Image processing in Perl graphic applications

Dmitry Karasik

Copenhagen University

2nd April, 2003

Abstract

The perl language is a powerful tool for creation of portable programs on a wide range of platforms. Support of the graphic user interface is not included in perl, and is provided by several stand-alone graphical toolkits. The most popular modules are Perl-Tk and Wx, based on multi-platform libraries, correspondingly Tcl-Tk and wxWindows, both written in C. The toolkit Prima was written to provide features non-existent in the other toolkits, and it was used as a base for development of research applications with needs of both graphic interface and image processing.

The target biological research comprises a study of the effects of applying various peptide-based preparations on neuron culture, with subsequent quantific ation of morphological changes in cells. The software provides both manual and automatic quantification, where the former depends on graphic interface functionality, and both on numerical processing.

The toolkits Prima, PDL, and IPA are used to demonstrate perl capabilities in the intersection area between image processing and graphic user interface. Prima is a platform-independent perl graphic toolkit with an object oriented interface. Its features include an extensive set of perl-coded widgets, a wide range of image types and conversion routines, and a visual builder. IPA is an image processing toolkit based on Prima functionality, and provides a set of common two-dimensional operators. PDL is a popular numerical toolkit featuring efficient storage of numerical arrays. It contains a wide spectrum of calculational functionalities, including a set of image processing operators designed to work with two and more dimensional data.

The work illustrates the usage of Prima, IPA, and PDL libraries for image conversion and display.

Introduction

In the last years signifi cant amount of efforts has been put into porting the perl language to a widening range of platforms. Mac OS, VMS, OS/2, Win32, just to name a few from the list of the supported non-unix systems are used to run scripts and modules, written for the other platforms [7]. Such is the effect of the perl API[1] being close to identical on different platforms, which facilitates the creation of portable perl programs. As perl evolves, it faces new areas of programming, and one of these is the graphic user interface.

Traditionally, GUIs[2] are written using toolkits, which range from simple, single-platform wrappers found in abundance, to vast suites with long-formed traditions and aged community. Most of them have in common that they are based on at least one platform-specifi c API, and defi ne their own API levels to cover the platform details and introduce new features.

In the perl area, Perl-Tk [8] is probably the most popular library for the GUI programming. It has a large advantage over its rivals that it is a platform-independent tool, and has a relatively long history. Its closest competitor, WxWindows [9], shares most of Perl-Tk functionality, but its distribution is somewhat lesser. These toolkits offer a more or less orthogonal set of functions that cover window management, input, 2D graphic output, standard dialogs, and the event loop.

[1] API - application program interface

[2] GUI - graphic user interface

As a base for small scripts and programs with infrequent graphic needs, these libraries can be seen as an ideal solution. When it comes to larger projects though, C toolkits are usually preferred. The key in understanding why does this happen lies in the nature of perl connection to the user C code, namely the XS interface. Although this area is very well documented, it often turns into an unexpected obstacle to one who needs accessing C code from perl, but have not been aware of existence of the XS layer.

On the other hand, it is not possible to abolish the need of using C code for any particular perl toolkit, nor this is desirable. Moreover, the balance between the functionality provided at perl level and the expected functionality is the decision factor between perl and C tools. Both Perl-Tk and Wx are based on C-written libraries, correspondingly Tcl-Tk and Wx-Windows. Their integration in the perl domain introduced the decision line, which delimits features accessible and not accessible from perl. The new toolkit Prima is designed so its functionality lies mostly in perl, and in particular requires no C code to extend its widget set, - which is not so for Tk and Wx.

As a demonstration of a successful combination of the both C and perl, the PDL toolkit can be used as an example. PDL stands for Perl Data Language and is devised for manipulation of multi-dimensional numeric tables. One of its modules, PDL::PP, provides an efficient translation of pseudo-code into XS code, which converted into C code. With the calculational speed of C, and coding convenience of perl, the toolkit presents an easily expandable framework.

The PDL toolkit [6] is a popular solution for perl-based implementation of numeric algorithms. It is also an option when the delivery of a program is troublesome due to the licensing or other restrictions of research tools.

