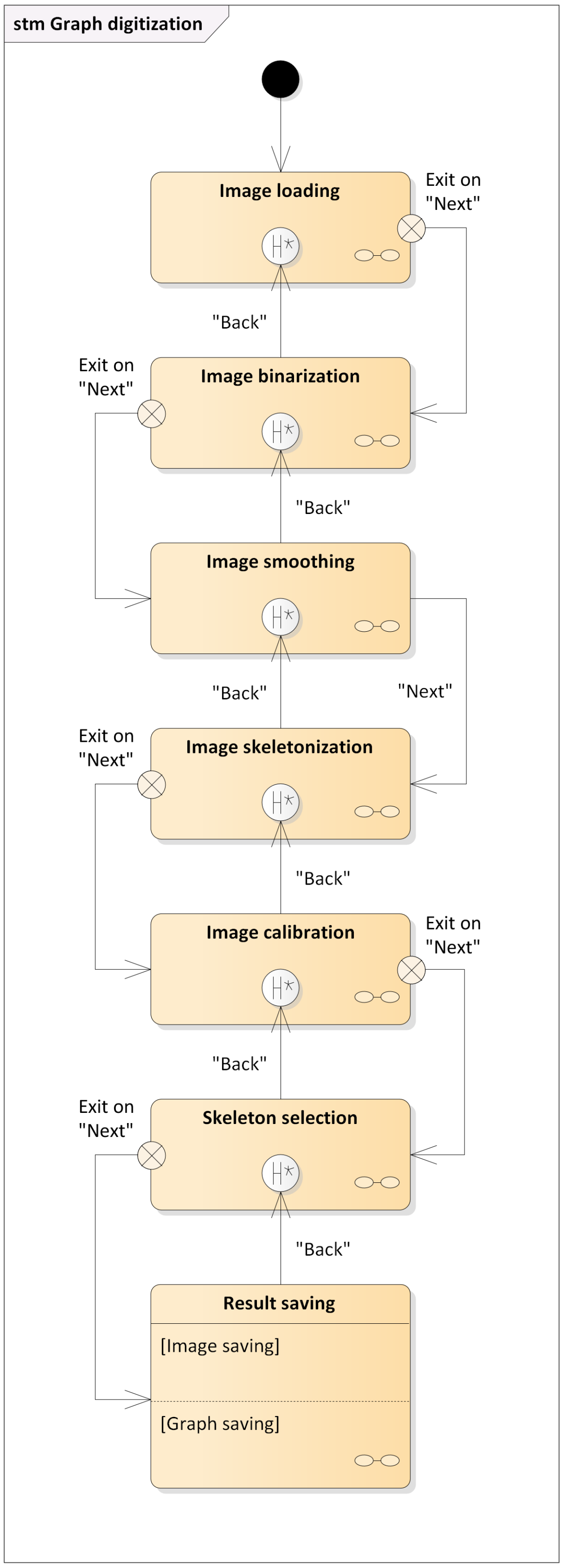


Figure 3.7: Level-2 UML state machine diagram

Figure 3.7 reveals the internals of the ”Graph digitization” composite state. This diagram shows seven states, each belonging to one of the graph digitization steps. Generally, clicking the ”Next” and ”Back” buttons triggers the transition between adjacent states.

Clicking ”Next” in itself is enough to leave the current state only in case of ”Image smoothing”, because smoothing is an optional step. All seven states are composite, and in case of the states other than ”Image smoothing”, the presence of the exit point implies that the software must be in a proper sub-state to be able to receive the ”Next” event, and transition to the next state. Naturally, these sub-states are the ones which are reached when the work of the actual processing step is finished.

Clicking ”Back” always triggers a transition from the current state to the deep history state of the previous one, irrespectively of the current internal state. This means that the ”Back” button effectively revokes every change that was made during the actual step. This is the expected behavior of wizard applications.

Opening up the seven composite states, we arrive at the level-3 UML state machine diagrams, which explain the logic of each graph digitization step.

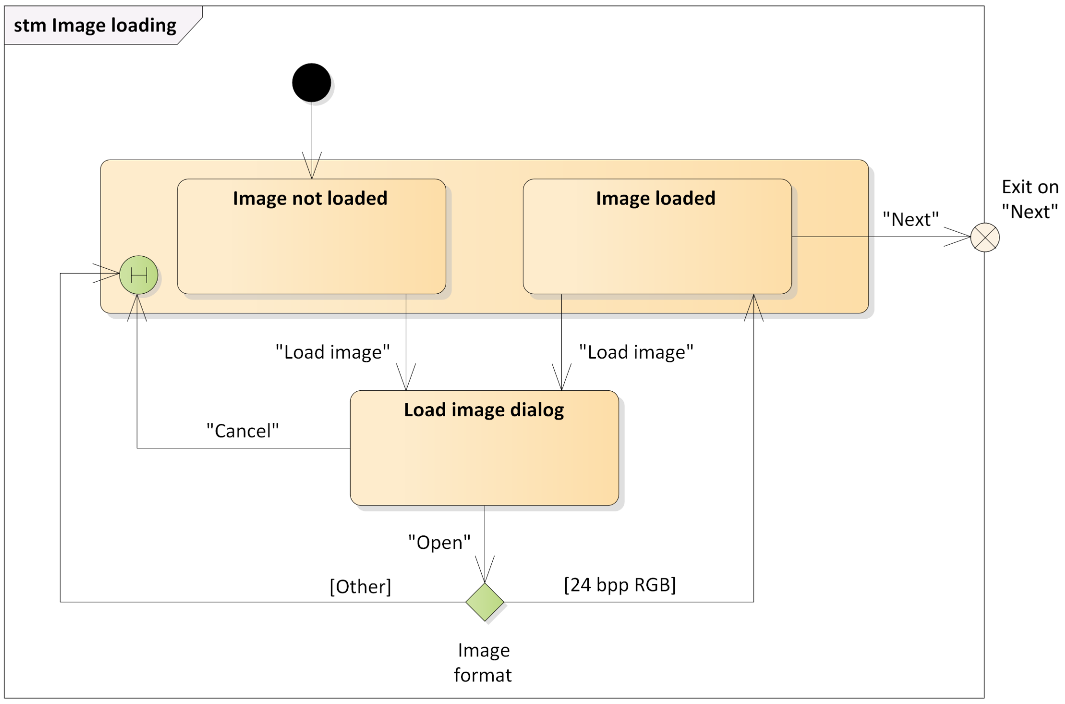


Figure 3.8: Level-3 UML state machine diagram of image loading

Figure 3.8 shows how image loading should work. Initially the image is ”Not loaded”. It becomes ”Loaded” only if the user selects a proper 24 bits-per-pixel RGB image. An already loaded image can be replaced by loading another image, if needed.

Moving forward from the "Image loading" composite state is only possible if the image is "Loaded".

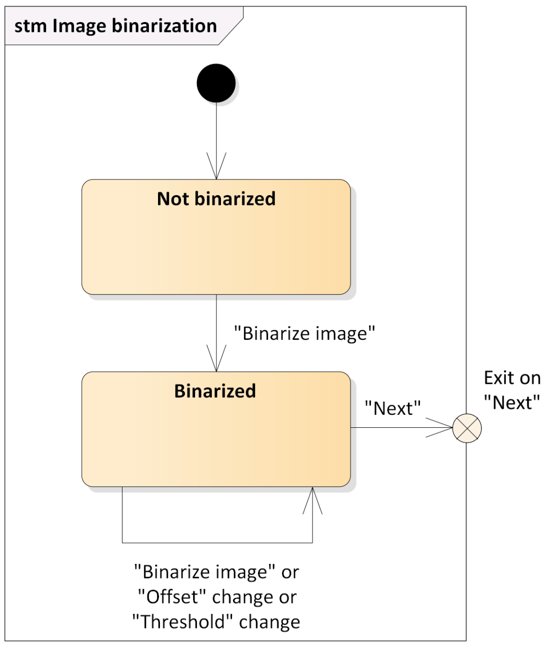


Figure 3.9: Level-3 UML state machine diagram of image binarization

Figure 3.9 shows that after binarizing the image the first time, it can be binarized again with modified parameters. This step is mandatory: the "Image binarization" state can only be left after the image has been "Binarized".

