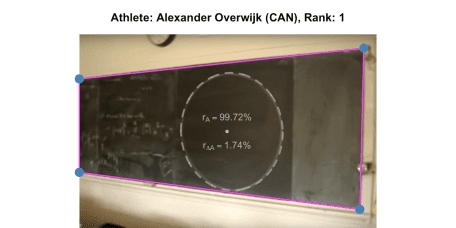
**Abstract:**

In 2007 Alexander Overwijk went viral with his ‘Perfect Circle’ video. The same year a World Freehand Circle Drawing Championship was organized, which he won. In this post we show how a mobile camera, R and the imager package can be used to develop an image analysis based method to judge future instances of the championship.



**Introduction**

*They have a laser machine called the* ***circleometer*** *that creates the perfect circle closest to the one you drew. The circleometer then calculates the difference in area between the laser circle and the circle that you drew. The machine then calibrates the area difference as if you had drawn a circle with radius one meter. The person with the smallest area difference is declared the world freehand circle drawing champion.*

Aha! Imaginary circleometers are expensive and my dean of study most likely isn’t open for investing in perfect circle measurement equipment… So here is a cheaper solution involving a mobile device camera, R and the imager package . Altogether a combination of modern **data science** tools, which my dean of study is most likely to approve! We’ll use a screenshot from the perfect circle video as motivating example to guide through the 3 phases of the method:

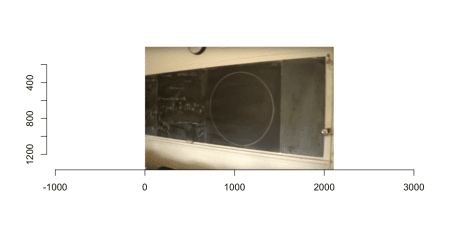
1. Image Rectification
2. Freehand circle identification and perfect circle estimation
3. Quantifying deviation from the perfect circle

We start by loading the screenshot into R using imager:

library("imager")

file <- "circle2.png"

img <- imager::load.image(file.path(fullFigPath, file))



**Image Rectification**

The image clearly suffers from perspective distortions caused by the camera being positioned to the right of the circle and, hence, not being orthogonal to the blackboard plane. Furthermore, small lense distortions are also visible – for example the right vertical line of the blackboard arcs slightly. Since the video contains no details about what sort of lense equipment was used, for the sake of simplicity, we will ignore lense distortions in this post. If such information is available one can use a program such as [RawTherapee](http://rawtherapee.com/) (available under a GNU GPL v3 license) to read the EXIF information in the meta data of the image and automatically correct for lens distortion.

To rectify the image we estimate the parameters of the 2D projection based on 4 ground control points (GPC). We use R’s locator function to determine the pixel location of the four corner points of the blackboard in the image, but could just as well use any image analysis program such as [Gimp](https://www.gimp.org/). Furthermore, we need the true object coordinates of these GPC. Unfortunately, these are only approximately available to due lack of knowledge of the size of the blackboard in the classroom. As a consequence a *guesstimate* of the horizontal length is used.

plot(img)

p <- locator(4)

p <- round(cbind(p$x, p$y))

dump(list=c("p"), "")

These points are now used to rectify the image by a Direct Linear Transformation (DLT) based on exactly 4 control points (Hartley and Zisserman 2004, Chapter 4). That is the parameters of the 3×3 transformation matrix \(H\) in homogeneous coordinates are estimated such that \(p' = H p\).

Complete Code – Judging Freehand Circle Drawing Competition

```{r,include=FALSE,echo=FALSE,message=FALSE}

##If default fig.path, then set it.

if (knitr::opts\_chunk$get("fig.path") == "figure/") {

knitr::opts\_knit$set( base.dir = '/Users/hoehle/Sandbox/Blog/')

knitr::opts\_chunk$set(fig.path="figure/source/2018-07-31-circle/")

}

fullFigPath <- paste0(knitr::opts\_knit$get("base.dir"),knitr::opts\_chunk$get("fig.path"))

filePath <- file.path("","Users","hoehle","Sandbox", "Blog", "figure", "source", "2018-07-31-circle")

knitr::opts\_chunk$set(echo = TRUE,fig.width=8,fig.height=4,fig.cap='',fig.align='center',echo=FALSE,dpi=72\*2)#, global.par = TRUE)

options(width=150)

suppressPackageStartupMessages(library(tidyverse))

suppressPackageStartupMessages(library(magrittr))

suppressPackageStartupMessages(library(knitr))

## Packages used for this post

suppressPackageStartupMessages(require(nleqslv))

suppressPackageStartupMessages(require(viridis))

##Configuration

options(knitr.table.format = "html")

theme\_set(theme\_minimal())

#if there are more than n rows in the tibble, print only the first m rows.

options(tibble.print\_max = 10, tibble.print\_min = 5)

```

## Abstract:

In 2007 Alexander Overwijk went viral with his 'Perfect Circle'

video. The same year a World Freehand Circle Drawing Championship was

organized which he won. In this post we show how a mobile camera, R

and the imager package can be used to develop an image analysis based

method to judge future instances of the championship.

<center>

```{r,results='asis',echo=FALSE,fig.cap="The cable problem"}

cat(paste0("![]({{ site.baseurl }}/",knitr::opts\_chunk$get("fig.path"),"HUD-1.png"),")")

```

</center>

{% include license.html %}

## Introduction

A few years back I watched with awe the 2007 video of

[Alexander Overwijk](https://twitter.com/AlexOverwijk)'s freehand

drawing a 1m diameter circle:

<center>

<iframe width="560" height="315" src="https://www.youtube.com/embed/eAhfZUZiwSE" frameborder="10" allow="autoplay; encrypted-media" allowfullscreen></iframe>

</center>

<FONT COLOR="bbbbbb">Note: Depending on your browser you might need to click "[Watch on Youtube](https://www.youtube.com/watch?v=eAhfZUZiwSE)" to see the video.</FONT>

<p>

Ever since watching that video I have wondered how one would go about to

judge the winner of such an alleged

[World Freehand Circle Drawing Championship](https://www.youtube.com/watch?v=u1J5ANnq0T8)

(WFHCDC).

:

\*They have a laser machine called the \*\*circleometer\*\* that creates the

perfect circle closest to the one you drew. The circleometer then

calculates the difference in area between the laser circle and the

circle that you drew. The machine then calibrates the area difference

as if you had drawn a circle with radius one meter. The person with

the smallest area difference is declared the world freehand circle

drawing champion.\*

Aha! Imaginary circleometers are expensive and my dean of study most

likely isn't open for investing in perfect circle measurement

equipment... So here is a cheaper solution involving a mobile device

camera, [R](https://www.r-project.org) and the

[`imager`](https://cran.r-project.org/web/packages/imager/index.html)

package by

[Simon Barthelmé](https://sites.google.com/site/simonbarthelme/) et

al. Altogether a combination of modern \*\*data science\*\* tools, which my dean of study is

most likely to approve! We'll use a screenshot from the perfect

circle video as motivating example to guide through the 3 phases of

the method:

1. Image rectification

2. Freehand circle identification and perfect circle estimation

3. Quantifying deviation from the perfect circle

We start by loading the screenshot into R using `imager`:

```{r LOADIMAGE, echo=TRUE, message=FALSE, warning=FALSE}

library("imager")

file <- "circle2.png"

img <- imager::load.image(file.path(fullFigPath, file))

```

```{r PLOTSCREENSHOT}

plot(img)

```

## Image rectification

The image clearly suffers from perspective distortions caused by the

camera being positioned to the right of the circle and, hence, not being

orthogonal to the blackboard plane. Furthermore, small lense

distortions are also visible - for example the right vertical line of

the blackboard arcs slightly. Since the video contains no details

about what sort of lense equipment was used, for the sake of

simplicity, we will ignore lense distortions in this post. If such

information is available one can use a program such as

[RawTherapee](http://rawtherapee.com) (available under a GNU GPL v3

license) to read the EXIF information in the meta data of the image and

automatically correct for lens distortion.

To rectify the image we estimate the parameters of the 2D projection

based on 4 ground control points (GPC). We use R's `locator` function

to determine the pixel location of the four corner points of the

blackboard in the image, but could just as well use any image analysis

program such as [Gimp](https://www.gimp.org). Furthermore, we need the

true object coordinates of these GPC. Unfortunately, these are only

approximately available to due lack of knowledge of the size of the

blackboard in the classroom. As a consequence a \*guesstimate\* of the

horizontal length is used.

```{r AQUIREGPC, echo=TRUE, eval=FALSE}

plot(img)

p <- locator(4)

p <- round(cbind(p$x, p$y))

dump(list=c("p"), "")

```

```{r DEFINEGPC}

## Corner coordinates of blackboard in image (found with Gimp or locator())

p <- rbind(c(0,318),c(2016,100),c(1988,1242),c(0,980))

## True coordinates - since we don't know the exact measurements of the

## blackboard we have to guess a little. Move coordinates to the right, s.t. all

## object coordinates are positive (otherwise they will not appear in the image.

## Offset to the right here: 1000 pixels

dx\_blackboard <- 1200

pp <- rbind(c(-dx\_blackboard ,100),c(2016,100),c(1988,1242),c(-dx\_blackboard,1242)) + matrix(c(dx\_blackboard, 0),4,2,byrow=TRUE)

```

These points are now used to rectify the image by a Direct Linear Transformation (DLT)

based on exactly 4 control points

[@hartley\_zisserman2004, Chapter 4]^[Alternatively, see slide 18 and onward in https://ags.cs.uni-kl.de/fileadmin/inf\_ags/3dcv-ws11-12/3DCV\_WS11-12\_lec04.pdf]. That

is the parameters of the 3x3 transformation matrix $H$ in homogeneous

coordinates are estimated

such that $p' = H p$, see the [code](`r paste0("https://raw.githubusercontent.com/hoehleatsu/hoehleatsu.github.io/master/\_source/",current\_input())`)

on github for details.

```{r DLT, results='hide'}

## Direct Linear Transformation (DLT) algorithm using inhomogeneous

## solution, i.e. H[3,3] = 1 and w\_i=1. See Sect 4.1.2 of

## Hartley and Zisserman (2004)

##

## (H[1,1]\*x + H[1,2]\*y + H[1,3]) - x' \* (H[3,1]\*x + H[3,2]\*y + 1) = 0

## (H[2,1]\*x + H[2,2]\*y + H[2,3]) - y' \* (H[3,1]\*x + H[3,2]\*y + 1) = 0

##

## [[ x y 1 0 0 0 -x'\*x -x'\*y = x'] ,

## [ 0 0 0 x y 1 -y'\*x -y'\*y = y']]

A <- list()

b <- list()

##Build RHS and LHS matrix components for each point pair

for (i in 1:4) {

A[[paste0(i)]] <- matrix(c(

p[i,1],p[i,2],1, 0, 0, 0, -pp[i,1]\*p[i,1], -pp[i,1]\*p[i,2],

0, 0, 0, p[i,1], p[i,2], 1, -pp[i,2]\*p[i,1], -pp[i,2]\*p[i,2]),

2,8, byrow=TRUE)

b[[paste0(i)]] <- matrix(c(pp[i,1], pp[i,2]),2,1)

}

##Glue matrices together

A\_matrix <- do.call(rbind, A)

b\_matrix <- do.call(rbind, b)

##Solve equation system

h\_vec <- solve(A\_matrix, b\_matrix)

##Form H matrix from the solution

H <- matrix(c(h\_vec,1), 3,3, byrow=TRUE)

H

```

We can implement the rectifying transformation using the

`imager::warp` function:

```{r RECTIFY, echo=TRUE}

##Transform image coordinates (x',y') to (x,y), i.e. note we specify

##the back transformation p = H \* p', so H here is the inverse.

map.persp.inv <- function(x,y, H) {

out\_image <- H %\*% rbind(x,y,1)

list(x=out\_image[1,]/out\_image[3,], y=out\_image[2,]/out\_image[3,])

}

##Pad dx\_blackboard pixels to the right to make space for the blackboard coming closer

img\_padded <- pad(img, nPix=dx\_blackboard, axes="x", pos=1)

##Warp image

warp <- imwarp(img\_padded, map=function(x,y) map.persp.inv(x,y,solve(H)),coordinates="absolute", direction="backward")

```

The result looks as follows:

```{r PLOTRECTIFICATION, fig.height=8}

layout(c(1,2))

plot(img, main="Original")

lines(c(p[,1],p[1,1]), c(p[,2],p[1,2]), col="magenta",lwd=2)

points(p[,1], p[,2], cex=3, pch=20, col="steelblue")

plot(warp, main="Rectified")

lines(c(pp[,1],pp[1,1]), c(pp[,2],pp[1,2]), col="magenta",lwd=2)

points(pp[,1], pp[,2], cex=3, pch=20, col="steelblue")

```

Please notice the different x-axes of the two images when comparing

them. For faster computation and better visualization in the

remainder of this post, we crop the x-axis of the image to the

relevant parts of the circle.

```{r, echo=TRUE}

warp <- imsub(warp, x %inr% c(dx\_blackboard, nrow(warp)))

```

```{r PLOTRECTIFIED}

plot(warp, main="Rectified and Cropped")