As to the author's knowledge, there are no viable results of attempts to combine a mathematical and a GUI toolkit within the reach of the perl language. Here PDL is demonstrated to be successfully used together with Prima.

Prima, perl GUI toolkit

Prima (`http://www.prima.eu.org/`) is an platform-independent, perl-based GUI toolkit, working on Unix/X11, Win32, and OS/2 platforms. In addition to the functionality of the reviewed GUI libraries, it provides an extensible set of widgets, coded purely in perl. Having been designed originally for image processing needs, the toolkit developed into a library with several features never being implemented or well-developed in the perl GUI world before [1].

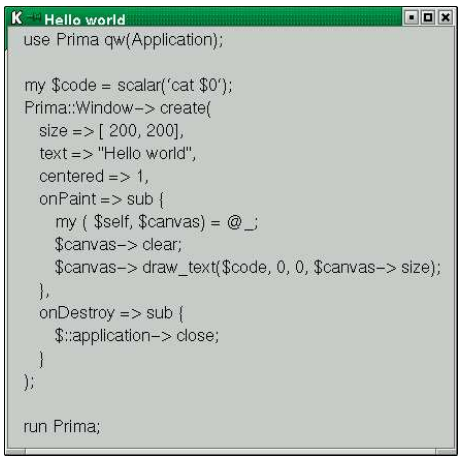
As a proof of the statement, below exemplified the toolkit concept of callback functions. The most widespread paradigm in object-oriented programming is to define a set of virtual subroutines, where overloading is performed via sub-classing. While this approach is also implemented in the toolkit, the other, more favorable method was used, when one or more anonymous subroutines is attached to the object instance, contrary to the object class.

This technique yields a convenient coding style when a widget or a window needs custom functionality. For example, *onPaint* callback type is called whenever the re-painting is needed, very much alike in many GUI designs, but the difference is that the function can be either anonymous or a classic overloaded subroutine. Moreover, there can be more than one anonymous callback function for each callback type. The concept aims the same goals as the subroutine overloading approach does, but achieves these with greater flexibility. In the example above, *onDestroy* callback function is not the only code executed when the *Destroy* notification is triggered; the class *Prima::Window* incapsulates intrinsic code for the notification, plus the object instance can contain other *onDestroy* callback functions.

The example in Fig.1 is a fully functional script, that shows a window with string "Hello world" and terminates when the window is closed by the user. The concepts of object options (*size*, *centered*, etc), and the event loop is similar to the ones in Tk, except the painting routine, which allows wider set of functions.

This aspect is important to the applications, dealing with image drawing and data plotting. While the Tk approach allows easier management of the on-screen objects, it is badly scalable, and requires C coding when the more complex drawing routines are required, for example, with custom clipping regions or raster logical operations.

Another feature of the Prima toolkit is a large set of supported pixel formats, and the means to convert between these. In addition to 1, 4, 8, and 24 bits per pixel formats, also byte, short, long, float, double, and




```
use Prima
qw(Application);

my $code = scalar('cat $0');
Prima::Window-> create(
   size => [ 200, 200],
   text => "Hello world",
   centered => 1,
   onPaint => sub {
      my ( $self, $canvas) = @_;
      $canvas-> clear;
      $canvas-> draw_text($code, 0,
0, $canvas-> size);
   },
   onDestroy => sub {
     $::application-> close;
   }
);

run Prima;
```

Figure 1: A "Hello world" program

complex pixel types are supported. If an image conversion involves data down-sampling, one of four error distribution algorithms can be selected.

List of the other features of the toolkit includes a RAD[3]-style visual builder, Pod[4] viewer classes, range of supported image fi le formats, unicode, PostScript(tm) printer interface.

IPA

The Prima toolkit was originally designed for image processing, but it contains little 2-D processing algorithms. This function was delegated to another toolkit, IPA, which stands for Image Processing Algorithms. Being derived from Prima, it employs its image storage and conversion capabilities, and is designed to be a portable and platform-independent library.