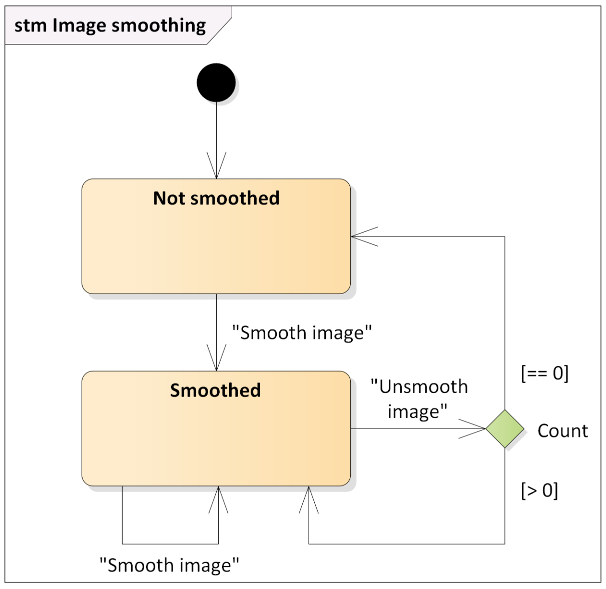


Figure 3.10: Level-3 UML state machine diagram of image smoothing

Figure 3.10 shows how the image smoothing step works: the image can be smoothed any number of times (including zero!), and the smoothing rounds can be reverted, back and forth, until the user is satisfied with the result.

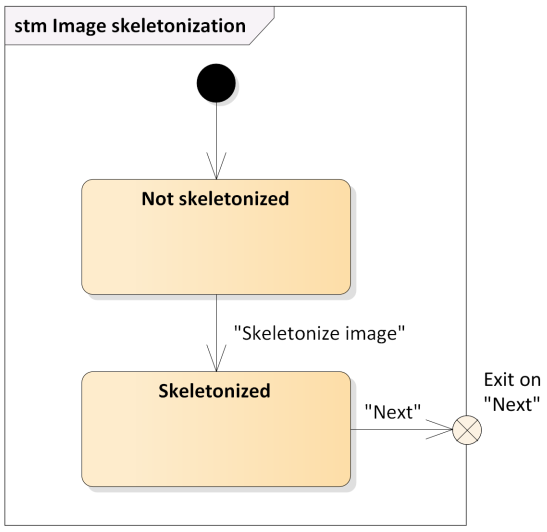


Figure 3.11: Level-3 UML state machine diagram of image skeletonization

Figure 3.11 shows the apparent simplicity of skeletonization. The algorithm is fixed, there is nothing to parameterize, undo or redo. Clicking the "Skeletonize image" button simply does its job. After that, the user can proceed.

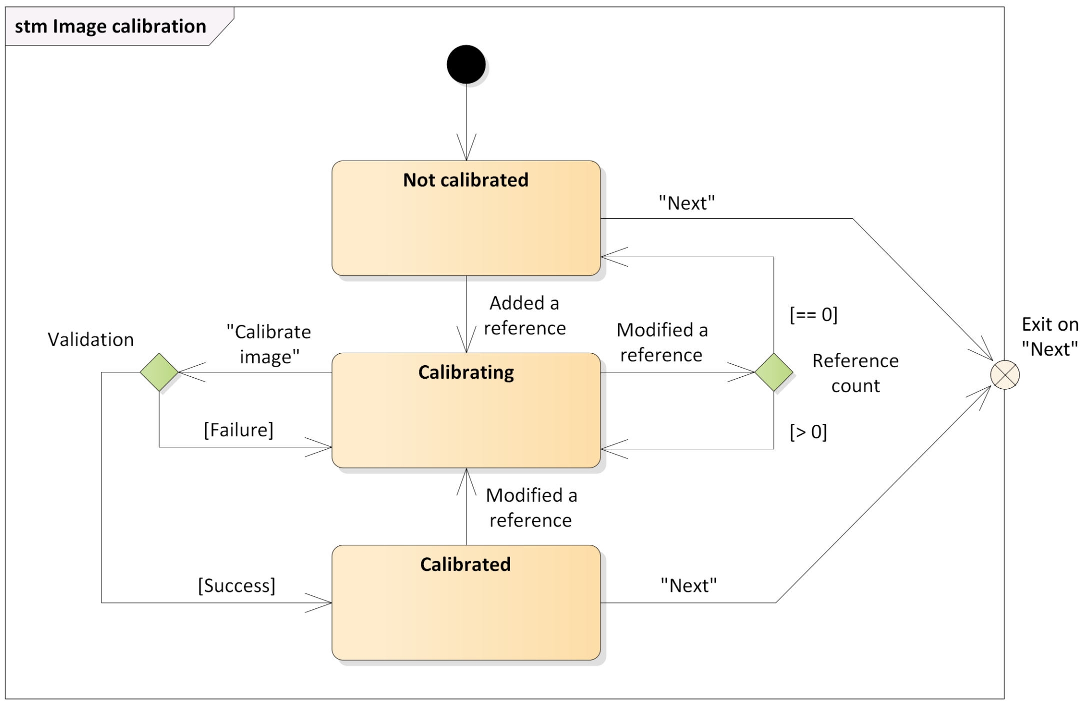


Figure 3.12: Level-3 UML state machine diagram of image calibration

Figure 3.12 explains the slightly more complex image calibration process. The user can proceed either if the image is "Not calibrated", or if it is "Calibrated", the latter