```

## Freehand circle identification

As described in the `imager`

[edge detection tutorial](https://dahtah.github.io/imager/canny.html)

we use length of the gradient to determine the edges in the

image. This can be done by applying filters to the image.

```{r, echo=TRUE}

##Edge detection function. Sigma is the size of the blur window.

detect.edges <- function(im, sigma=1) {

isoblur(im,sigma) %>% imgradient("xy") %>% enorm() %>% imsplit("c") %>% add

}

#Edge detection filter sequence.

edges <- detect.edges(warp,1) %>% sqrt

```

To detect the circle from this we specify a few seed points for a

[watershed](https://en.wikipedia.org/wiki/Watershed\_%28image\_processing%29)

algorithm with a priority map inverse proportional to gradient magnitude. This includes a few points inside and outside the

circle and a few points \*on\* the circle. Note: a perfect circle

would have no border, but when drawing a circle with a piece of chalk

it's destined to have a thin border line.

```{r}

#Priority map with priority inverse proportional to gradient magnitude

pmap <- 1/(1+edges)

```

```{r, eval=FALSE}

layout(1)

plot(pmap)

background <- locator(2)

background <- round(cbind(background$x, background$y))

dump("background","")

foreground <- locator(2)

foreground <- round(cbind(foreground$x, foreground$y))

dump("foreground","")

```

```{r SEGMENTATION, results='hide'}

##Seed image

seeds <- imfill(dim=dim(pmap))

background <- structure(c(1400, 956, 239, 507), .Dim = c(2L, 2L))

foreground <- structure(c(508, 1078, 1018, 234), .Dim = c(2L, 2L))

seeds[cbind(background,1,1)] <- 1

seeds[cbind(foreground,1,1)] <- 2

##Number of seed points

sum(seeds >0)

##Run watershed algorithm

wt <- watershed(seeds,pmap)

mask <- add.colour(wt) #We copy along the three colour channels

layout(t(1:2))

plot(warp \* (mask==1),main="Background")

points(background, col="steelblue", pch=20)

plot(warp \* (mask==2),main="Foreground")

points(foreground, col="magenta", pch=20)

```

We can now extract the circle by:

```{r, echo=TRUE}

##Just the circle

freehandCircle <- (warp \* (mask==2) > 0) %>% grayscale

##Total area covered by the circle

freehandDisc <- label(freehandCircle, high\_connectivity=TRUE) > 0

```

...and by morphological operations we get just the outer border as a hairline

```{r THINBORDER, echo=TRUE}

dilatedDisc <- freehandDisc %>% dilate\_rect(sx=3,sy=3)

freehandCircleThinBorder <- (freehandDisc - dilatedDisc) != 0

```

```{r PLOTTHINBORDER, fig.width=8, fig.height=4}

par(mar=c(2,2,2,2))

plot(1-freehandCircleThinBorder, main="Extracted freehand circle")

```

## Perfect circle estimation

Once the freehand circle path in the image has been identified, we need

to find the best fitting \*perfect\* circle matching this path. This problem is

elegantly solved by @coope1993, who formulates the problem as finding

center and radius of the circle minimizing the squared

Euclidean distance to $m$ data points $a\_j$, $j=1,\ldots,m$.

Denoting by $c$ the center of the circle and by $r>0$ the radius we

want to find the solution of

$$

\min\_{c\in \mathbb{R^2}, r>0} \sum\_{j=1}^m F\_j(c,r)^2, \quad\text{where}\quad F\_j(c,r) = \left|r - ||c-a\_j||\_2\right|,

$$

and $||x||\_2$ denotes Euclidean distance. Because the curve fitting

minimizes the distance between an observed point $a\_j$ and its closest

point on the circle and thus involves both the $x$ and the $y$

direction , this is a so called

\*\*[total least squares](https://en.wikipedia.org/wiki/Total\_least\_squares)\*\*

problem. The problem is non-linear and can only be solved by iterative

numerical methods. However, the dimension of the parameter space can

be reduced by one, because given the center $c$ we can determine that

$r(c)=\frac{1}{m} \sum\_{j=1}^m ||c-a\_j||\_2$.

```{r FITCIRCLE, echo=TRUE}

##Compute radius given center

radius\_given\_center <- function(center, dist=NULL) {

if (is.null(dist)) {

a <- as.matrix(where(freehandCircleThinBorder > 0))

dist <- sqrt((a[,1] - center[1])^2 + (a[,2] - center[2])^2)

}

return(mean(dist))

}

##Target functin of the total least squares criterion of Coope (1993)

target\_tls <- function(theta) {

##Extract parameters

center <- exp(theta[1:2])

##Total least squares criterion from Coope (1993)

a <- as.matrix(where(freehandCircleThinBorder > 0))

dist <- sqrt((a[,1] - center[1])^2 + (a[,2] - center[2])^2)

##Compute radius given center

radius <- radius\_given\_center(center, dist)

F <- abs( radius - dist)

sum(F^2)

}

res\_tls <- optim(par=log(c(x=background[1,1], y=background[1,2])), fn=target\_tls)

center <- exp(res\_tls$par)

fit\_tls <- c(center,radius=radius\_given\_center(center))

fit\_tls

```

We illustrate the freehand circle (in black) and the fitted circle

(magenta) on top of each other using the alpha channel. You have to

study the image carefully to detect differences between the two curves!

```{r DRAWCIRCLE, fig.width=8, fig.height=4}

par(mar=c(2,2,0,2))

nPoints <- 1000

t <- seq(0,2\*pi, length=nPoints)

circle <- cbind(fit\_tls[3]\*cos(t) + fit\_tls[1], fit\_tls[3]\*sin(t)+fit\_tls[2])

plot(1-freehandCircleThinBorder)

lines(circle[,1], circle[,2], col=rgb(0.8,0.4,0.9,0.3),lwd=3)

points(fit\_tls[1], fit\_tls[2], pch=20, cex=1, col=rgb(0.8,0.4,0.9,0.4))

```

## Quantifying the circularness of the freehand circle

We quantify the \*\*circularness\*\* of the freehand circle by contrasting

the area covered by it with the area of the fitted perfect circle. The

closer this ratio is to 1 the more perfect is the freehand circle.

```{r, echo=TRUE}

##Area of the freehand drawn disc

areaFreehandDisc <- sum(freehandDisc)

##Area of the disc corresponding to the idealized circle fitted

##to the freehand circle

areaIdealDisc <- pi \* fit\_tls["radius"]^2

##Ratio between the two areas

ratio\_area <- as.numeric(areaFreehandDisc / areaIdealDisc)

ratio\_area

```

Yup, it's a pretty perfect circle! Since the fitted circle already

takes the desired shape into account, my intuition is that this ratio

is a pretty good way to quantify circularness. However, to avoid

\*\*measurehacks\*\*, we use as backup measure the circleometer approach:

for each point on the freehand circle we measure its distance to the

freehand circle and integrate/sum this up over the path of the

freehand circle. We can approximate this integration using image

pixels as follows.

```{r DISTANCE, echo=TRUE}

##Create a pixel based circle in an image of the same size as the

##freehandCircle img. For visibility we use a border of 'border' pixels

##s.t. circle goes [radius - border/2, radius + border/2].

Circle <- function(center, radius, border) {

as.cimg(function(x,y) {

lhs <- (x-center[1])^2 + (y-center[2])^2

return( (lhs >= (radius-border/2)^2) & (lhs <= (radius+border/2)^2))

}, dim=dim(freehandCircle))

}

##Build pixel circle based on the fitted parameters

C\_tls <- Circle(fit\_tls[1:2], fit\_tls[3], border=1)

##Calculate Euclidean distance to circle for each pixel in the image

dist <- distance\_transform(C\_tls, value=1, metric=2)

##Distance between outer border of freehand circle and perfect circle

area\_difference <- sum(dist[freehandCircleThinBorder>0])

##Compute area difference and scaled it by the area of the fitted disc

ratio\_areadifference <- as.numeric(area\_difference / areaIdealDisc)

```

The image below illustrates this by overlaying the result on top of

the distance map. For better visualization we zoom in on the 270-300

degree part of the circle (i.e. the bottom right). In magenta is the

fitted perfect circle, in gray the freehand circle and the area

between the two paths is summed up over the entire path of the

freehand circle:

```{r PLOTAREADIFF}

CircleDisc <- function(center, radius, border) {

as.cimg(function(x,y) {

lhs <- (x-center[1])^2 + (y-center[2])^2

return( lhs <= (radius+border/2)^2)

}, dim=dim(freehandCircle))

}

CDisc <- CircleDisc(fit\_tls[1:2], fit\_tls[3], border=1)

xr <- c(fit\_tls[1], fit\_tls[1] + 0.7\*fit\_tls[3])

yr <- c(fit\_tls[2]+0.7\*fit\_tls[3], fit\_tls[2] + 1.1\*fit\_tls[3])

distsub <- imsub(dist, x%inr% xr, y %inr% yr)

par(mar=c(0,0,0,0))

plot( distsub/max(distsub) + imsub(freehandDisc \* (CDisc==0), x %inr% xr, y %inr% yr), axes=FALSE)

highlight(imsub(C\_tls>0, x %inr% xr, y %inr% yr), col="magenta")

highlight(imsub(freehandCircleThinBorder, x %inr% xr, y %inr% yr), col="gray")

```

We obtain `ratio\_areadifference`= `r sprintf("%.5f",ratio\_areadifference)`. Thus also this measure tells

us: it's a pretty perfect circle! To summarise: The output on the

display of the judge's Circle-O-Meter App (available under a GPL v3

license) at the World Freehand Circle Drawing Championship would be as

follows: `r emo::ji("wink")`

```{r HUD}

##Show our findings Head-up-display (HUD) style on top of image

par(mar=c(0,0,3,0))

plot(img, axes=FALSE,main="Athlete: Alexander Overwijk (CAN), Rank: 1")

lines(c(p[,1],p[1,1]), c(p[,2],p[1,2]), col="magenta",lwd=2)

points(p[,1], p[,2], cex=3, pch=20, col="steelblue")

##Points of the circle

t <- seq(0,2\*pi, length=nPoints)

circle <- cbind(fit\_tls[3]\*cos(t) + fit\_tls[1], fit\_tls[3]\*sin(t)+fit\_tls[2])

circle[,1] <- circle[,1] + dx\_blackboard

##Transform and show points of circle

p\_circle <- map.persp.inv(circle[,1], circle[,2], H=solve(H))

lines(p\_circle$x, p\_circle$y, col=rgb(0.8,0.8,0.8,0.5),lwd=3, lty=2)

##Transform and show center

p\_center <- map.persp.inv(fit\_tls[1] + dx\_blackboard, fit\_tls[2], H=solve(H))

points(p\_center$x, p\_center$y, pch=20, cex=1, col=rgb(0.8,0.8,0.8,0.8))

##Add HUD text

text(p\_center$x, p\_center$y - 100, substitute(r[A] == x, list(x=sprintf("%.2f%%",100\*ratio\_area))), col=rgb(0.8,0.8,0.8,0.8))

text(p\_center$x, p\_center$y + 100, substitute(r[Delta \* "A"] == x, list(x=sprintf("%.2f%%",100\*ratio\_areadifference))), col=rgb(0.8,0.8,0.8,0.8))

```

## Discussion

We took elements of computer vision, image analysis and total least

squares to segment a chalk-drawn circle on a blackboard and provided

measures of it's circularness. Since we did not have direct access to

the measurements of the blackboard in object space, a little

guesstimation was necessary, nevertheless, the results show that it

was a pretty circular freehand circle!

With the machinery in place for judging freehand circles, its time to

send out the call for contributions to the \*\*2nd World Freehand Circle

Drawing Championship\*\* (online edition). Stay tuned for the call:

You can spend the anxious waiting time practicing your freehand 1m

diameter circles - it's a good way to loosen up long and unproductive

meetings!