Whereas it is not attainable to cover all range of image processing diversity within such a library, nor the competition with the existing comprehensive software was the goal, still IPA features a set of basic processing functions grouped in modules by algorithms. Examples of the most signifi cant of these are:

- Point - brightness thresholding, addition, subtraction
- Local - median fi lter, Sobel operator, Deriche operator
- Global - area fi lter, Fourier transform
- Morphology - erosion, dilation, skeletonizing

The full list of functions is present in the library documentation.

As many numerical packets provide an interactive shell, also does IPA. Its minimalistic shell `iterm` is a combination of a command line with an image viewer window. Its usage primarily concerns interactive exploring of image processing operators.

Being covered by the same type of license as Prima, IPA presents a portable open-source solution for both perl and C programmers.

[3] RAD - rapid application development

[4] POD - plain old documentation, the standard perl document format

PDL

The Perl Data Language is a numerical framework, that in particular is suitable for implementing image processing algorithms. Some algorithms that apply to more than 2D image processing, for example, Fourier transform, are present in PDL already. But instead of re-implementing the routines to be accessible from under PDL, a connector package `PDL-PrimaImage` was written.

As the conversion between PDL scalars and Prima/IPA images is made easy, the platform for using the calculational powers of the both can be established with lesser efforts. Although PDL does provide very good visualization aids, and has intrinsic plotting functions, these are badly portable. In particular, `PDL::FAQ` complains about lack of graphics support on win32 platform, what is probably an extra reason to befriend PDL and Prima.

PDL itself contains a relatively large amount of numeric routines, and therefore its thorough description is not the goal of the paper; other sources [6] and books [4] cover the topic more comprehensively. PDL modules, useful for the image processing, are listed below:

- PDL::Graphics::LUT - look-up tables
- PDL::Image2D - set of 2-D image processing functions
- PDL::ImageND - set of N-D image processing functions
- PDL::ImageRGB - utility functions for RGB images

Applications

As a part of my research study, a set of programs has been implemented to facilitate the quantification of morphology of biological structures. The software development is performed primarily on FreeBSD, whereas the end users run the software on MS Windows.

The research project conducted in the Protein Laboratory, Copenhagen University, involves measuring the effect of application of extensive set of peptides onto neural cells. The treatment response is reflected in the morphology of the cell, in particular, in the development of neurites [5].

The quantification of changes in morphology is performed manually, and is both tedious and laborsome, since it consists primarily of the point-and-click registration routine. The application of set of algorithms on the images, recorded using different kinds of microscopy, produces reliable results that closely correlate to the data, acquired by human observers.

A program, used in the Protein Laboratory performes recognition of one hundred of 512x768 8-bit grayscale images in 294 seconds on a 2.4GHz machine under FreeBSD.

Discussion

The basic image processing operators have been re-implemented many times during the last 20 years. The most widely known software packages that include these, are commercial Intel's IPL (Image Processing Library), Microsoft's Visual SDK, Matlab, Imagtek's REX, and freeware ImageMagick, SciLab, ipl98, etc. Many software and hardware vendors come with their own code also. This, plus the availability of books on image processing algorithms implementation [2, 3] drives new libraries onto a highly competitive ground. Though none, except Matlab, of the listed libraries are not reported being connected to perl, it is not an unfeasible task to find a suitable library and make it available for perl. What makes IPA special in this regard, that it is not bound to a single platform, nor it is bound to any third-party (except perl itself) copyrights.

As an alternative to IPA, Intel's IPL is the most suitable library for multi-platform image processing. Although IPL is superior to IPA, it has no built-in support of perl. As IPL has its own memory allocation policies, it can be connected to perl via PDL, which has a specifically designed interface for such a linkage.

Matlab is another alternative, but its connection to perl is other way round - instead of providing access of its internal functions to perl, it integrates perl into itself. Although such a linkage does not leave any place

for numerical perl packages, still it realizes a way of interchanging data from perl and Matlab, especially given the power of the latter. Matlab also contains a data visualisation package, and in same regard it is superior to Prima, Perl-Tk, Wx, and is competitive to PDL's PGPLOT as a research tool. It is hardly possible though to use Matlab as a standalone library, and also to ship due to its licensing restrictions.