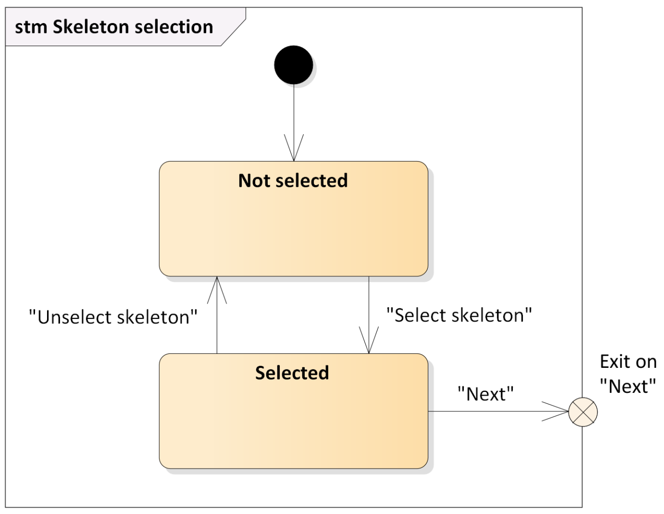


Figure 3.13: Level-3 UML state machine diagram of skeleton selection

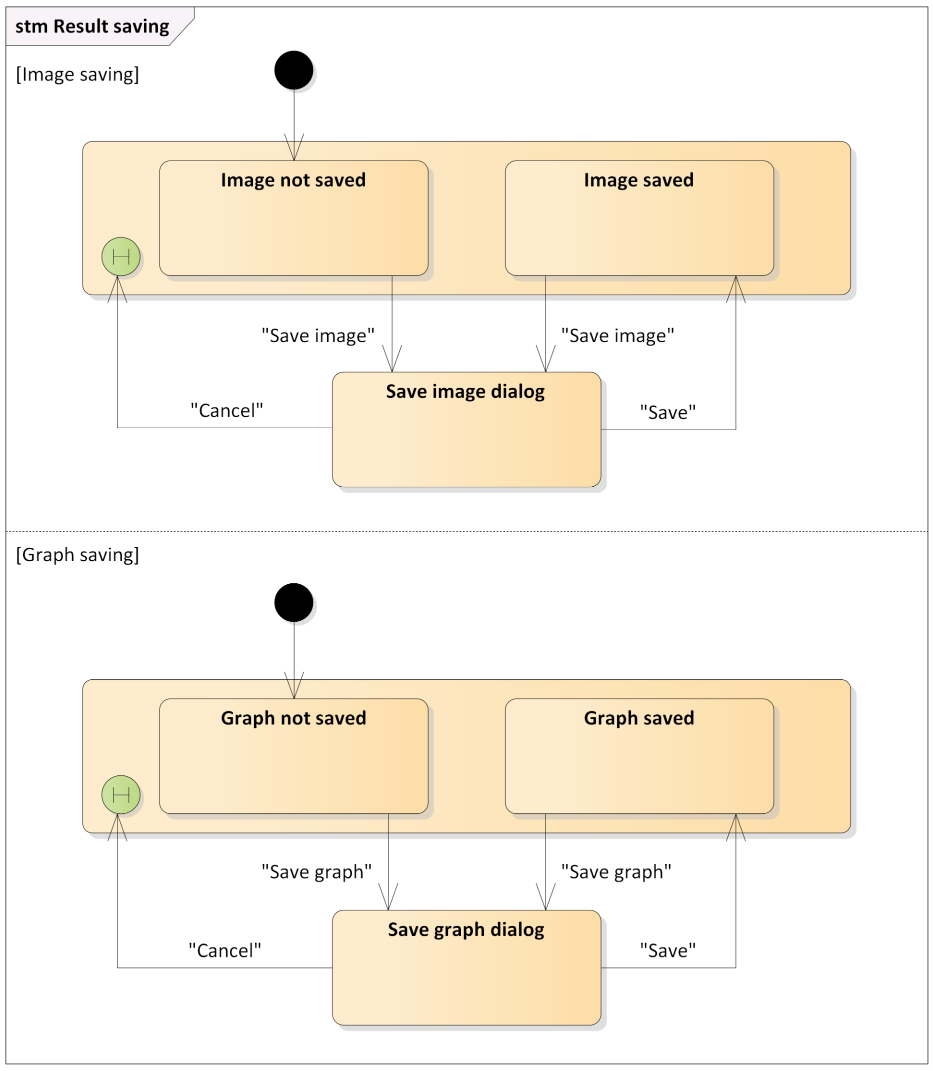


Figure 3.14: Level-3 UML state machine diagram of result saving

meaning that the user has entered a valid set of calibration parameters. When a new reference is added, or an existing reference is modified (its parameters are changed, or it is deleted), the software enters the "Calibrating" state, which can be left either by deleting every parameter, or by clicking the "Calibrate image" button, but only if the the parameter set passes validation.

Figure 3.13 shows the skeleton selection process. The user can only proceed if the skeleton of the graph is selected. But the selection can be modified, which is very useful if there are multiple curves to be extracted from the same image. The selection algorithm sometimes requires the manual addition or deletion of pixels (not shown).

Finally, Figure 3.14 shows the result saving step. It is composed of two concurrent composite states: one for saving the processed image, and another for saving the extracted graph. These are very similar: clicking one of the "Save" buttons displays a "Save" dialog box, which gives the user the opportunity to select an output folder and a filename. The saving process can be repeated, if saving to another location and/or with a different filename is needed.

3.5 Application structure

3.5.1 Model, view, controller

The application's architecture follows the Model-View-Controller (MVC) pattern:

Model (M)
The model is composed of the application's core data structures and algorithms. In Graph Digitizer, the data structures are related to the processed image and the extracted skeleton, and most algorithms perform image or skeleton processing.

View (V)
The view is the application's user interface. In Graph Digitizer, the main form, its panel hierarchy, and the controls on the panels make up the view.

Controller (C)
The controller orchestrates the cooperation of the model and the view. Graph Digitizer has a high-level controller for the whole digitization process, and low-level controllers for each digitization step.

Traditional MVC applies the Observer design pattern: changes of the model notify the view directly. In Graph Digitizer, an MVC variant is used, called Passive View

[5]. In the Passive View paradigm, there is no direct connection between model and view: every piece of information flows through the controller. The view transmits incoming user events to the controller, which interprets the events, and modifies the model and the view accordingly. This approach follows the Mediator design pattern: the controller coordinates the complex interaction of independent view and model objects.

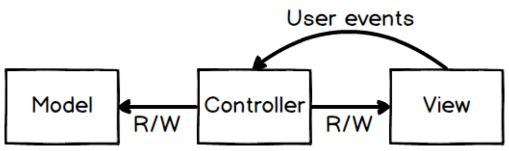
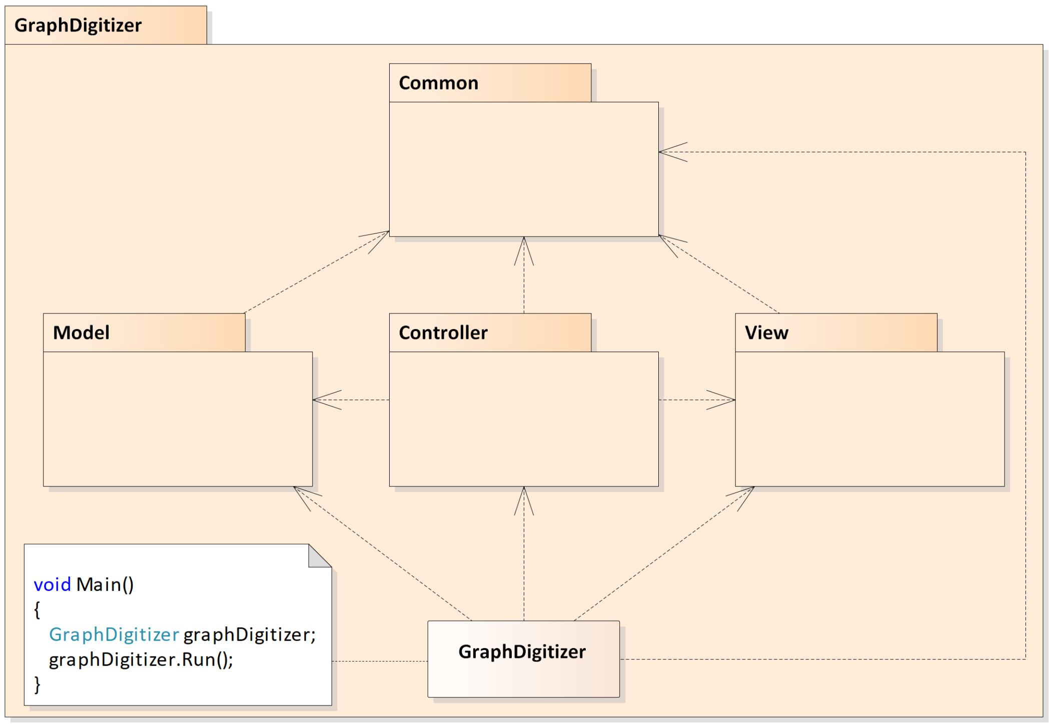


Figure 3.15: Passive View (R/W = read-write operations)

3.5.2 The GraphDigitizer package

The top-level package (and the corresponding C++ namespace) of the application is called `GraphDigitizer`. It contains the application's entry point, the `Main` function; the application's main class, `GraphDigitizer`; and four embedded packages (C++ namespaces): `Common`, `Model`, `View`, and `Controller` (Figure 3.16).



```
void Main()
{
  GraphDigitizer graphDigitizer;
  graphDigitizer.Run();
}
```

Figure 3.16: UML package diagram of the application

The `Main` function creates a `GraphDigitizer` object, and calls its `Run` method. The application is built using Dependency Injection: the constructor of `GraphDigitizer` is the Composition Root [6], where every persistent object is created, and receives its dependencies via constructor injection. The `Run` method does two things: starts the main controller (see below), and starts the application's message loop.

3.5.3 The Common package

The `Common` package contains simple types, which are used by types of more than one packages (Figure 3.17).

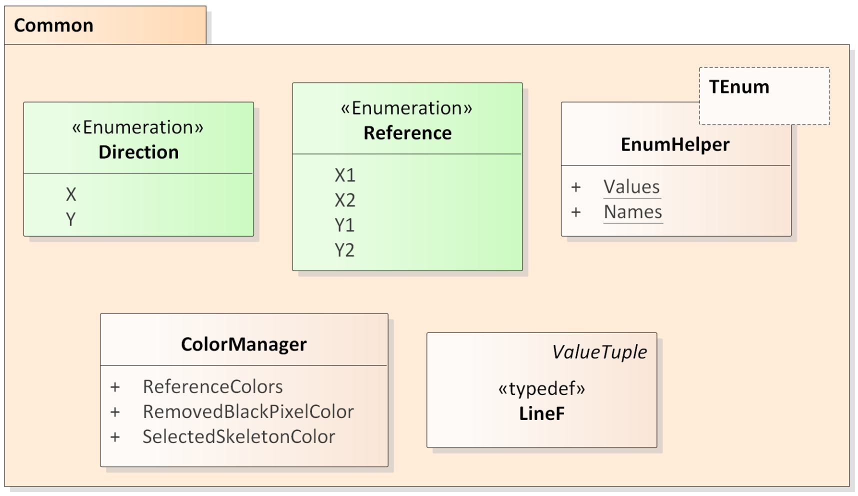


Figure 3.17: The Common package

The `Direction` enumeration represents the directions used in images: the X direction runs from left to right, while the Y direction from top to bottom. The origin is at the upper-left corner.

The `Reference` enumeration defines one pair of references per direction: (X_1, X_2) and (Y_1, Y_2), respectively. References are used for calibration purposes.