## Appendix

If we instead of the total sum of squares criterion involving

$F\_j(c,r)$ mentioned in the text solve the related criterion

$$

\sum\_{j=1}^m f\_j(c,r)^2, \quad\text{where}\quad f\_j(c,r) =

||c-a\_j||\_2^2 - r^2,

$$

then a much simpler solution emerges. @coope1993 explains that

this alternative criterion geometrically corresponds to minimizing the

product

$$

\text{(distance to the closest point on the circle)}\times \text{(distance to

the furthest away point point on the circle)}

$$

over the measurement point. In order to obtain the solution write the

residuals $f\_j$ as $f\_j(c,r) = c^T c - 2 c^T a\_j +

a\_j^T a\_j - r^2$ and perform a change of variables from $(c\_1, c\_2,

r)'$ to

$$

y =

\left[

\begin{matrix}

2 c\_1 \\

2 c\_2 \\

r^2 - c^T c \\

\end{matrix}

\right]

\quad \text{and let} \quad

b\_j =

\left[

\begin{matrix}

a\_{j1} \\

a\_{j2} \\

1

\end{matrix}

\right].

$$

The minimization problem then becomes

$$

\min\_{y \in \mathbb{R}^3} \sum\_{j=1}^m \left\{ a\_j^T a\_j - b\_j^T y \right\},

$$

which can be written as a linear least square (LLS) expression

$$

\min\_y ||By - d||\_2^2,

$$

where $B$ is a $3\times m$ matrix with the $b\_j$-vectors as columns

and $d=||a\_j||\_2^2$. This expression is then easily solved using the standard

least squares machinery.

```{r, echo=TRUE}

##Fast linear least squares problem as described in Coope (1993)

fitCircle\_lls <- function(freehandCircle) {

a <- as.matrix(where(freehandCircle > 0))

b <- cbind(a,1)

B <- b

d <- a[,1]^2 + a[,2]^2

y <- solve(t(B) %\*% B) %\*% t(B) %\*% d

x <- 1/2\*y[1:2]

r <- as.numeric(sqrt(y[3] + t(x) %\*% x))

return(c(x=x[1], y=x[2], radius=r))

}

##Fit using linear least squares procedure of Coole (1993)

fit\_lls <- fitCircle\_lls(freehandCircleThinBorder)

##Compare TLS and LLS fit

rbind(lls=fit\_lls,tls=fit\_tls)

```

In other words: the results are nearly identical.

```{r, eval=FALSE}

layout(1)

C\_lls<- Circle(center=fit\_lls[1:2], radius=fit\_lls[3],border=1)

plot(colorise(1-freehandCircleThinBorder, C\_tls>0, col="blue", alpha=0.5))

highlight(C\_lls, col="magenta")

```

## Literature

We can implement the rectifying transformation using the imager::warp function:

##Transform image coordinates (x',y') to (x,y), i.e. note we specify

##the back transformation p = H \* p', so H here is the inverse.

map.persp.inv <- function(x,y, H) {

out\_image <- H %\*% rbind(x,y,1)

list(x=out\_image[1,]/out\_image[3,], y=out\_image[2,]/out\_image[3,])

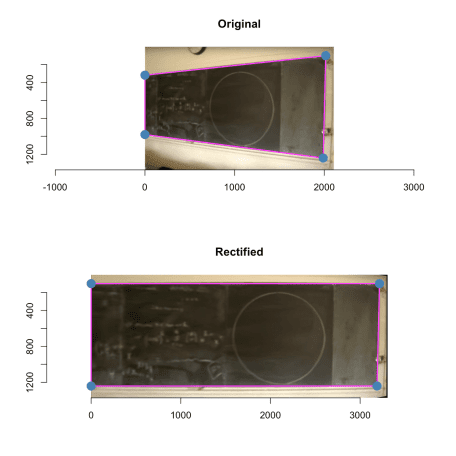
}

##Pad dx\_blackboard pixels to the right to make space for blackboard

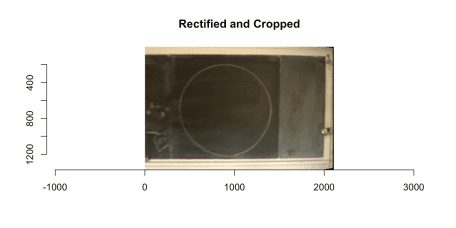
img\_padded <- pad(img, nPix=dx\_blackboard, axes="x", pos=1)

##Warp image

warp <- imwarp(img\_padded, map=function(x,y) map.persp.inv(x,y,solve(H)),coordinates="absolute", direction="backward")

The result looks as follows: Please notice the different x-axes of the two images when comparing them. For faster computation and better visualization in the remainder of this post, we crop the x-axis of the image to the relevant parts of the circle.

warp <- imsub(warp, x %inr% c(dx\_blackboard, nrow(warp)))



**Freehand circle identification**

As described in the imager edge detection tutorial we use length of the gradient to determine the edges in the image. This can be done by applying filters to the image.

##Edge detection function. Sigma is the size of the blur window.

detect.edges <- function(im, sigma=1) {

isoblur(im,sigma) %>% imgradient("xy") %>% enorm() %>% imsplit("c") %>% add

}

#Edge detection filter sequence.

edges <- detect.edges(warp,1) %>% sqrt

Edge Detection Tutorial

# 1 Step I: denoising

Noise in the image can cause illusory edges to be detected. The traditional recommendation is Gaussian filtering, which is easy enough:

**library**(imager)

im <- grayscale(boats) %>% isoblur(2) *#2 pix. blur*

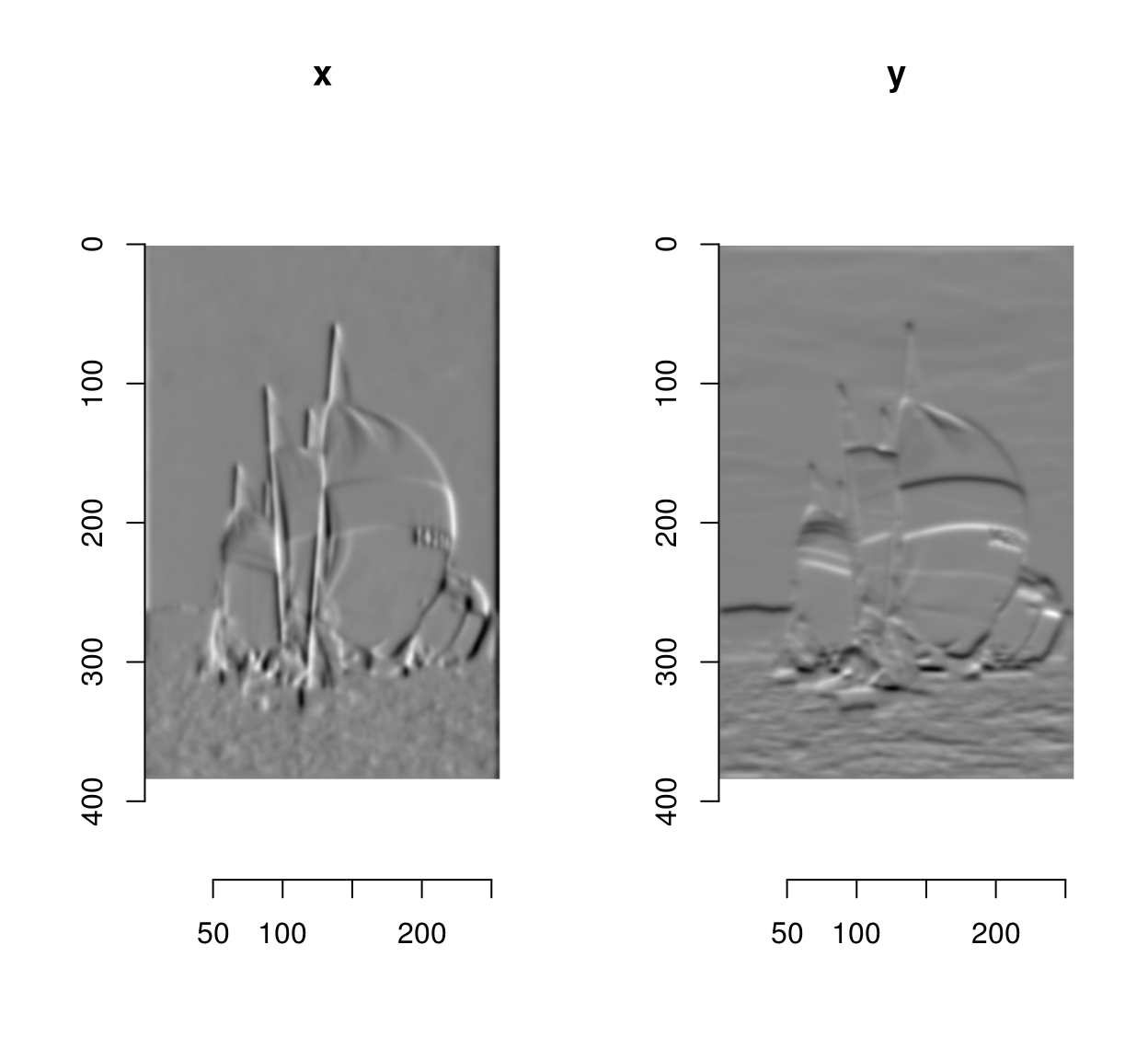
(NB: other filters may actually perform better here, for example the median filter).

# 2 Step II: computing the image gradient, its magnitude and angle

The next step is to compute an image gradient, which boils down to:

gr <- imgradient(im,"xy")

plot(gr,layout="row")

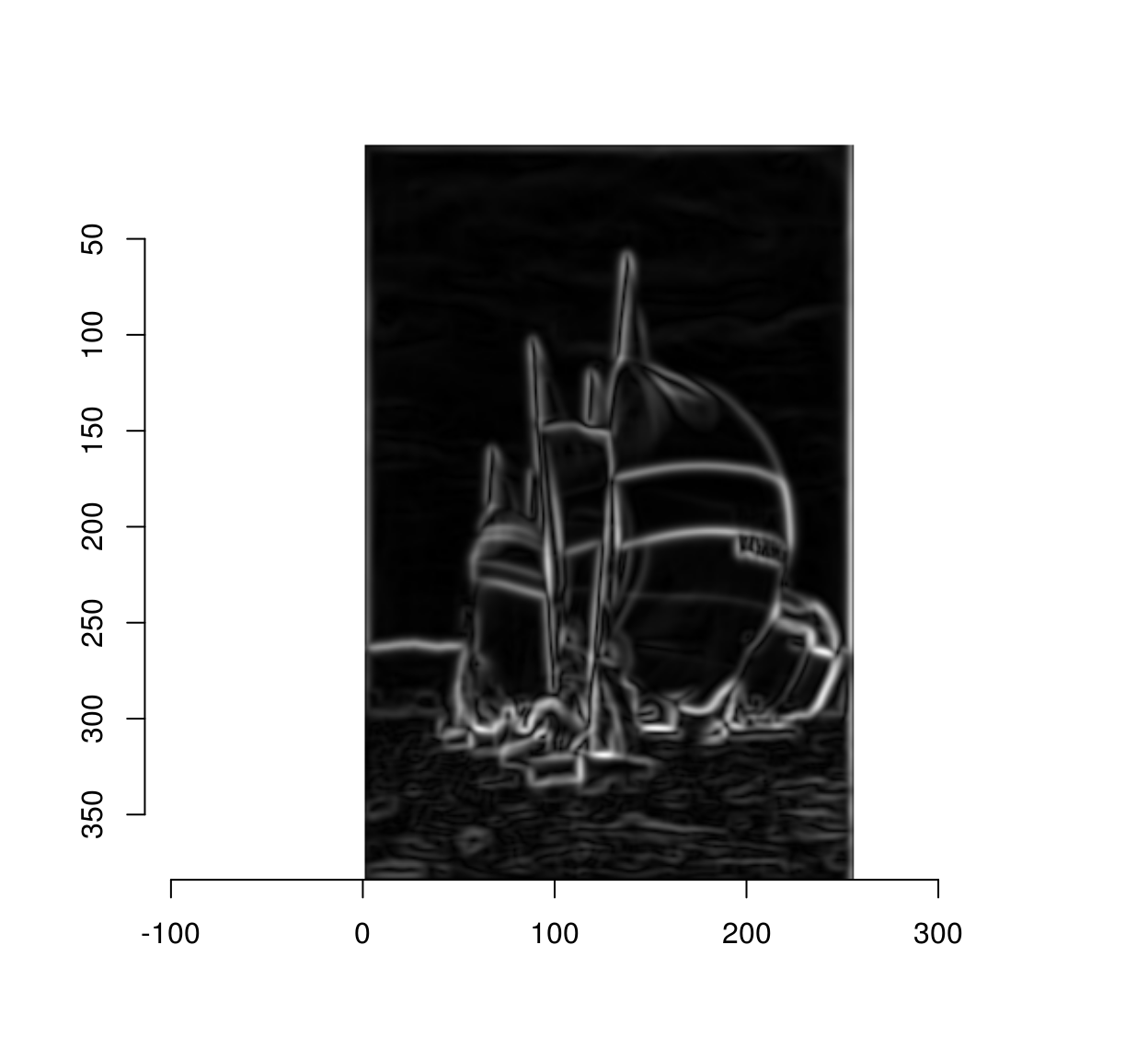


The gradient has two components (the image derivatives along xx and yy), and is stored as an image list, with two components named “x” and “y”.