Among the freeware tools the most promising is Scilab `http://cyvision.if.sc.usp.br/~rfabbri/sip`, the scientifi c software suite, which adopts certain similarity to Matlab. The Scilab also contains an image processing toolboxSIP `http://cyvision.if.sc.usp.br/~rfabbri/sip/`. The suite also provides interface to Tcl/Tk and several contributed packages that use the Tcl scripting.

Prima's closest competitor, Perl-Tk, includes larger base of code, and unifi es both perl and tcl communities by preserving original Tcl-Tk API paradigm. This Perl-Tk feature is possible due to the fact that large amount of code is written in C, which also makes its porting easier for under the other languages; the same is valid for Wx also. From the other hand, porting Prima to another platform would require much less effort, because it does not include access to the platform widgets and standard dialogs. This is especially important for X11-based environments, with the most popular X11/Athena, Motif, GTK, and Qt toolkits. Wx supports several such front-ends, but it comes at the cost of supporting the common X11 code.

The functionality of Prima and Perl-Tk overlap, but single program cannot use features of the both, except of Prima image subsystem, which can be used separately. Perl-Tk unique features are widget `pack` and `grid` geometry managers; these are planned to be implemented in Prima also. List of Prima unique features include pure-perl implemented set of widgets, image conversion subsystem, and a visual builder. The perl implementation of sophisticated widgets, like a HTML browser, is not prohibitively expensive any longer, given the speed of the modern day computers. Such an implementation is (arguably) easier to develop and support than a C or C++ one, and in particular saves the expenses of revealing eventual memory corruptions and leaks, intrinsic to the low-level language implementations.

Conclusions

Development of all of the described toolkits is not fi nished. Being open source projects, they can be easily contributed. In particular, development of Prima and IPA toolkits is open, and is discussed on mailing list `prima@prima.eu.org`. To subscribe to the list, send mail to `<majordomo@prima.eu.org>` and include `subscribe prima <optional address>` in the body of the message.

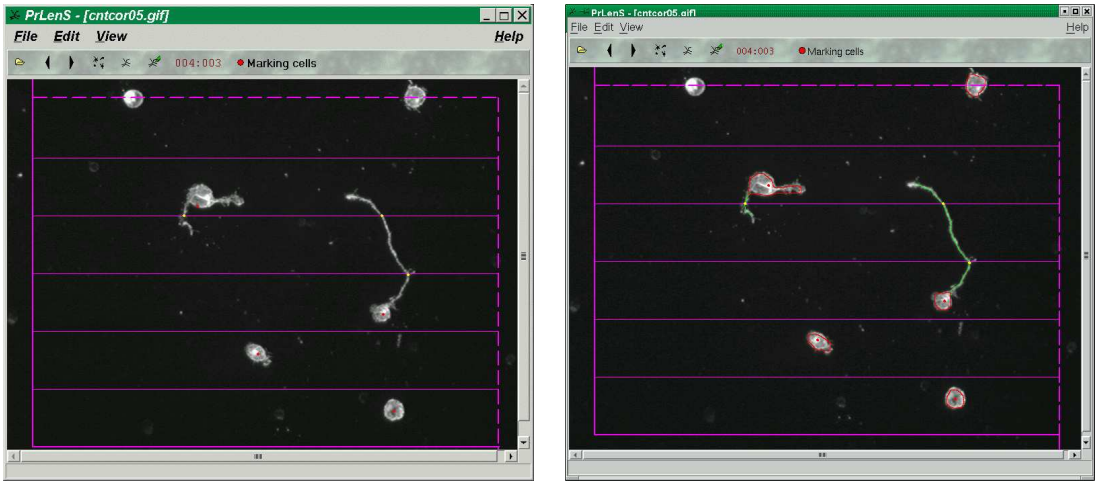


Figure 2: Manual and automated sampling of lengths of neurites. The picture on the left is a screenshot of the program, running under MS Windows in manual mode. Centers of cells and crossings of neurites are marked manually by the user. The picture on the right is a screenshot of the same program, running under X11 in automatic mode. Centers of the cells and neurites are extracted and outlined in different colors.