The `EnumHelper` class offers handy static properties for any enumeration given as its type parameter. `Values` is a collection of the values of the enumeration, which is useful in loops. `Names` is an associative array, which maps enumeration values to their names.

The `ColorManager` class collects a handful of colors, used at multiple places: the color code of the four references, the color of pixels removed by skeletonization (a shade of gray), and the color of the selected skeleton (red).

The `LineF` type definition exists because handling the crosshairs needs a line type, and the `System::Drawing` namespace does not have one.

3.5.4 The Model package

The `Model` package collects mainly data structures and algorithms (Figure 3.18).

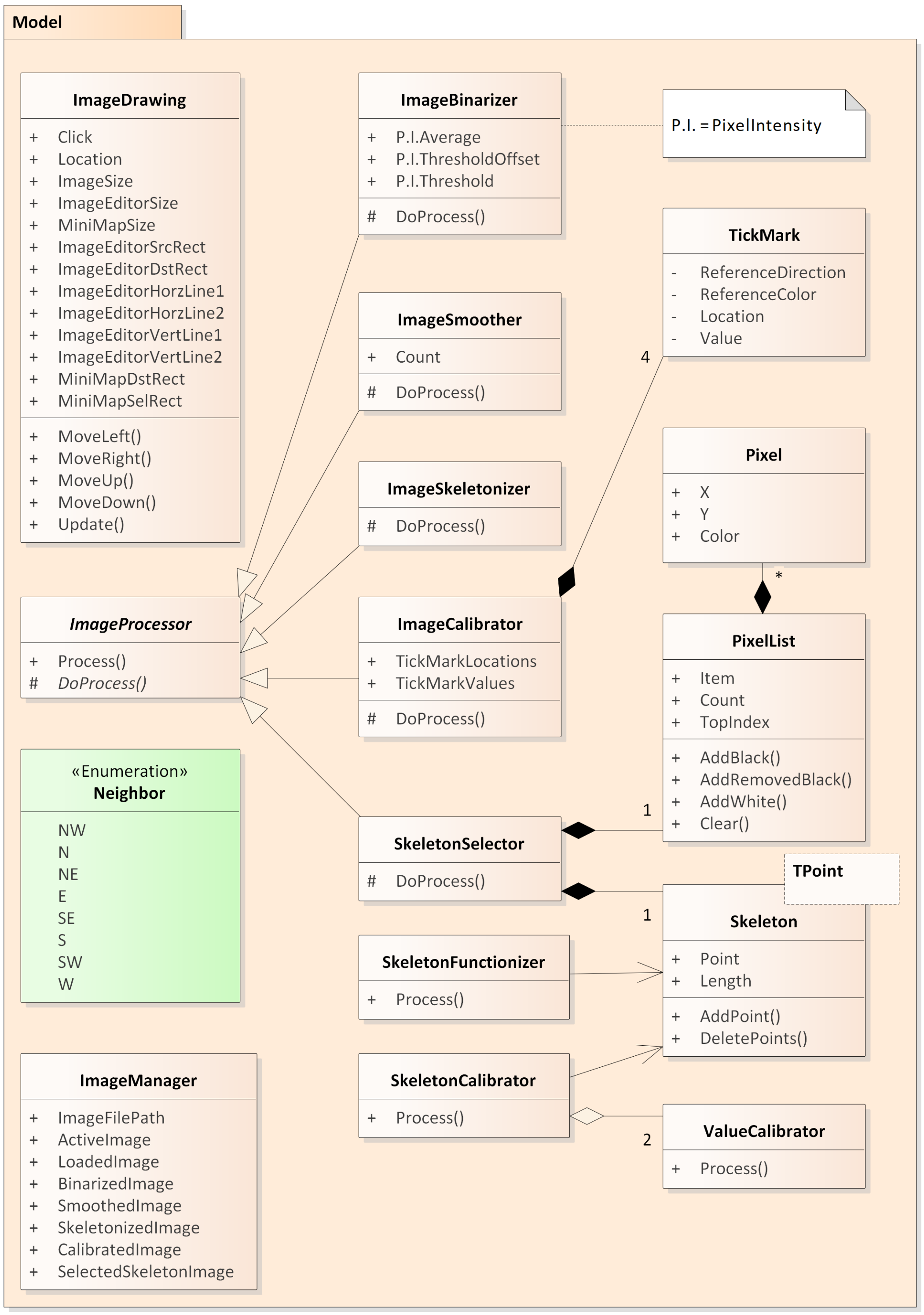


Figure 3.18: The Model package

The `ImageDrawing` class contains the algorithms required to calculate image drawing data. Its inputs are the location of the crosshairs; the size of the image editor panel, the minimap panel, and the image; and occasionally, when the user clicks on the minimap, the coordinates of the mouse click event. Its outputs are the rectangle of the image to be displayed in the image editor panel, and the rectangle of the latter to draw the image into; the coordinates of the four lines of the crosshairs; the rectangle of the minimap into which the whole image is drawn; and the selection rectangle, which shows in the minimap which part of the image is magnified into the image editor.

The `ImageManager` class stores the path of the loaded image file, which is used to generate the default path of the result files; every version of the image, created at different steps of the digitization process; and a reference to the image version displayed on the GUI at any given moment. Storing every image version is required by the "Back" button functionality.

The `Neighbor` enumeration is used by the image processing algorithms. Given a pixel, it lists the directions of its neighbors (where N = North, E = East, S = South, W = West), in clockwise direction, starting from the upper-left (North-West, NW) neighbor. The numeric value of enumeration members runs from 0 to 7, which can be regarded as indexes (Figure 3.19). In a binary image, pixels can be white (0) or black (1), so there are $2^8 = 256$ possible neighbor configurations.

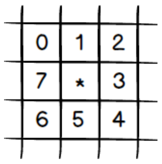


Figure 3.19: The indexing of the neighbors of a pixel

The `ImageProcessor` abstract class provides a `Process` method, which converts an image into another. The Template Method design pattern is applied: The `Process` method clones the image, stores its properties in protected fields, copies its pixel bytes into a byte array, and calls the abstract (pure virtual) `DoProcess` method, to do the pixel processing. After `DoProcess` returns, it copies back the pixel bytes into the clone, and returns it. The `DoProcess` method is implemented in the five subclasses of `ImageProcessor`:

1. The `ImageBinarizer` subclass first calculates the pixel intensities (I) of the image, effectively converting it to grayscale, using the following formula: $I = 0.3R + 0.6G + 0.1B$ [7], where R, G and B are the red, green and blue components of the pixel color, respectively. Then it calculates the average intensity, and a binarization threshold, which is the sum of the average intensity and an offset value. The default threshold offset is -25, which proved good enough in practice. The user can change the offset from the default, should the need arise. Every pixel with an intensity below the threshold are converted to black; other pixels are converted to white. The image was kept in 24 bpp format, because colors other than black and white are also used in later steps.

2. The `ImageSmoother` subclass has a `Count` parameter: the number of smoothing rounds to be performed, given by the user. In each round, the smoothing algorithm counts the number of black neighbors for every pixel, then flips the color of every pixel, which has three or less neighbors with the same color as its own.

3. The `ImageSkeletonizer` subclass implements a modified version of the skeletonization algorithm described in [1]. It utilizes a classification of pixels, based on their neighbor configuration (see above). The 256 possible configurations are grouped into six types, numbered from 0 to 5 (Figure 3.20).