We can compute the gradient magnitude via:

mag <- with(gr,sqrt(x^2+y^2))

plot(mag)

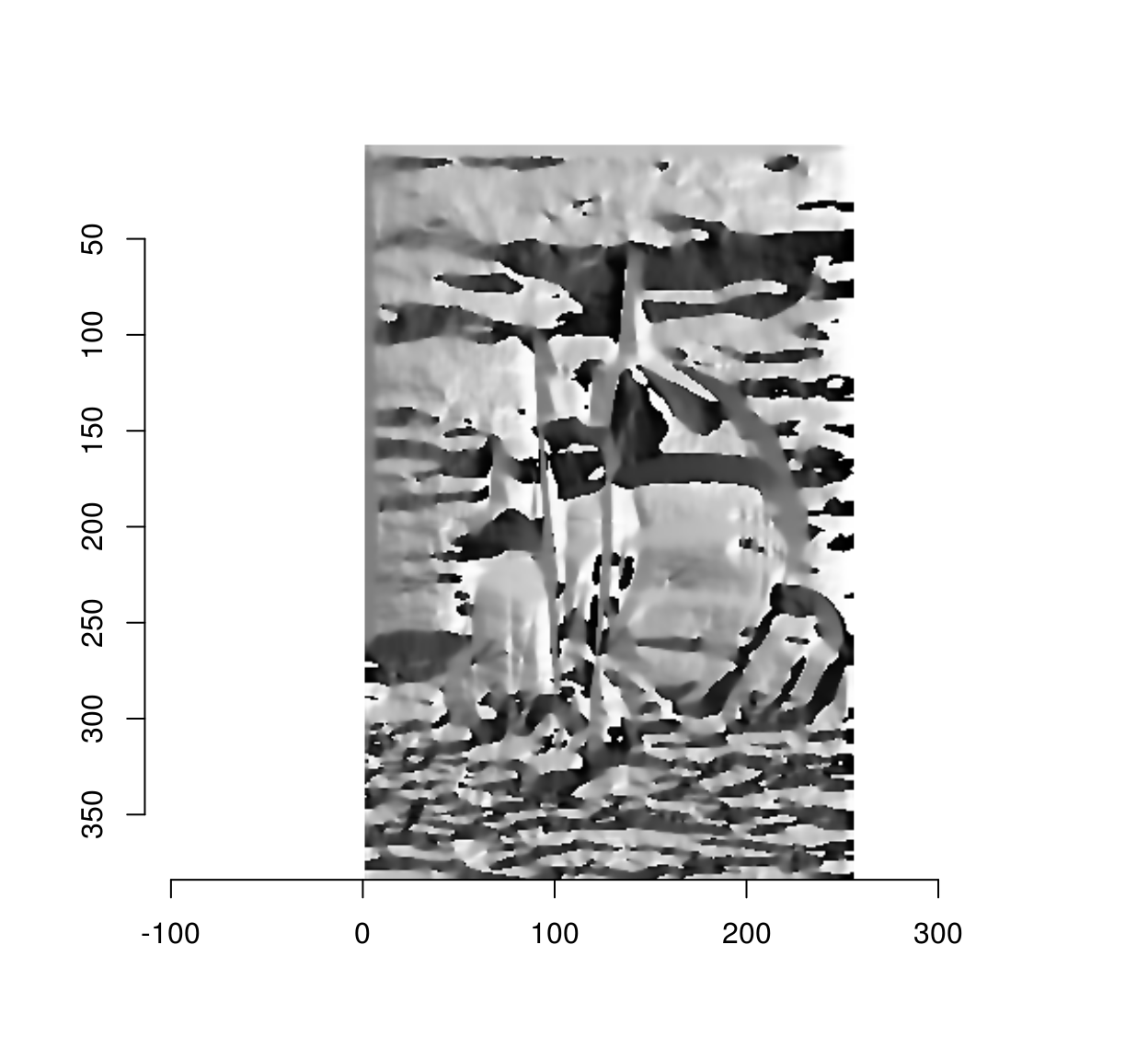


The Canny edge detector is essentially a cleaned-up version of the above picture.

The gradient angle determines the local orientation of image edges:

ang <- with(gr,atan2(y,x))

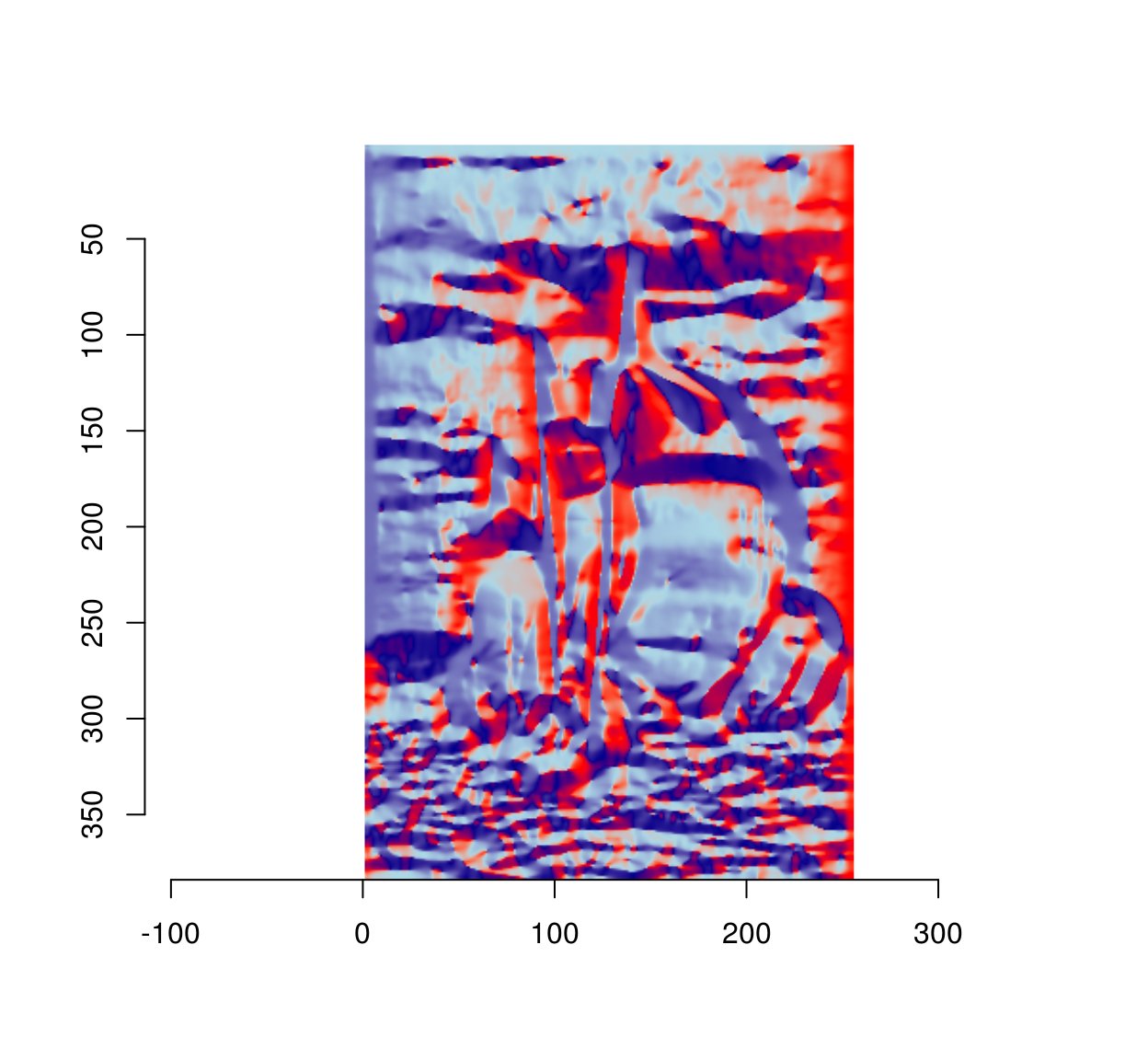
plot(ang)



The above figure looks interesting but is hard to read. One of the issues is that an angular variable (taking values in −π,π−π,π is mapped to a linear scale (black and white correspond to −π−π and ππ, but −π−π and ππ correspond to the same angle). Here it is on a circular colour scale:

cs <- scales::gradient\_n\_pal(c("red","darkblue","lightblue","red"),c(-pi,-pi/2,pi/2,pi))

plot(ang,colourscale=cs,rescale=FALSE)

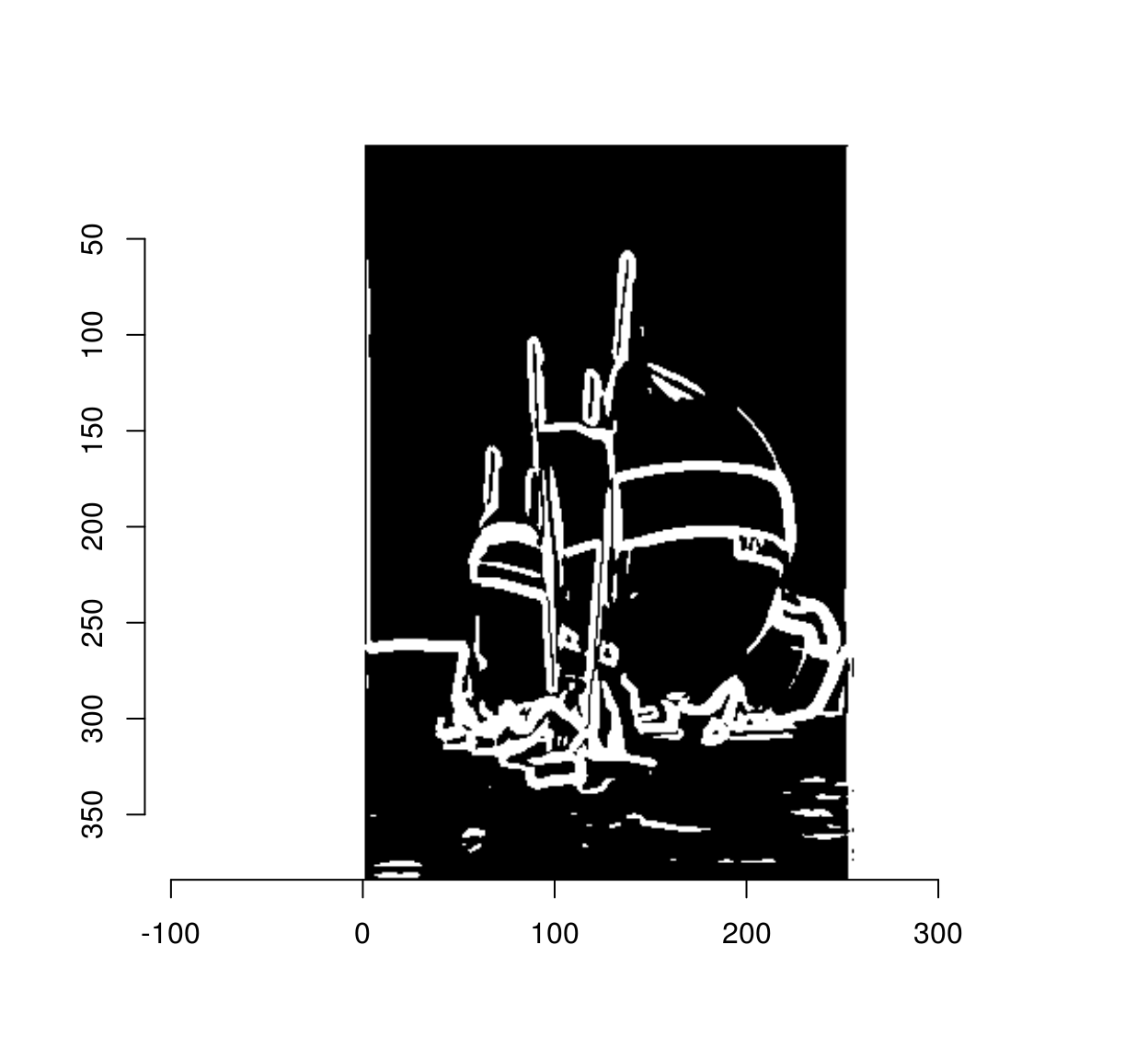


For a better way of visualising image gradients, see here.