Examples

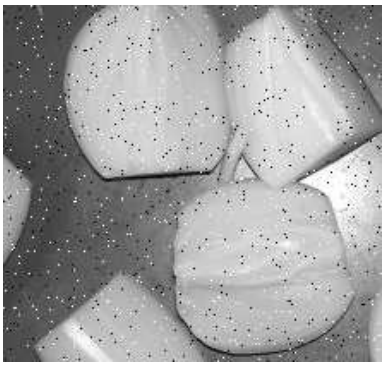
Figure 3: Demonstration of advanced plotting techniques using the Prima toolkit

```
# Displays window with text and background, filled with
# concentric circles of different colors.
use strict;
use Prima qw(Application);
# Construct image large enough to fit word "Gradient"
# written with a font of 100-point size
my $i = Prima::Image-> create(
   preserveType => 1,
   type => im::BW,
   font => { size => 100, style => fs::Bold|fs::Italic },
);
$i-> begin_paint_info;
my $textx = $i-> get_text_width( "Gradient");
my $texty = $i-> font-> height;
$i-> end_paint_info;
$i-> size( $textx + 20, $texty + 20);
# draw the word in white onto black background
my @is = $i-> size;
$i-> begin_paint;
$i-> color( cl::Black);
$i-> bar(0,0,@is);
$i-> color( cl::White);
$i-> text_out( "Gradient", 0,0);
$i-> end_paint;
# construct grayscale palette for better representation
# on paletted displays
my @xpal = ();
for ( 1..32) {
    my $x = (32-$_) * 8;
    push(@xpal, $x,$x,$x);
};
# construct window
my $w = Prima::Window-> create(
    onDestroy=> sub {$::application-> close;},
    size   => [ @is],
    centered => 1,
    buffered => 1,
    palette => [ @xpal],
    onPaint => sub {
      my ( $self, $canvas) = @_;
      $canvas-> clear;
      # paint the background circles
      my $xrad = $is[0] / 62;
      for ( 1..32) {
         my $x = (32-$_) * 8;
         $x = ($x<<16)|($x<<8)|$x;
         $canvas-> color($x);
         $canvas-> fill_ellipse($is[0]/2,$is[1]/2, $xrad*(32-$_)*2, $xrad*(32-$_)*2);
      };
      # apply clipping, so all drawing is confined to the bitmap region
      $canvas-> region( $i);
      # paint the circles with inverse gradation
      for ( 1..32) {
         my $x = ($_-1) * 8;
         $x = ($x<<16)|($x<<8)|$x;
         $canvas-> color($x);
         $canvas-> fill_ellipse($is[0]/2,$is[1]/2, $xrad*(32-$_)*2, $xrad*(32-$_)*2);
```

```
      };
  },
);
# event loop
run Prima;
```

Figure 4: Image conversion facilities of Prima toolkit. Down-sampling of a grayscale image: with constant threshold, with the threshold matrix, with error distribution.

```
use Prima;
my $i = Prima::Image-> load('lena.gif');
die "Cannot load:$@\n" unless $i;
my $result = Prima::Image-> create(
    width => $i-> width * 2,
    height => $i-> height * 2,
    type   => $i-> type,
);
my $k;
# copy normal, non-converted image into the upper left corner
$result-> put_image( 0, $i-> height, $i);
# copy result of simple thresholding into the upper right corner
$k = $i-> dup;
$k-> set( conversion => ict::None, type => im::bpp1);
$result-> put_image( $i-> width, $i-> height, $k);
# copy 8x8 ordered image into the lower left corner
$k = $i-> dup;
$k-> set( conversion => ict::Ordered, type => im::bpp1);
$result-> put_image( 0, 0, $k);
# finally convert the original image using error diffusion algorithm
$i-> set( conversion => ict::ErrorDiffusion, type => im::bpp1);
$result-> put_image( $i-> width, 0, $i);
$result-> save('lena4.gif');
```