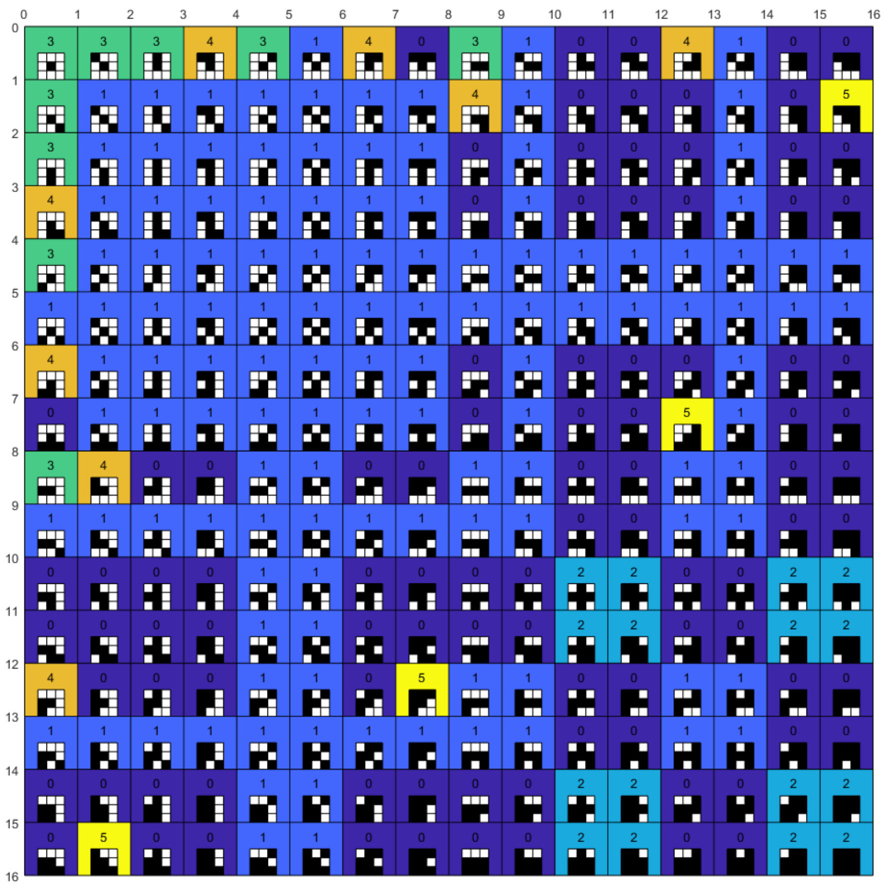


Figure 3.20: The pixel type lookup-table used by the skeletonization algorithm

The algorithm has two phases. The first phase is called **large-scale thinning**. In this phase, all pixels are removed, which are not internal (not type-2) but have at least one internal (type-2) neighbor, and not local articulation points (not type-1). Local articulation points are those pixels the removal of which would disconnect their 3x3 neighborhood. That is, this phase peels off boundary pixels, without severing 1-pixel-wide connections. This is repeated until no more boundary pixels can be removed. The second phase, the **small-scale thinning** refines the result by gradually removing type-5, type-0 and type-4 pixels, in this order. For the user to be able to assess the result of skeletonization, removed black pixels are not replaced by white pixels, but colored to gray instead. So to be precise, the algorithm works with black and non-black pixels, not with black and white pixels.

4. The `ImageCalibrator` subclass manages the tick marks of references, modeled by the `TickMark` class. There may be two tick marks in the X, and two in the Y direction. The tick marks are short line segments of bright color (defined in `ColorManager`). The central point of the tick defines its `Location`, and the coordinate of the central point in the direction of the reference gives its `Value`. This value can be called "image value", because it is given in pixel coordinates, compared to the "graph value" of the reference, which is given in real, physical units (see below).

5. The `SkeletonSelector` subclass has a `Skeleton` and a `PixelList` object, which are empty initially. The selection process will run only if at least one skeleton point is defined by the user. This must be a real skeleton point (that is, black), and it will be regarded as the leftmost point of the skeleton. The selection process is very simple. Assuming that the skeleton follows the graph of a function, which goes up and down, running from left to right, the algorithm checks the N, NE, E, SE and S neighbors of the current pixel, and if it finds a black neighbor, adds it to the `Skeleton` object. Unfortunately sometimes the skeleton is not ideal, and the user has to manually edit it by adding black, gray ("removed black") or white pixels to the image. These extra pixels are stored in the `PixelList`, and are drawn on the image before the selection algorithm is run. The "Undo" and "Redo" operations for manual pixel changes are implemented via the `TopIndex` property of the `PixelList`: the "Undo" operation moves this index toward `-1`, while the "Redo" operation moves it towards `Count - 1`. Only pixels at or below `TopIndex` are drawn.

Note that out of the seven stages of the graph digitization process, neither the loading, nor the saving step uses an `ImageProcessor`. The loading stage does not have one, because it simply loads the image, and does not process it in any way. And the saving stage does not have one, because it does not work on the image, it works with the selected skeleton instead. There are two things to do with the selected skeleton before it can be saved: "functionization" and calibration.

The `SkeletonFunctionizer` class iterates over the points of the skeleton, and every time it finds a vertical column of points, it replaces this column with its middle point. This is needed because functions map each x value where they are defined to a single y value.

The SkeletonCalibrator class is used to calibrate the coordinates of skeleton points, by converting their pixel coordinates to real units. This is done separately in the X and Y directions, but only if a ValueCalibrator object exists for the given direction. The latter is able to convert a single coordinate value to real units, so SkeletonCalibrator calls its Process method in a loop, which iterates over every skeleton point.

Note that Skeleton has a type argument. It can be either the point type which has integer coordinates (Point) or the one having floating-point coordinates (PointF). SkeletonSelector uses the former, because it extracts skeleton points from the image, and pixel coordinates are integers. SkeletonFunctionizer replaces columns with their middle point, so its output uses the latter, and SkeletonCalibrator obviously works with only the latter.

3.5.5 The View package

The View package collects the types which are used to build the Graph Digitizer GUI (Figure 3.21).

The DialogManager class handles all dialogs used by the application. This helps keeping the code clean.

The MainForm class is a subclass of Form, and represents the main window of the application.

To be able to move the crosshairs around by using the arrow keys, the ProcessCmdKey method of the Form class had to be overridden in MainForm. Since the Passive View paradigm requires every event to be handled by a controller object, the overriden method simply fires a newly defined event, CmdKeyPressed. To follow the standard .NET event pattern, a new event argument type had to be defined for this event, called CmdKeyEventArgs.

The MainPanel class is a subclass of TableLayoutPanel. It fills the client area of the MainForm, and provides the layout for the five main panels of the application:

1-2. The ImageEditorPanel and MiniMap classes are double-buffered subclasses of Panel. They are image display panels, so do not have any controls on them.

3. The StatusPanel class is a subclass of TableLayoutPanel. It holds three labels in a single row: the status label, and the X and Y crosshair location labels (Figure 3.4). It has three convenience methods to set the content of these labels