# 3 Step III: cleaning up using non-maxima suppression

The gradient magnitude we plotted above is blurry, and if we threshold it directly we’ll see that some pixels near edges are also labelled as being edges:

threshold(mag) %>% plot



The goal on non-maxima suppression is to eliminate such false positives. To do so we set to 0 all pixels in the gradient magnitude image that are not local maxima along the direction of the gradient. In a C++ implementation you would loop through all the pixels, and check that all the pixels have higher values than their two neighbours in the direction of the gradient. In R that would take ages, so we need to find a way to vectorise that operation. The solution is to use interpolation as a our main vectorised operation. Interpolation returns a list of image values at a set of locations. What we’ll do is take the whole set of pixels, compute a corresponding set of neighbouring locations along the gradients, and see what the image values are at the new locations. For example, here we go see what’s what along the gradient:

*#Going along the (normalised) gradient*

*#Xc(im) is an image containing the x coordinates of the image*

nX <- Xc(im) + gr$x/mag

nY <- Yc(im) + gr$y/mag

*#nX and nY are not integer values, so we can't use them directly as indices.*

*#We can use interpolation, though:*

val.fwd <- interp(mag,data.frame(x=as.vector(nX),y=as.vector(nY)))

We can naturally also go backwards along the gradient:

nX <- Xc(im) - gr$x/mag

nY <- Yc(im) - gr$y/mag

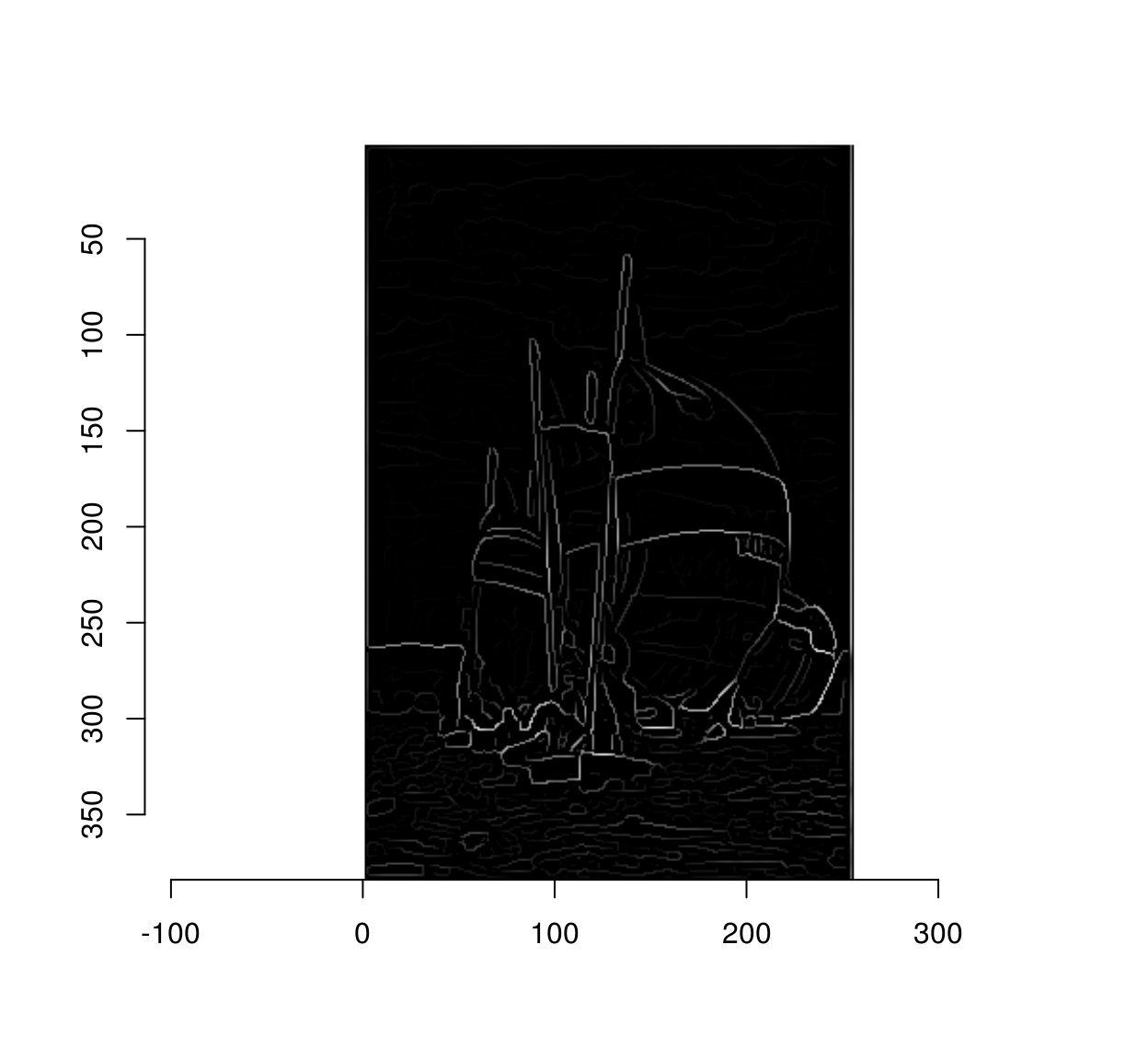
val.bwd <- interp(mag,data.frame(x=as.vector(nX),y=as.vector(nY)))

Given these two sets of values, non-maxima suppression comes down to killing all values that aren’t larger than their two neighbours along the flow:

throw <- (mag < val.bwd) | (mag < val.fwd)

mag[throw] <- 0

plot(mag)



# 4 Step IV: hysteresis

At this stage we’re beginning to see some nice outlines, but the ultimate goal is to classify each pixel as edge/non-edge, which means picking a threshold. Instead of picking a single threshold, Canny suggested picking a double threshold t1,t2t1,t2, with t1<t2t1<t2, where

* points with magnitudes above t2t2 are called “strong edges”
* points with magnitudes above t1t1 are called “weak edges”

*#strong threshold*

t2 <- quantile(mag,.96)

*#weak threshold*

t1 <- quantile(mag,.90)

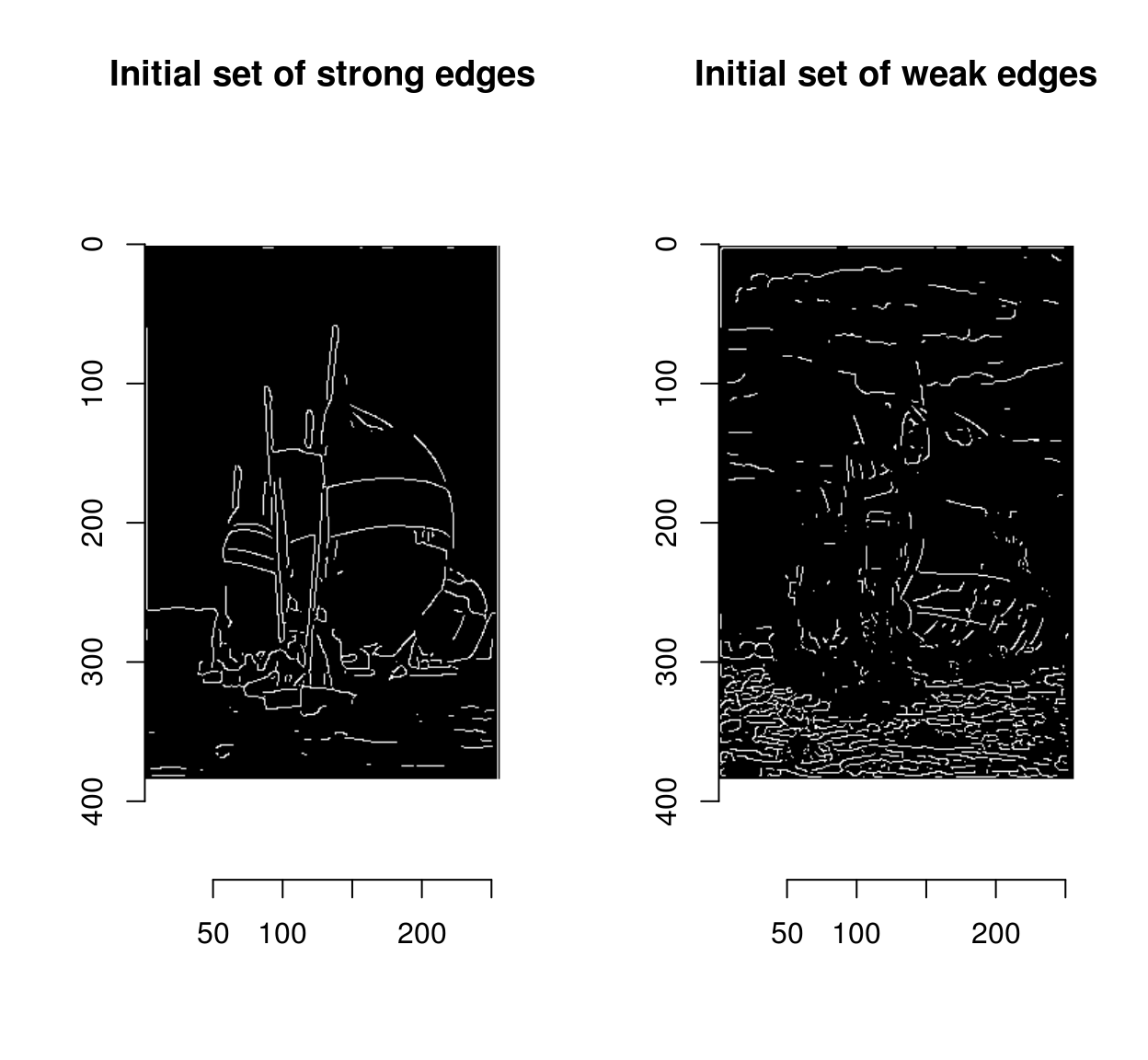
layout(t(1:2))

strong <- mag>t2

plot(strong,main="Initial set of strong edges")

weak <- mag %inr% c(t1,t2)

plot(weak,main="Initial set of weak edges")



Weak edges have a high chance of being false positives, but less so if there’s a strong edge somewhere nearby, because edges are usually continuous.

Hysteresis rescues such weak edges progressively. The usual implementation is iterative. First, we put all the strong edges in a stack (or queue). Then, we pop the first strong edge off the stack, look at all its neighbours, and if any of these neighbours is a weak edge you rescue it by labelling it as strong and adding it to the stack. Then we move on to the next item on the stack (or queue). When we run out of items in the queue we’re done.

hyst.loop <- **function**(strong,weak)

{

*#We make the queue a list so that it can grow or shrink relatively fast*

queue <- which(strong==1) %>% as.list

max.x <- width(strong)

max.y <- height(strong)

**while** (length(queue)>0)

{

ind <- queue[[1]]

*#get (x,y) coordinates of the current point*

cc <- coord.index(strong,ind)

*#explore the neighbourhood*

**for** (nx **in** (cc$x+c(-1,0,1)))

{

**for** (ny **in** (cc$y+c(-1,0,1)))

{

*#we have to mind boundary conditions*

**if** (nx > 0 && nx <= max.x && ny > 0 && ny <= max.y)

{

**if** (at(weak,nx,ny)==TRUE)

{

at(weak,nx,ny) <- FALSE

at(strong,nx,ny) <- TRUE

queue[[length(queue)+1]] <- index.coord(strong,data.frame(x=nx,y=ny))

}

}

}

}

queue[[1]] <- NULL

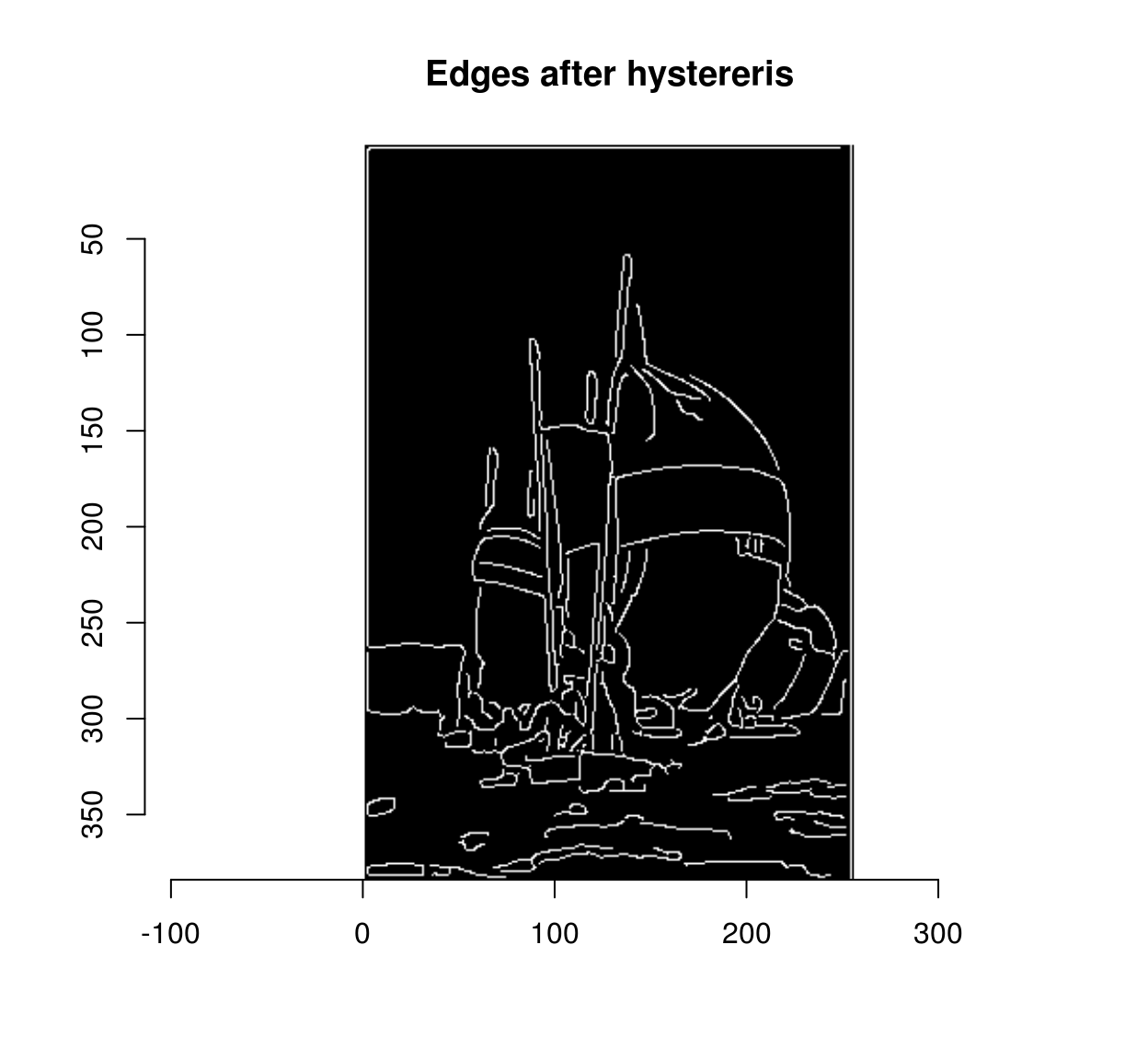
}

strong

}

canny <- hyst.loop(strong,weak)

plot(canny,main="Edges after hystereris")



R is impractically slow for such a process (try running hyst.loop on a large image).