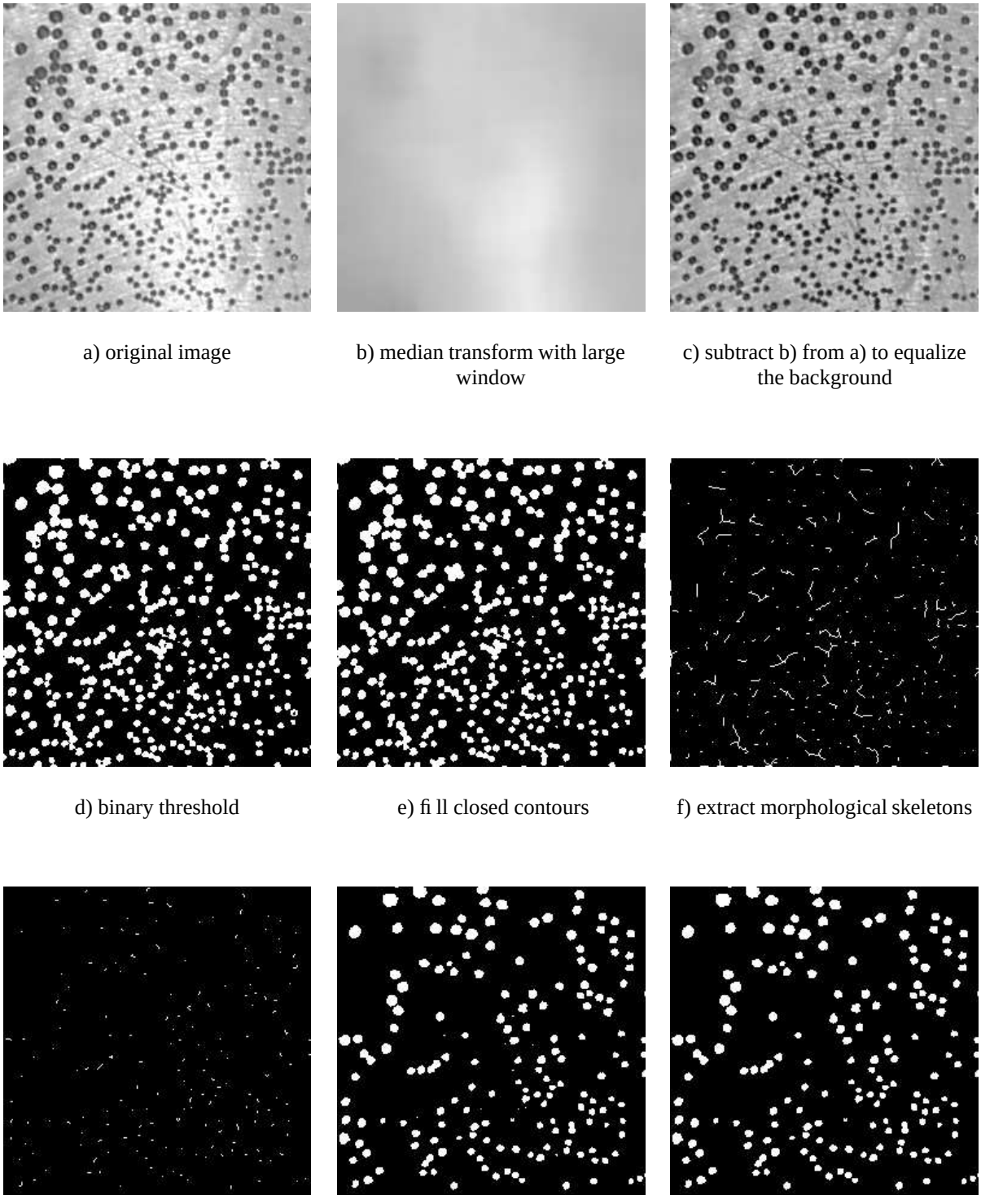


Original image with "salt and pepper" noise

Missing pixels averaged

Figure 5: PDL and IPA: De-noising a grayscale image by the averaging selected pixels

```
# demo of all toolkits
use PDL;
use PDL::Image2D;
use PDL::IO::Pic;
use Prima;
use IPA;
use IPA::Point;
use PDL::PrimaImage;
# read image as piddle scalar
my $img  = PDL-> rpic('onions.pgm');
# convert into Prima/IPA image
my $mask = PDL::PrimaImage::image( $img);
# map out all black and white pixels
$mask = IPA::Point::threshold( $mask, minvalue => 1,
      maxvalue => 254);
# convert back to piddle scalar
$mask = PDL::PrimaImage::piddle( $mask);
# normalize values to 0 and 1
$mask /= -255;
$mask++;
# average by PDL::Image2D::patch2d
my $result = patch2d $img, $mask;
```



a) original image

b) median transform with large window

c) subtract b) from a) to equalize the background

d) binary threshold

e) fi ll closed contours

f) extract morphological skeletons

g)remove large skeletons

h)combine e) and g) to fi lter out the clusters

i) remove very small spots

Figure 6: IPA toolkit demonstration: calculate mean vesicle area from the grayscale sample.

```
# Calculate mean vesicle area
  use Prima;
  use IPA;
  use IPA::Local;
  use IPA::Point;
  use IPA::Global;
  use IPA::Morphology;
  my $i = Prima::Image-> load('input.gif');
  die "Cannot load:$@\n" unless $i;
  # equalize image background
  my $median = IPA::Local::median( $i, w => 51, h => 51); # b)
  $i = IPA::Point::subtract( $i, $median);                # c)

  # binarize image
  $i = IPA::Point::threshold( $i, minvalue => 0,
        maxvalue => 128);                                 # d)
  $i = IPA::Global::fill_holes( $i);                      # e)
  # extract skeletons and remove the elongated ones, which
  # correspond to vesicle clusters
  my $skeletons = IPA::Morphology::thinning( $i);         # f)