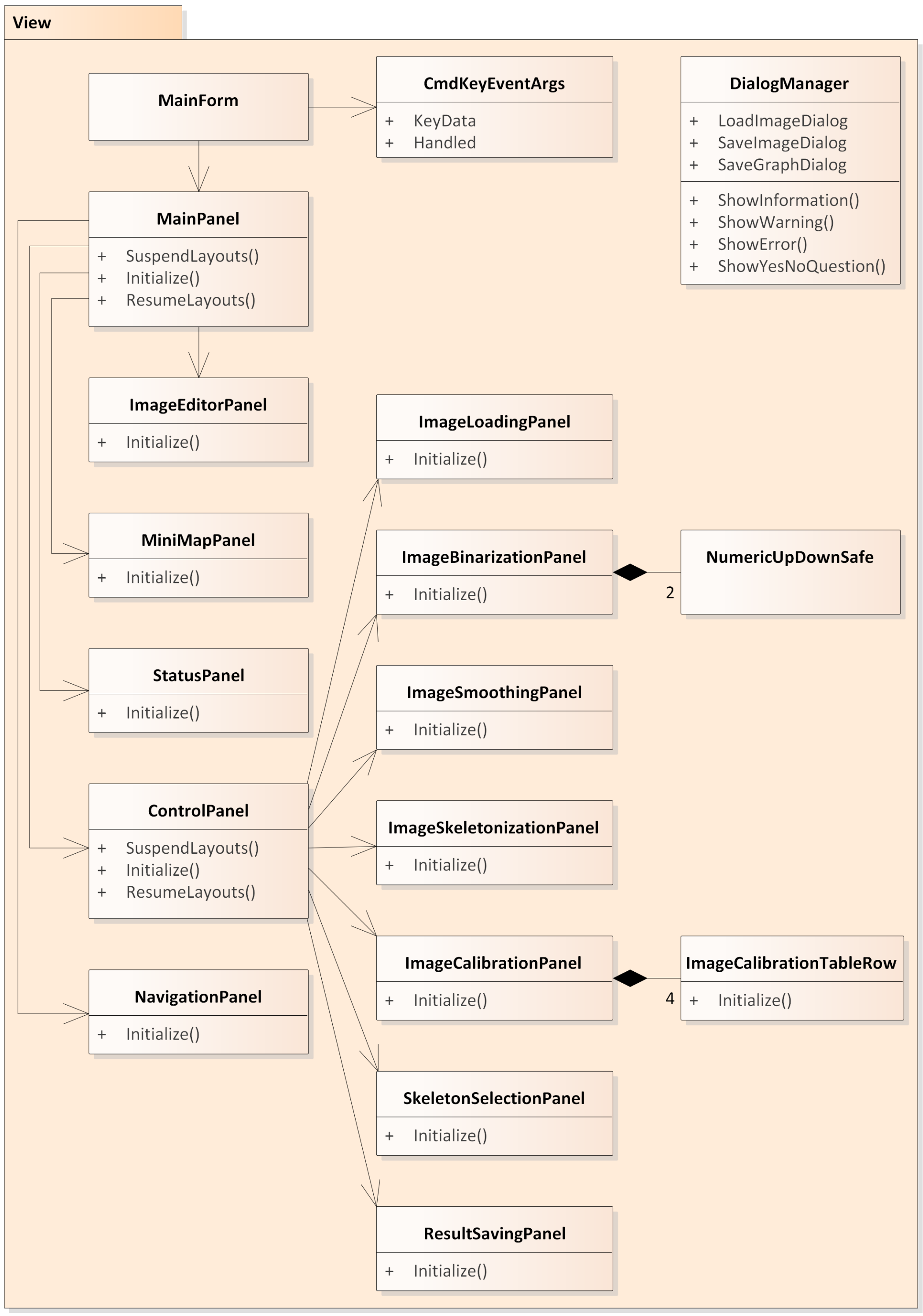


Figure 3.21: The View package

(not shown). These methods are responsible to format the content passed to them, before displaying it on the label.

4. The `ControlPanel` class is a `Panel` subclass, which holds seven sub-panels: `ImageLoading-`, `ImageBinarization-`, `ImageSmoothing-`, `ImageSkeletonization-`, ImageCalibration-, SkeletonSelection- and ResultSavingPanel. All of these sub-panels are subclasses of TableLayoutPanel, and they contain the controls shown in Figure 3.5. All controls are accessible from the outside, so that the corresponding controller could access their properties, and bind event handlers to them. The panels with a label provide convenience methods to set the label's content in a properly formatted way. There are two special controls created specifically for the sub-panels. One is NumericUpDownSafe [8], a subclass of NumericUpDown, which does not fire the ValueChanged event, if its Value property is changed from code. This control is used by the ImageBinarizationPanel. The other is ImageCalibrationTableRow, which contains all controls required to manage one reference, and has the color defined for this reference by ColorManager. The ImageCalibrationPanel contains four instances of the table row control, one for every reference.

5. The NavigationPanel class is a subclass of TableLayoutPanel. It only contains the "Back" and "Next" buttons (Figure 3.4).

Note that every panel has an Initialize method. This method is responsible to set the panel's own properties, and to initialize its child panels and/or controls. In case of panels, the SuspendLayout methods are called first, and after the initialization is finished, ResumeLayout calls follow.

3.5.6 The Controller package

The Controller package contains the classes which implement the application's behavior (Figure 3.21), as defined in section 3.4.

The structure and function of controller classes follow the technique described in [9]:

a) Controllers basically implement the UML state machine diagrams shown in section 3.4. The level-1 and level-2 state machines are implemented by the main controller class, GraphDigitizationController. The seven level-3 state machines are implemented by seven classes named after them.

b) Every controller class has a private nested State enumeration, and a state field of the same type ("state variable"). [9] identifies states with integers, but applying enumerations renders the code more readable.

c) For every state X there is a GoToStateX private method ("state procedure"), which sets the state variable accordingly, and sets the state of the controlled View objects. Specifically, it is typical that state procedures modify the Visible and/or

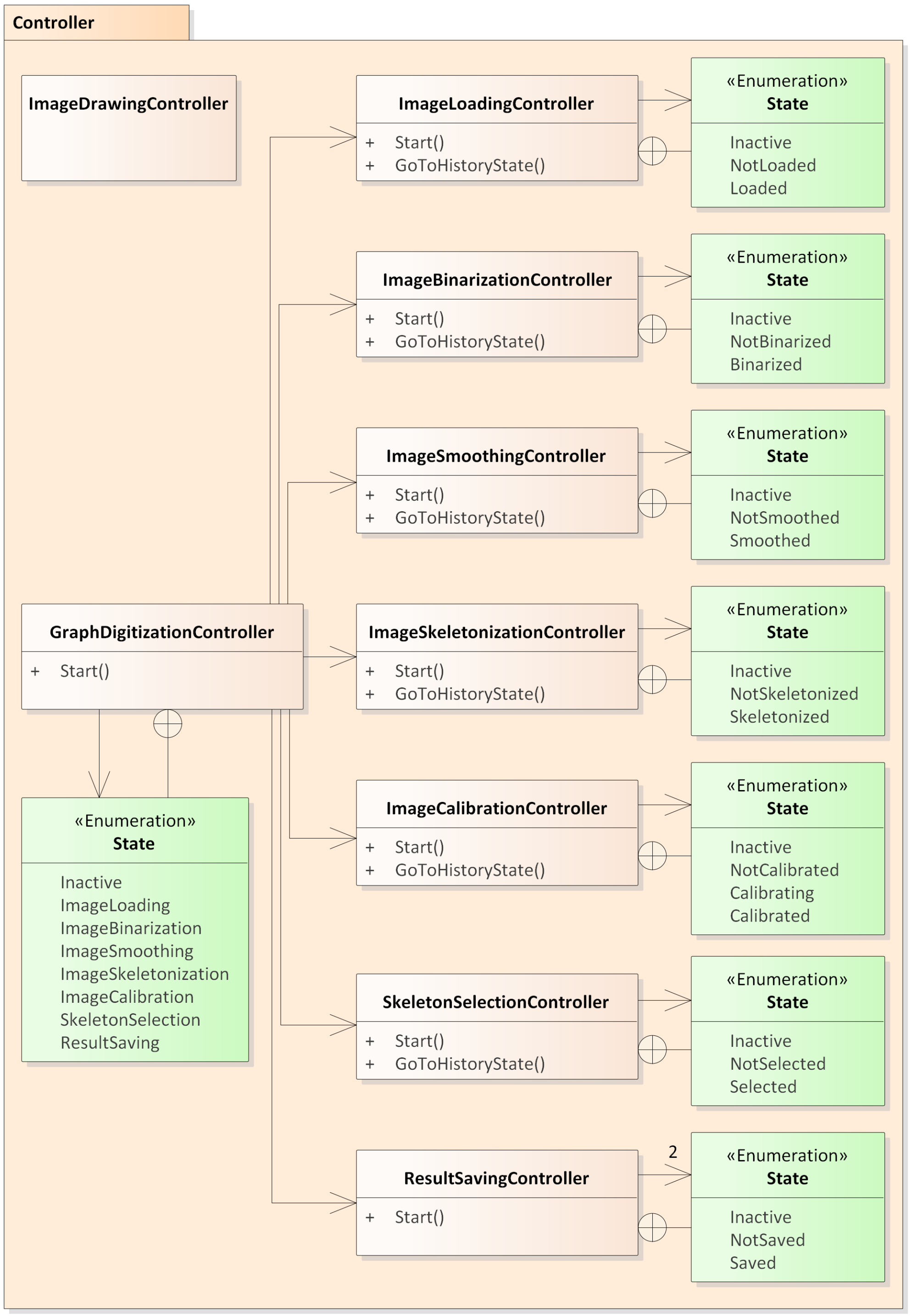


Figure 3.22: The Controller package

`Enabled` properties of controls, thereby defining the set of user events the application can receive in a given state. (State procedures could also contain the code of entry, do, and exit actions, but [9] advises against them, so they were not used.)

d) For every user event the controller is interested in, there is a private event handler method. These are wired to the corresponding view object in the constructor (controllers get their view dependencies via constructor injection, see above). Actions triggered by events are implemented in the corresponding event handlers. When the action is done, the handler calls the state procedure required to transfer the controller to the target state. If the event can be fired in more than one states, the handler has to choose the action and the target state based on the current state.

e) The constructor sets the state to `State::Inactive`. Every controller has a public `Start` method, which calls that state procedure which transfers it to its initial state. Generally, some state procedures can also be public, if they need to be called by higher-level controllers. In case of Graph Digitizer, only the `GoToHistoryState` method is public (for controllers which have one).

f) As mentioned above, the `Run` method of the `GraphDigitizer` object calls the `Start` method of the main controller, `GraphDigitizationController`, which in turn calls the `Start` method of `ImageLoadingController`. This is the reason why the application starts with waiting for the user to load an image.

g) The main controller's `OnNextButtonClick` event handler calls the `Start` method of the next step's controller object, but if there are any model objects used by this controller, it resets them first. On the other hand, the `OnBackButtonClick` event handler calls the `GoToHistoryState` method of the previous step's controller, but before doing that, it deletes the image version created by the current step, and sets the active image to the one created in the previous step.