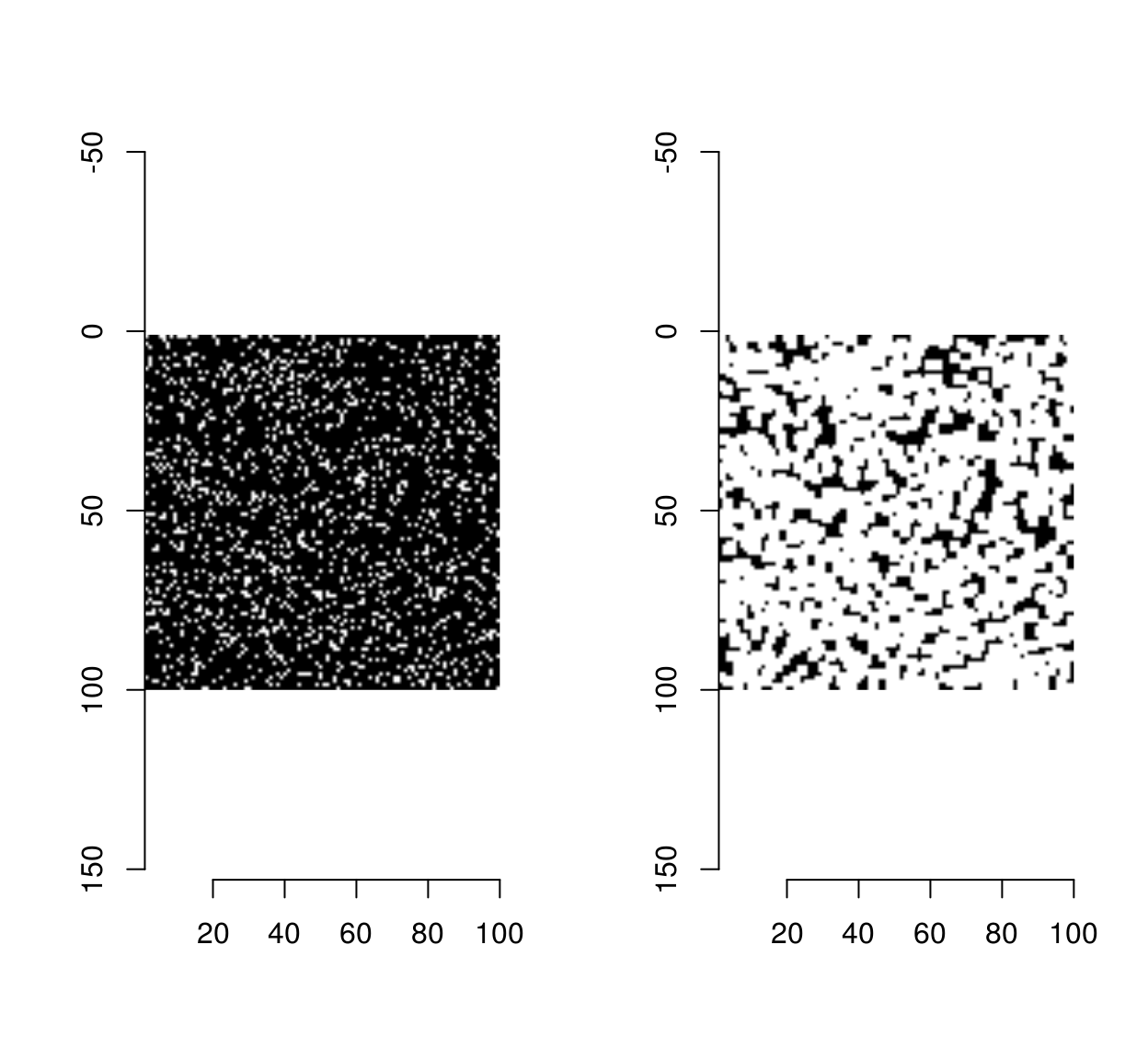
We can do much better by using morphological dilation as our computational primitive. Morphological dilation takes a B&W image and expands the set of white pixels by using a “structuring element”. In the simplest case the structuring element is a square and dilation corresponds to a kind of convolution: we move the square around, and we label the central element white if there is another white pixel in the square. This operation is called “grow” in imager, when applied to pixel sets. See here for more on morphology. Here it is at work on a small noise image:

px <- imnoise(100,100) > 1

layout(t(1:2))

plot(px,"Original")

plot(grow(px,3),"Dilated Set")

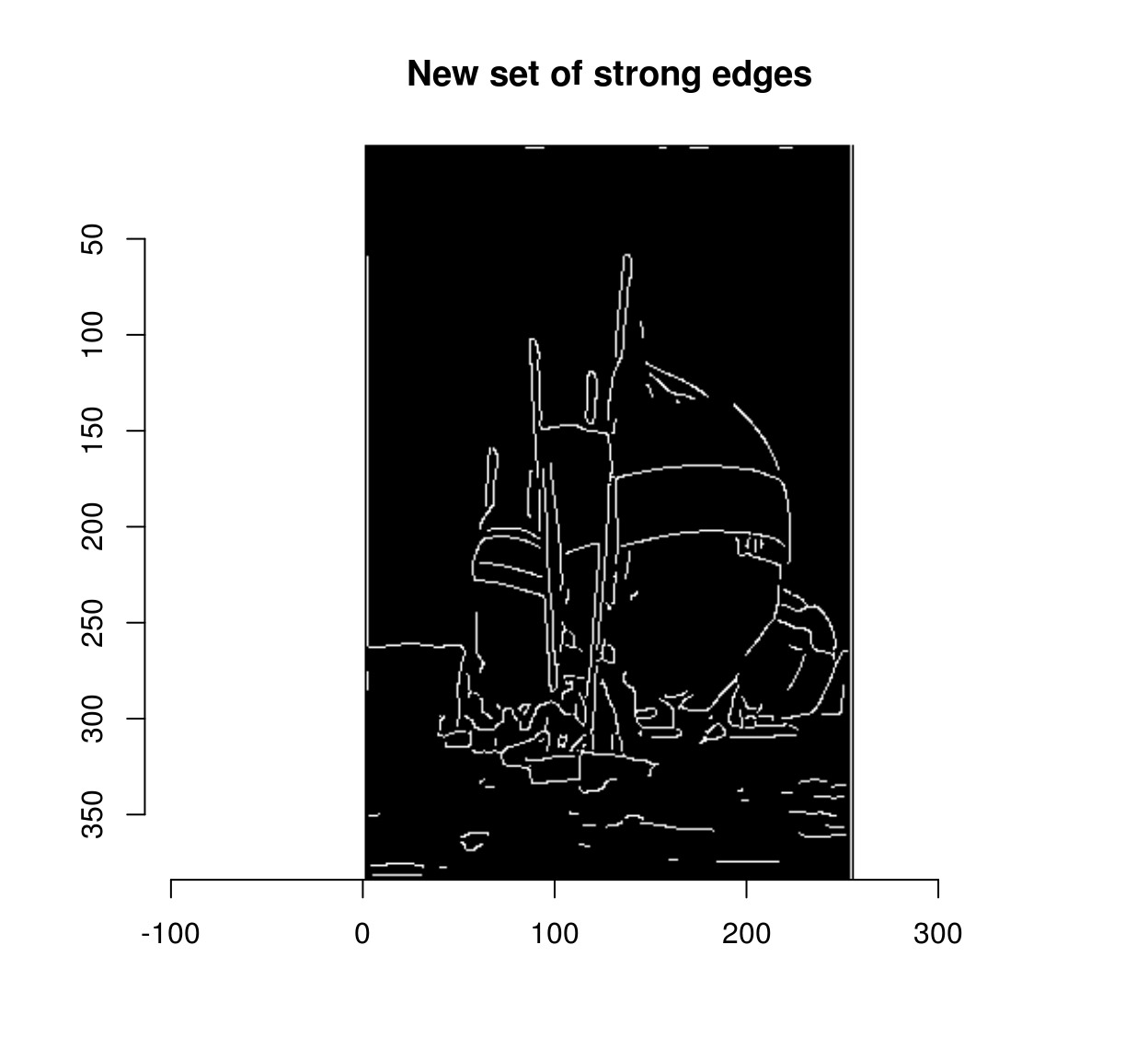


Morphological dilation gives us a way of implementing hysteresis. We start from the set of strong edges, expand it with dilation, and check its overlap with the set of weak-edges. Any points in the overlap can now be labelled “strong”:

overlap <- grow(strong,3) & weak

strong.new <- strong | overlap

plot(strong.new,main="New set of strong edges")



delta <- sum(strong.new)-sum(strong)

delta

## [1] 188

We have labelled 202 formerly weak edges as strong. There’s no reason to stop here: we can repeat the operation until the set of weak edges stops growing. The hysteresis operation is thus a fixed point iteration, an computation we go on doing until the results stop changing. The package “fixedpoint” provides a convenient mechanism for turning a function into a fixed point iteration.

*#run devtools::install\_github("dahtah/fixedpoints")*

**library**(fixedpoints)

*#Example of a fixed point iteration:*

*#divide a number by 2 until the result doesn't change*

f <- **function**(x) x/2

g <- fp(f)

g(3) *#Run the fixed point iteration from 3.*

## Result after 29 iterations. Converged: TRUE.

## [1] 5.587935e-09

g(0)

## Result after 1 iterations. Converged: TRUE.

## [1] 0

In our case the mapping to iterate is the “expansion, relabelling” operation we had above:

*#ws is a list containing two fields, "weak" and "strong"*

*#which are images where all pixels with value 1 are "weak edges" (resp. strong)*

*#the function expands the set of strong pixels via dilation and overlap*

*#and returns the expanded strong set and the (shrunk) weak set*

expandStrong <- **function**(ws)

{

overlap <- grow(ws$strong,3) & ws$weak

ws$strong[overlap] <- TRUE

ws$weak[overlap] <- FALSE

ws

}

*#hystFP is a new function that will call expandStrong repeatedly until*

*#the weak and strong sets don't change anymore*

hystFP <- fp(expandStrong)

*#Call hystFP*

out <- list(strong=strong,weak=weak) %>% hystFP

out

## Result after 71 iterations. Converged: TRUE.

## $strong

## Pixel set of size 5587. Width: 256 pix Height: 384 pix Depth: 1 Colour channels: 1

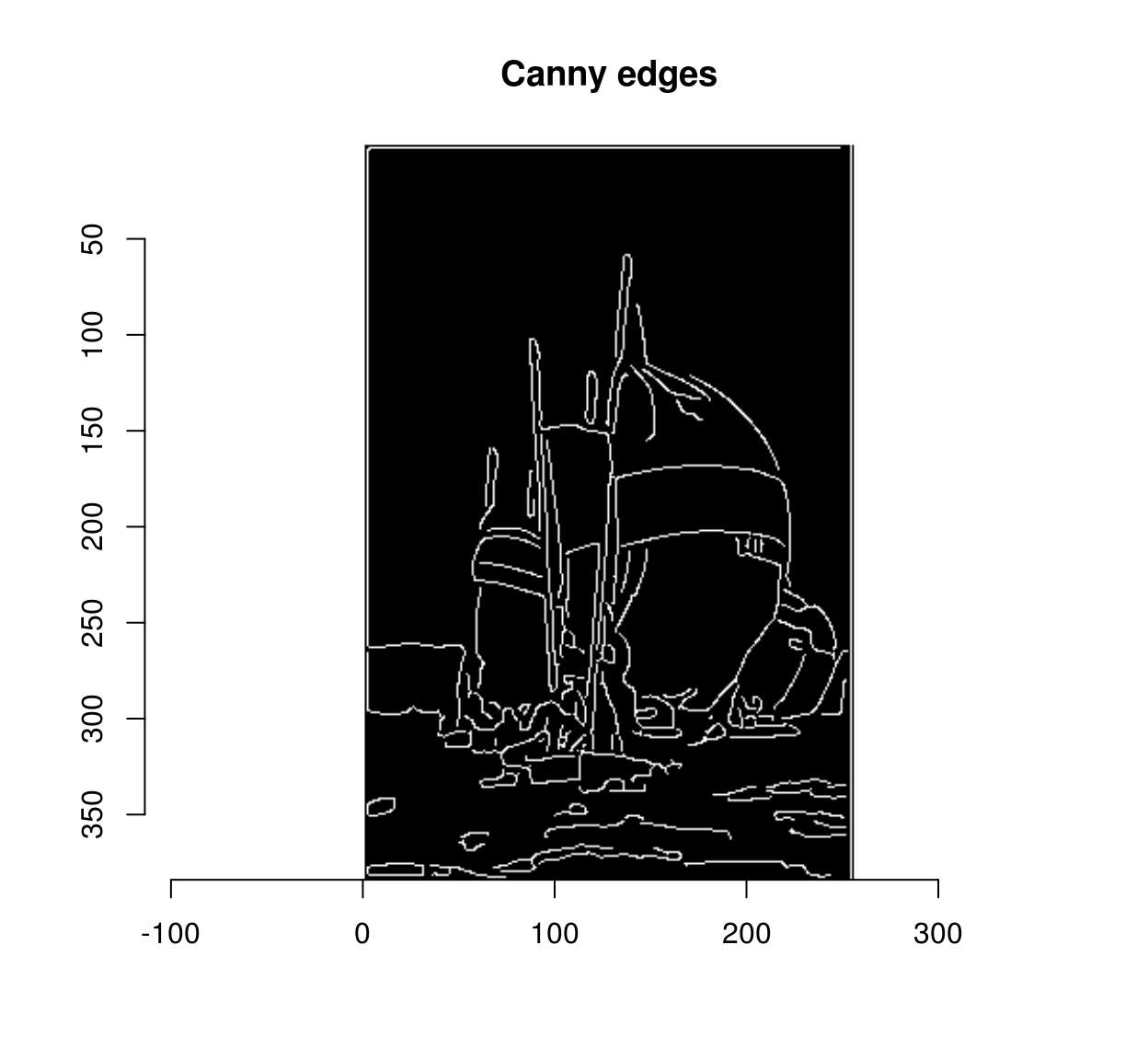
##

## $weak

## Pixel set of size 4244. Width: 256 pix Height: 384 pix Depth: 1 Colour channels: 1

canny <- out$strong

plot(canny,main="Canny edges")