  $skeletons = IPA::Global::area_filter( $skeletons,
      maxArea => 5);                                      # g)
  # filter the clusters and noise
  $i = IPA::Morphology::reconstruct( $i, $skeletons);     # h)
  $i = IPA::Global::area_filter( $i, minArea => 6);       # i)
  # get mean area
  my $n = scalar @{IPA::Global::identify_contours( $i)};
  my $area = $i-> sum / 255;
  printf "Mean area: %g pixels\n", $area / $n;
```

Links

- Prima toolkit `http://www.prima.eu.org/`
- IPA library `http://www.prima.eu.org/IPA`
- PDL toolkit `http://pdl.perl.org/`
- PDL-PrimaImage module `http://www.prima.eu.org/PDL-PrimaImage`

Acknowledgments

I would like to thank SourceForge for providing their Compile Farm facilities, and also Compaq for their Test Drive program. The both facilitated greatly the usability of Prima and IPA toolkit on PPC and Alpha processors, and the porting to SGI, SunOS, Solaris, OpenBSD, and NetBSD systems.

References

[1] Berezin A, Karasik D, Belman V, Berezin V, Bock E - PRIMA - Perl toolkit for X, win32 OS/2 PM. Proceedings on Third Perl Conference. O'Reilly 1999

[2] Gonzalez RC,Woods RE- Digital Image Processing. Addison-Wesley 1993

[3] Klette R,Zamperoni P - Handbook of Image Processing Operators. Wiley 1996

[4] Orwant J,Hietaniemi J,Macdonald J - Mastering algorithms with Perl. O'Reilly 1999

[5] Ronn LCB, Ralets I, Hartz BP, Bech M, Berezin A, Berezin V, Moller A, Bock E - A simple procedure for quantifi cation of neurite outgrowth based on stereological principles. J Neurosci. Methods, 100:25-32, 2000

[6] Soeller C, Lukka TJ, Glazebrook K, et al - The PDL manuals `http://pdl.sourceforge.net/PDLdocs/`

[7] Wall L,Christiansen T,Orwant J - Programming Perl, 3rd edition. O'Reilly 2000

[8] Walsh N - Learning Perl/Tk. O'Reilly 1999

[9] WxWindows documentation - http://www.wxwindows.org/