The `ImageDrawingController` is simpler than the other controllers, because it does not implement a state machine. Its task is to ensure that the proper portion of the image is visible in the `ImageEditorPanel` after the user loaded a new image, pressed an arrow key, clicked on the `MiniMapPanel`, or resized the `MainForm`. Its work is based on the calculations of the `ImageDrawing` object.

3.6 Testing

3.6.1 Image loading tests

Test case #1: Load 24 bits-per-pixel RGB images: BMP, JPG, PNG and TIF.

Expected result: The images are loaded, and displayed correctly.

Result: Same as expected.

Test case #2: Try to load a 32 bits-per-pixel RGB image.

Expected result: The image is not loaded, a warning dialog box is displayed.

Result: Same as expected.

3.6.2 Image drawing tests

Test case #1: Open a 1x1 image. Press the arrow keys. Click on different points of the minimap (in the middle, next to edges and corners). Resize the main form (make it wide, make it tall).

Expected result: In the image editor panel, the single pixel of the image fills the target area of the crosshairs. In the minimap, the single pixel of the image is a magnified square, with a size equal to the smaller dimension of the minimap. Pressing the arrows and clicking on the minimap has no effect. Resizing the main form has no effect on the image editor panel; in the minimap, the single pixel of the image is always as large as the minimap's smaller dimension.

Result: Same as expected.

Test case #2: Open an image small enough to fit into the image editor panel: 5x3. Try the same as above.

Expected result: In the image editor panel, the whole image is visible, and the crosshairs target its upper-left corner pixel. In the minimap, the image is as large as can be fit without changing the aspect ratio. Pressing the arrow keys move the crosshairs around the image; every pixel is reachable; the crosshairs cannot leave the image. Pressing any arrow key once, while Shift is held down, moves the crosshairs to the opposite edge of the image. Resizing the main form has no effect on the image editor panel; in the minimap, the image is always as large as can be fit with keeping the aspect ratio.

Result: Same as expected.

Test case #3: Open images large enough to not fit into the image editor panel: 35x25, 25x35, 35x35. Try the same as above.

Expected result: In the image editor panel, only part of the image is visible, and the crosshairs target its upper-left corner pixel. In the minimap, the image is as large as can be fit without changing the aspect ratio. The semi-transparent rectangle, which shows the magnified part's position and size, is visible, and its position and size is correct. Pressing the arrow keys move the crosshairs around the image; every pixel is reachable; the crosshairs cannot leave the image. The semi-transparent rectangle also moves correctly. Pressing any arrow key once, while Shift is held down, moves the crosshairs 10 pixels at once. Resizing the main form has no effect on the image editor panel, even if the window is large enough to show the whole image; in the minimap, the image is always as large as can be fit with keeping the aspect ratio.

Result: Same as expected.

3.6.3 Image binarization tests

Test case #1: Load an image with pixels having the following (R, G, B) colors: (0, 0, 0), (64, 64, 64), (128, 128, 128), (192, 192, 192), (255, 255, 255). Binarize the image. Decrease the threshold to 0, then increase it to 255.

Expected result: Pixels become black exactly when the threshold reaches their intensity, that is, at 0, 64, 128, 192 and 255, respectively.

Result: Same as expected.

Test case #2: Load an image with pixels having the following (R, G, B) colors: (255, 0, 0), (0, 255, 0), (0, 0, 255). Binarize the image. Decrease the threshold to 0, then increase it to 255.

Expected result: Pixels become black exactly when the threshold reaches their intensity, that is, at $\lceil 0.3 * 255 \rceil = 77$, $\lceil 0.6 * 255 \rceil = 153$, and $\lceil 0.1 * 255 \rceil = 26$, respectively.

Result: Same as expected.

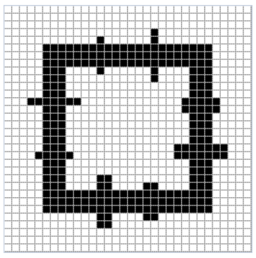


Figure 3.23: Smoothing test input

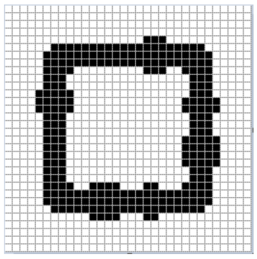


Figure 3.24: Smoothing test output

3.6.4 Image smoothing tests

Test case: Load the image shown in Figure 3.23. Binarize it (no effect, already binary), then smooth it.

Expected result: The result should look exactly like Figure 3.24.

Result: Same as expected.

3.6.5 Image skeletonization tests

Test case #1: Load the image of the uppercase MyFrida font [10]. Binarize it, smooth it, then skeletonize it.

Expected result: The skeleton of individual letters should follow the letter's shape, as in Figure 3.25.

Result: Same as expected.

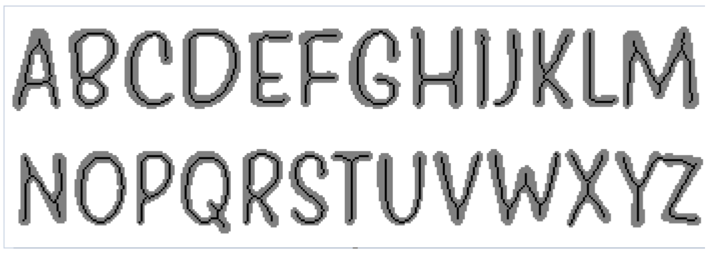


Figure 3.25: Font skeletonization test

Test case #2: Load the horse test image from the Scikit-Image documentation [11]. Binarize it, smooth it, then skeletonize it.

Expected result: The skeleton of the horse should follow the horse's shape (the tail, the legs, the body, the neck, the head, and the ears), as in Figure 3.26.

Result: Same as expected.

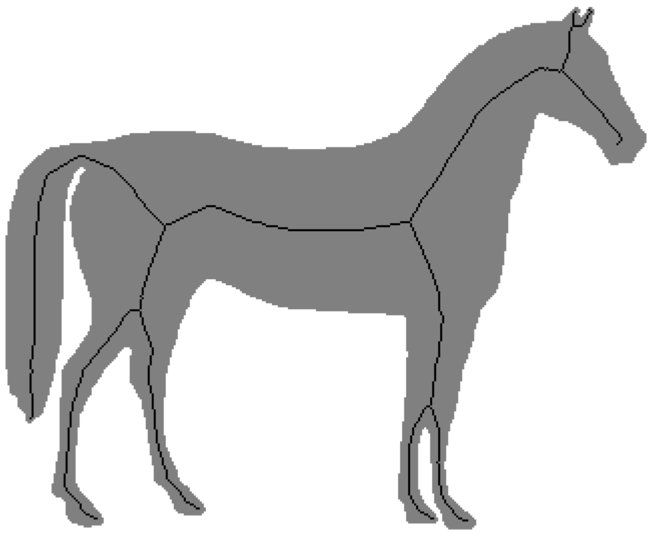


Figure 3.26: Horse skeletonization test

3.6.6 End-to-end graph digitization tests

In section 2.3, two end-to-end examples were described in detail. These examples demonstrate that the Graph Digitizer application fulfills the requirements listed in section 3.1.

The same tests have been repeated on every supported Windows version, to verify that apart from the visual differences, the application works the same everywhere. No problems have been found (Figure 3.27).

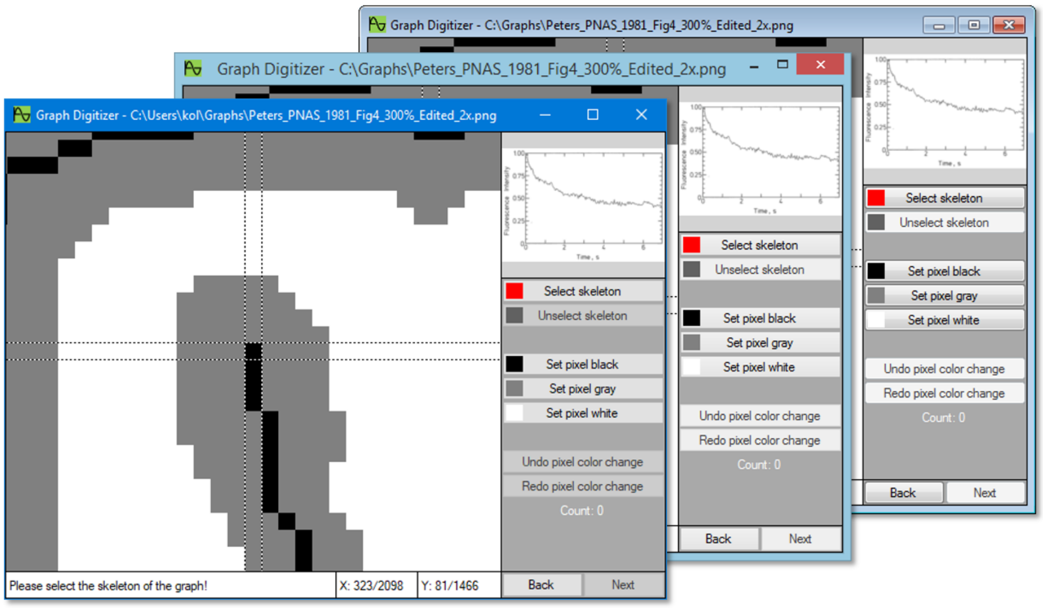

Graph Digitizer - C:\Graphs\Peters_PNAS_1981_Fig4_300%_Edited_2x.png
Graph Digitizer - C:\Graphs\Peters_PNAS_1981_Fig4_300%_Edited_2x.png
Graph Digitizer - C:\Users\kol\Graphs\Peters_PNAS_1981_Fig4_300%_Edited_2x.png
Select skeleton
Unselect skeleton
Set pixel black
Set pixel gray
Set pixel white
Undo pixel color change
Redo pixel color change
Count: 0
Back
Next
Please select the skeleton of the graph!
X: 323/2098
Y: 81/1466

Figure 3.27: End-to-end tests on every supported Windows version

Normally, the original data is unavailable, so the error of the graph digitization process cannot be determined. This is not surprising, because the whole purpose of the software is the recovery of the original data. In order to test the precision of the software, a test graph should be generated, based on known input data.

Test case: Load the image of the $sin(x)/x$ function. Extract the calibrated waveform, and compare it to the original data (this requires resampling).

Expected result: The difference between the two curves should be minimal, preferably less than 1% of the amplitude.

Result: Same as expected, see Figures 3.28 and 3.29.

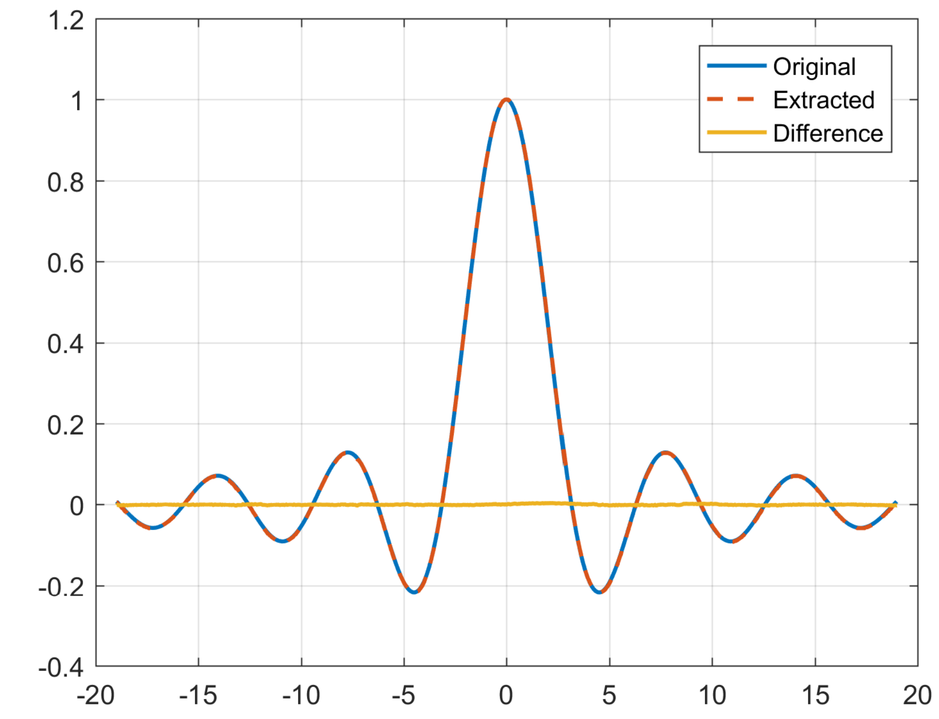


Figure 3.28: The original vs. the extracted $sin(x)/x$ function

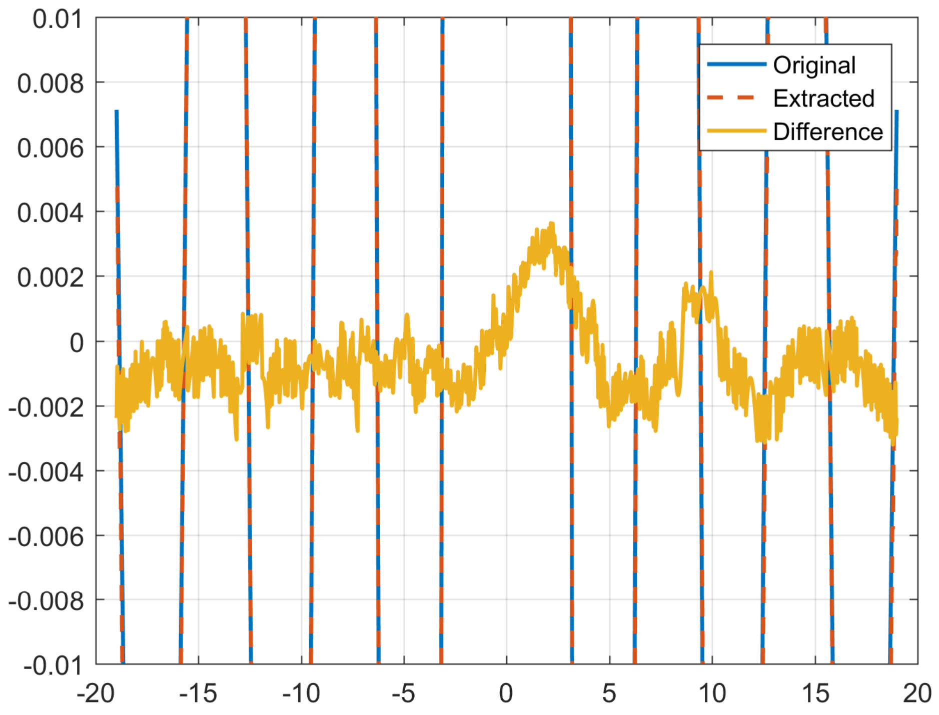


Figure 3.29: The digitization error, magnified

Bibliography

[1] D. Eberly. *Skeletonization of 2D Binary Images.* Geometric Tools, 2008. URL https://www.geometrictools.com/Documentation/Skeletons.pdf. Accessed: 2018-02-14.

[2] R. Peters, A. Brünger, and K. Schulten. Continuous fluorescence microphotolysis: A sensitive method for study of diffusion processes in single cells. *PNAS*, 78:962–6, 1981.

[3] R. Kelly, C. Hayward, A. Avolio, and M. O'Rourke. Noninvasive determination of age-related changes in the human arterial pulse. *Circulation*, 80:1652–9, 1989.

[4] IGN. *Command & Conquer: Red Alert 2.* Accessed: April 1, 2018. URL http://www.ign.com/articles/2000/10/24/command-conquer-red-alert-2.

[5] M. Fowler. *Passive View.* Accessed: April 1, 2018. URL https://martinfowler.com/eaaDev/PassiveScreen.html.

[6] M. Seemann. *Composition Root.* Accessed: April 1, 2018. URL http://blog.ploeh.dk/2011/07/28/CompositionRoot.

[7] Matlab. *The rgb2gray function.* Accessed: April 1, 2018. URL https://www.mathworks.com/help/matlab/ref/rgb2gray.html.

[8] StackOverflow. *NumericUpDownSafe class.* Accessed: April 1, 2018. URL https://stackoverflow.com/a/2230797/600135.

[9] I. Horrocks. *Constructing the User Interface with Statecharts.* Addison-Wesley, 1st edition, 1999. ISBN 0201342782.

[10] FFonts. *MyFrida Font (Uppercase).* Accessed: April 1, 2018. URL https://www.ffonts.net/Myfrida.font.

[11] Scikit-Image. *Skeletonization.* Accessed: April 1, 2018. URL http://scikit-image.org/docs/dev/auto_examples/edges/plot_skeleton.html.