Of course, in a way we are cheating here: the fixed point iteration is really a loop. But instead of looping over each pixel (which is slow, because there are many pixels), we run a global computation on all pixels, 70 times, which is much faster. The fact that it takes 70 iterations to rescue all the weak edges tells us that, in this image at least, some weak edges are hard to reach: there is a connection between the initial set of strong edges and some far-away weak edges. A better computational primitive in this case is the bucket fill (also called the “flood fill”), which you probably know from using your favourite image editor.

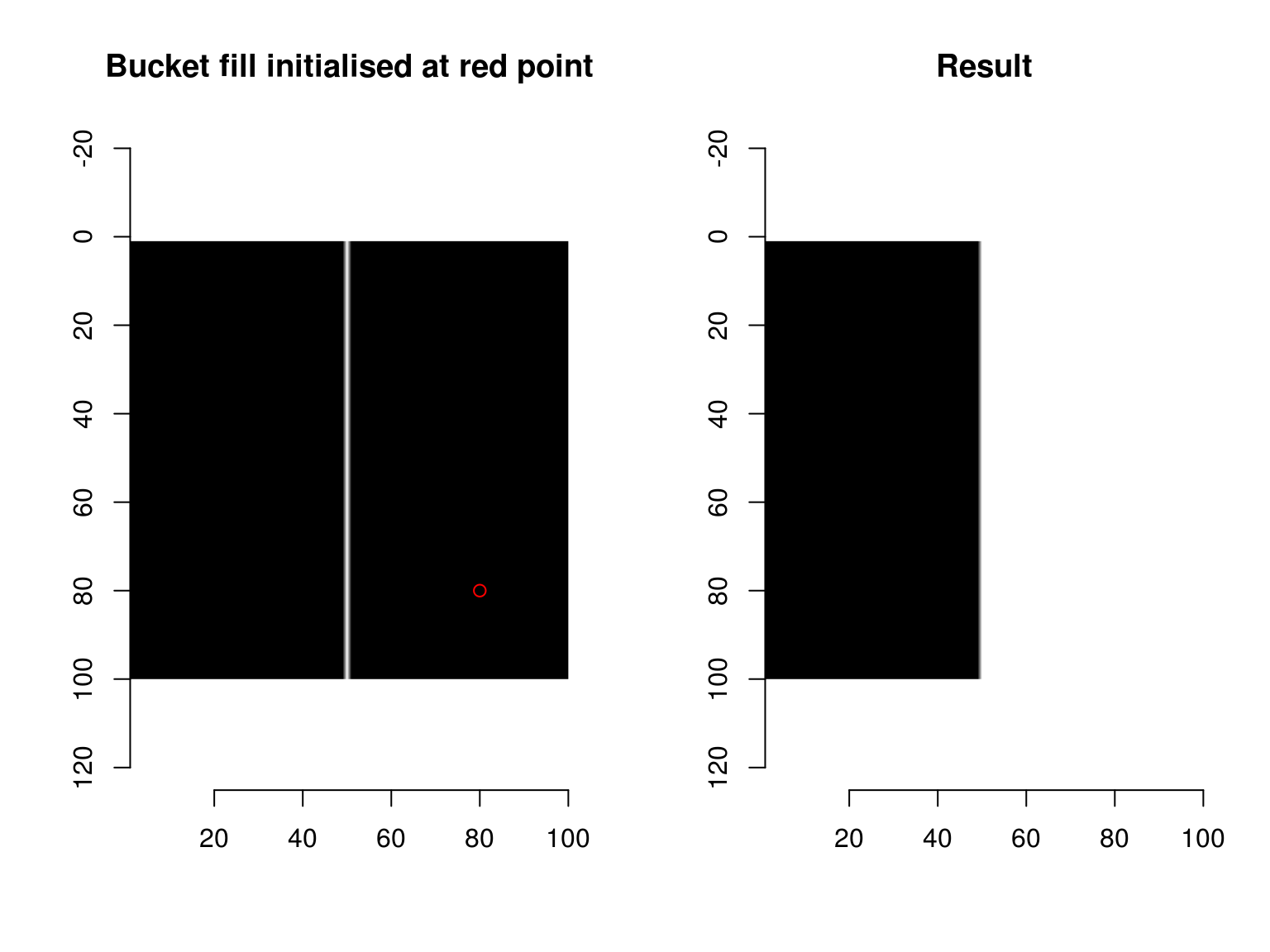
im1 <- as.cimg(**function**(x,y) x == 50,100,100)

layout(t(1:2))

plot(im1,main="Bucket fill initialised at red point")

points(80,80,col="red")

bucketfill(im1,80,80,color=1,high\_connex=TRUE) %>% plot(main="Result")



The bucketfill has a tolerance parameter, which sets a stopping criterion: if nearby pixels are more than “sigma” apart in value, they won’t be filled. The trick will consist in starting a bucket fill inside a strong region, and painting nearby weak pixels as strong. If we do this from all strong regions, we’ll have rescued all the weak edges that need to be rescued. split\_connected splits a pixel set into its connected regions:

pxs <- split\_connected(strong,high\_connectivity=TRUE)

length(pxs)

## [1] 124

Each element in pxs is a pixel set representing a set of connected strong edge pixels.

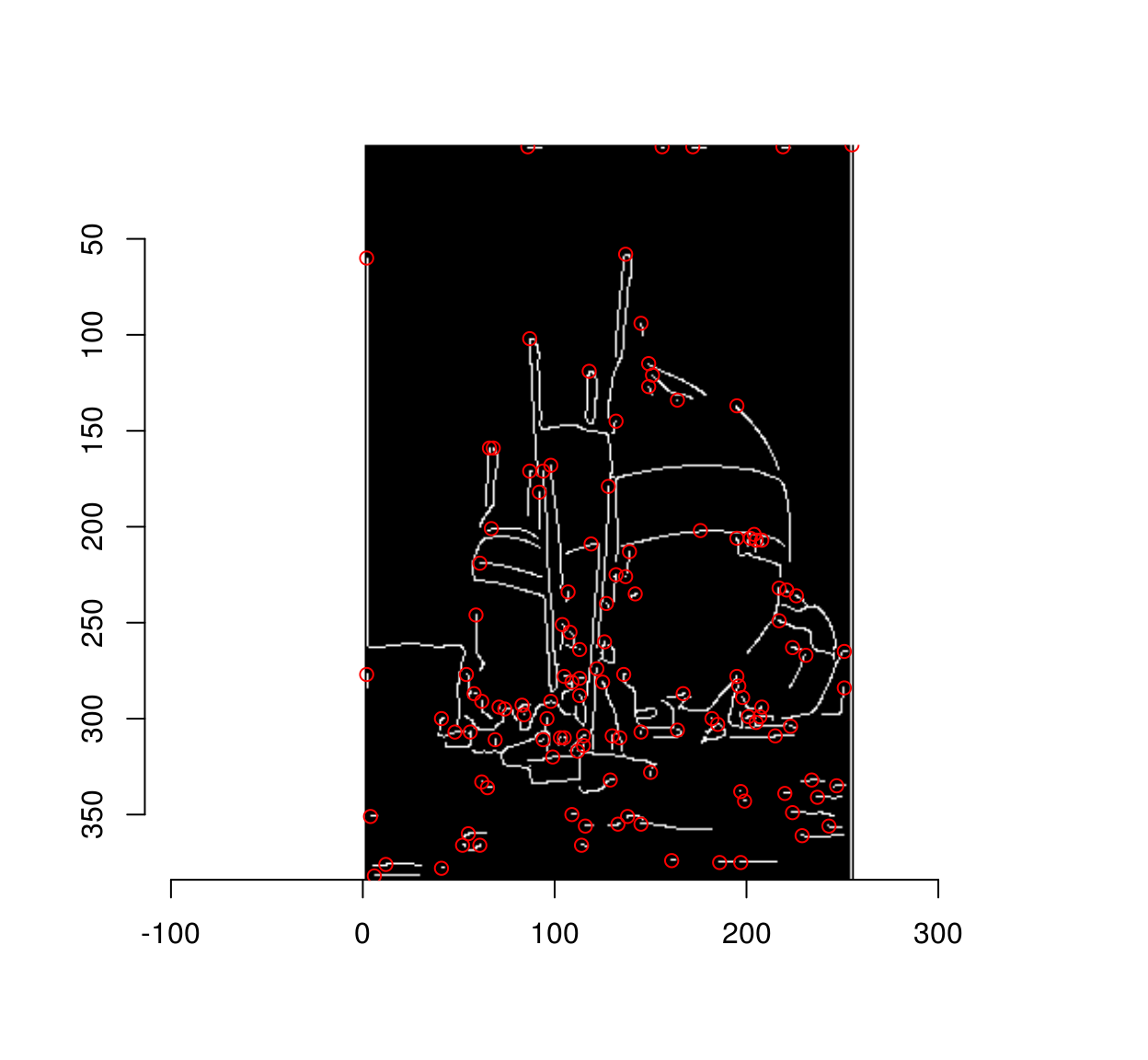
We need seed pixels from each region, which we can collect using purrr’s map function:

*#Collect seed pixels and plot their location*

plot(strong)

**library**(purrr)

map\_df(pxs,~ where(.)[1,]) %$% points(x,y,col="red")



For the bucket fill trick to work we need to be able to spread the value of strong pixels to their weak neighbour (but not elsewhere). We assign different values to strong and weak pixels:

v <- as.cimg(strong)

v[weak==1] <- .9 *#Strong pixels have value 1, weak .9, and the rest are 0.*

Finally we need to go through the list of seed pixels, and apply bucket fill every time. We could write a loop, but to keep with the functional style we have used so far, we’ll cast this operation as a fold, using reduce from the purrr package. reduce takes a function of two arguments (an accumulator and an item), and reduces a list to a single item by accumulating. It’s best illustrated by example:

**library**(purrr)

add <- **function**(l) reduce(l,**function**(acc,item) acc+item,.init=0)

add(1:3) *#equals sum(1:3)*

## [1] 6

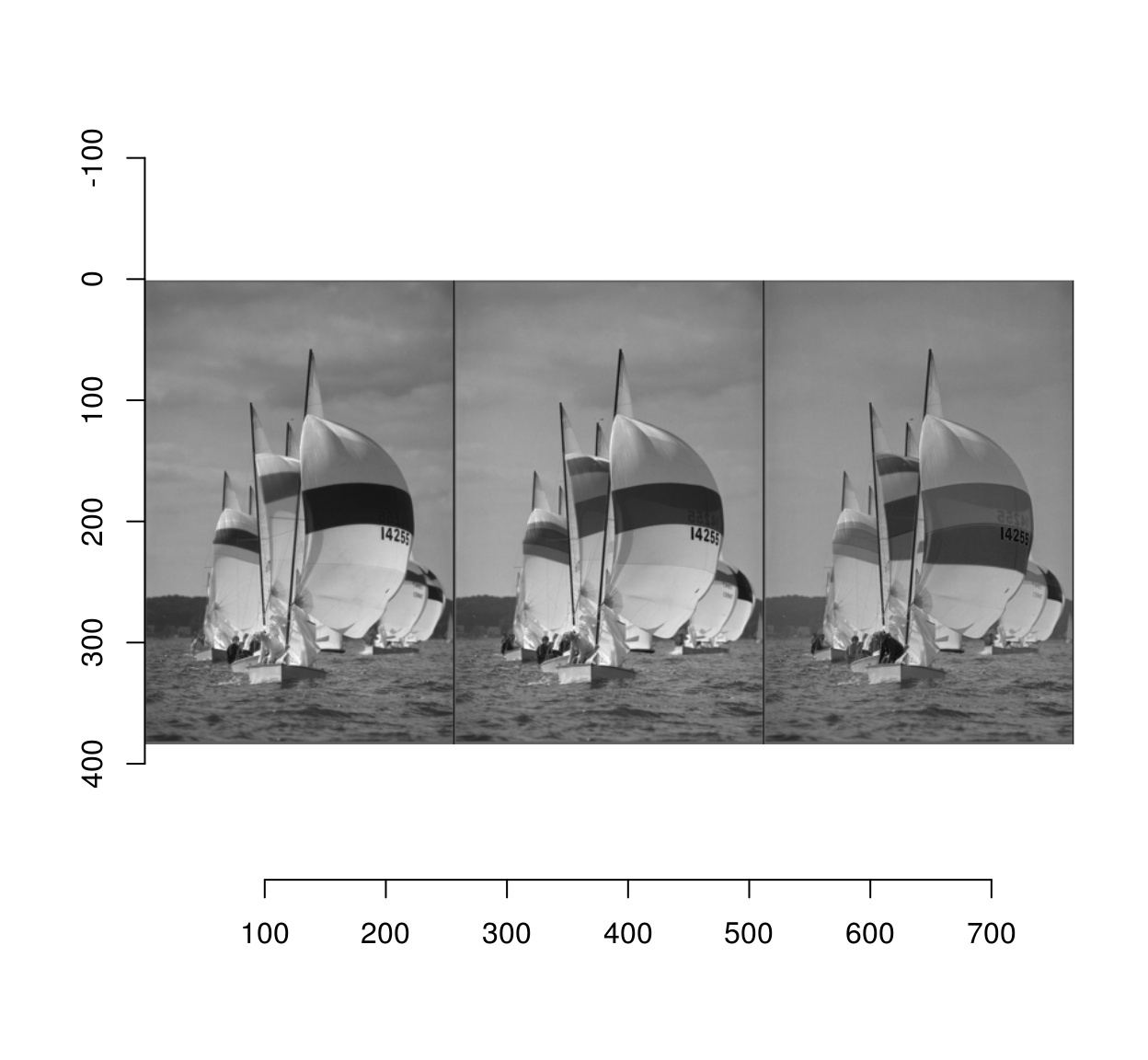
mult <- **function**(l) reduce(l,**function**(acc,item) acc\*item,.init=1)

mult(1:3) *#equals prod(1:3)*

## [1] 6

*#put the three colour channels side-by-side*

imsplit(boats,"c") %>% reduce(**function**(acc,l) imappend(list(acc,l),"x")) %>% plot



*#Gives a set of initial locations for the bucket fill*

fillInit <- **function**(strong)

{

pxs <- split\_connected(strong,high\_connectivity=TRUE)

map\_df(pxs,~ where(.)[1,])

}

*#Starts a fill at each successive location, and accumulates the results*

rescueFill <- **function**(strong,weak)

{

v <- as.cimg(strong)

v[weak] <- .9

loc <- fillInit(strong)

*#Transform the data.frame into a list of locations*

loc <- transpose(loc)

*#Fold*

out <- reduce(loc,**function**(v,l) bucketfill(v,l$x,l$y,color=1,sigma=.1,high=TRUE),

.init=v)

out==1

}

canny2 <- rescueFill(strong,weak)

all.equal(canny,canny2)

## [1] TRUE

The second function is much faster, at least on this image (for this particular choice of thresholds):

system.time(hystFP(list(strong=strong,weak=weak)))

## user system elapsed

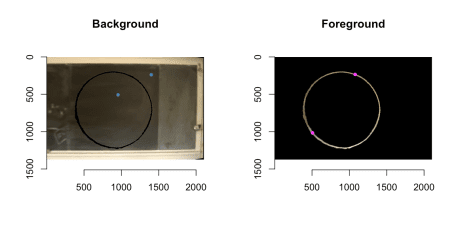
## 0.688 0.012 0.698

system.time(rescueFill(strong,weak))

## user system elapsed

## 0.172 0.004 0.174

To detect the circle from this we specify a few seed points for a [watershed](https://en.wikipedia.org/wiki/Watershed_%28image_processing%29) algorithm with a priority map inverse proportional to gradient magnitude. This includes a few points outside the circle and a few points inside the circle. Note: a perfect circle would have no border, but when drawing a circle with a piece of chalk it’s destined to have a thin border line.

We can now extract the circle by:

##Just the circle

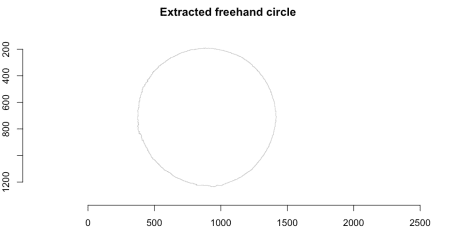
freehandCircle <- (warp \* (mask==2) > 0) %>% grayscale

##Total area covered by the circle

freehandDisc <- label(freehandCircle, high\_connectivity=TRUE) > 0

dilatedDisc <- freehandDisc %>% dilate\_rect(sx=3,sy=3)

freehandCircleThinBorder <- (freehandDisc - dilatedDisc) != 0



**Perfect circle estimation**

Once the freehand circle path in the image has been identified, we need to find the best fitting *perfect* circle matching this path. This problem is elegantly solved by Coope (1993), who formulates the problem as finding center and radius of the circle minimizing the squared Euclidean distance to \(m\) data points \(a\_j\), \(j=1,\ldots,m\). Denoting by \(c\) the center of the circle and by \(r>0\) the radius we want to find the solution of

\[  
\min\_{c\in \mathbb{R^2}, r>0} \sum\_{j=1}^m F\_j(c,r)^2, \quad\text{where}\quad F\_j(c,r) = \left|r – ||c-a\_j||\_2\right|,  
\]

and \(||x||\_2\) denotes Euclidean distance. Because the curve fitting minimizes the distance between an observed point \(a\_j\) and its closest point on the circle and thus involves both the \(x\) and the \(y\) direction , this is a so called [**total least squares**](https://en.wikipedia.org/wiki/Total_least_squares) problem. The problem is non-linear and can only be solved by iterative numerical methods. However, the dimension of the parameter space can be reduced by one, because given the center \(c\) we can determine that \(r(c)=\frac{1}{m} \sum\_{j=1}^m ||c-a\_j||\_2\).

##Compute radius given center

radius\_given\_center <- function(center, dist=NULL) {

if (is.null(dist)) {

a <- as.matrix(where(freehandCircleThinBorder > 0))

dist <- sqrt((a[,1] - center[1])^2 + (a[,2] - center[2])^2)

}

return(mean(dist))

}

##Target functin of the total least squares criterion of Coope (1993)

target\_tls <- function(theta) {

##Extract parameters

center <- exp(theta[1:2])

##Total least squares criterion from Coope (1993)

a <- as.matrix(where(freehandCircleThinBorder > 0))

dist <- sqrt((a[,1] - center[1])^2 + (a[,2] - center[2])^2)

##Compute radius given center

radius <- radius\_given\_center(center, dist)

F <- abs( radius - dist)

sum(F^2)

}

res\_tls <- optim(par=log(c(x=background[1,1], y=background[1,2])), fn=target\_tls)

center <- exp(res\_tls$par)

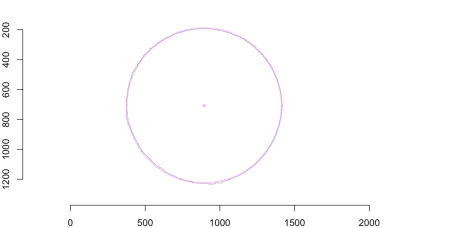
fit\_tls <- c(center,radius=radius\_given\_center(center))

fit\_tls

## x y radius

## 894.3885 707.3191 518.4119

We illustrate the freehand circle (in black) and the fitted circle (magenta) on top of each other using the alpha channel. You have to study the image carefully to detect differences between the two curves!



**Quantifying the circularness of the freehand circle**

We quantify the **circularness** of the freehand circle by contrasting the area covered by it with the area of the fitted perfect circle. The closer this ratio is to 1 the more perfect is the freehand circle.

##Area of the freehand drawn disc

areaFreehandDisc <- sum(freehandDisc)

##Area of the disc corresponding to the idealized circle fitted

##to the freehand circle

areaIdealDisc <- pi \* fit\_tls["radius"]^2

##Ratio between the two areas

ratio\_area <- as.numeric(areaFreehandDisc / areaIdealDisc)

ratio\_area

## [1] 0.9971778

Yup, it’s a pretty perfect circle! Note also that the calibration is to a circle with an area of 1 pixel unit and not a circle with diameter of 1m as described in the above text about the "circleometer". Since the fitted circle already takes the desired shape into account, my intuition is that this ratio is a pretty good way to quantify circularness. However, to avoid **measurehacks**, we use as backup measure the circleometer approach: for each point on the freehand circle we measure its distance to the freehand circle and integrate/sum this up over the path of the freehand circle. We can approximate this integration using image pixels as follows.

##Create a pixel based circle in an image of the same size as the

##freehandCircle img. For visibility we use a border of 'border' pixels

##s.t. circle goes [radius - border/2, radius + border/2].

Circle <- function(center, radius, border) {

as.cimg(function(x,y) {

lhs <- (x-center[1])^2 + (y-center[2])^2

return( (lhs >= (radius-border/2)^2) & (lhs <= (radius+border/2)^2))

}, dim=dim(freehandCircle))

}

##Build pixel circle based on the fitted parameters

C\_tls <- Circle(fit\_tls[1:2], fit\_tls[3], border=1)

##Calculate Euclidean distance to circle for each pixel in the image

dist <- distance\_transform(C\_tls, value=1, metric=2)

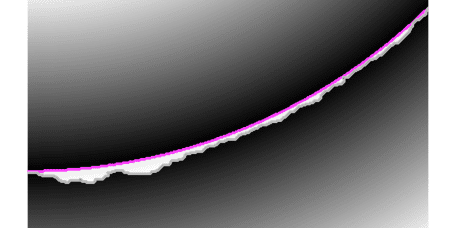
##Distance between outer border of freehand circle and perfect circle

area\_difference <- sum(dist[freehandCircleThinBorder>0])

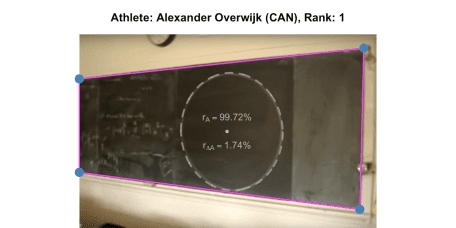
##Compute area difference and scaled it by the area of the fitted disc

ratio\_areadifference <- as.numeric(area\_difference / areaIdealDisc)

The image below illustrates this by overlaying the result on top of the distance map. For better visualization we zoom in on the 270-300 degree part of the circle (i.e. the bottom right). In magenta is the fitted perfect circle, in gray the freehand circle and the area between the two paths is summed up over the entire path of the freehand circle:



We obtain ratio\_areadifference= 0.01735. Thus also this measure tells us: it’s a pretty perfect circle! To summarise: The output on the display of the judge’s Circle-O-Meter App (available under a GPL v3 license) at the World Freehand Circle Drawing Championship would be as follows: 



**Discussion**

We took elements of computer vision, image analysis and total least squares to segment a chalk-drawn circle on a blackboard and provided measures of it’s circularness. Since we did not have direct access to the measurements of the blackboard in object space, a little guesstimation was necessary, nevertheless, the results show that it was a pretty circular freehand circle!

With the machinery in place for judging freehand circles, its time to send out the call for contributions to the **2nd World Freehand Circle Drawing Championship** (online edition!). Stay tuned for the call: participants would upload their photo plus minor modifications of a general analysis R-script computing the two area ratios measures and submit their contribution. You can spend the anxious waiting time practicing your freehand 1m diameter circles – it’s a good way to loosen up long & unproductive meetings!

Leaderboard.R

|  |
| --- |
| library(purrr) |
|  | library(dplyr) |
|  |  |
|  | ##Directory containing the image and .csv files with the seed points |
|  | path <- file.path("round-1") |
|  | img\_list <- list.files( path=path, pattern="\*.jpg") |
|  |  |
|  | ##Common scale factor for scoring all images |
|  | scaleFactor <- 50 |
|  |  |
|  |  |
|  | ##Compute the circularity scores for the entire batch of jpg files located in a certain directory. |
|  | leaderboard <- map\_df(img\_list, ~ { |
|  | cat("Processing: ", .x, "\n") |
|  | #Load image and coordinates |
|  | file\_path <- file.path(file\_path, .x) |
|  | img <- imager::load.image(file\_path) |
|  | seedPoints <- read.csv(file=gsub("\\.jpg$","\\.csv", file\_path)) |
|  | names(seedPoints) <- NULL |
|  |  |
|  | ##Scale |
|  | img <- imager::resize(img, -scaleFactor, -scaleFactor) |
|  | seedPoints[,1:2] <- as.matrix(seedPoints[,1:2]) \* scaleFactor/100 |
|  |  |
|  | ##Measure |
|  | res <- perfectcircle::circularity(warp=img, seedPoints=seedPoints, progress=NULL) |
|  | res$fileName <- gsub("\\.csv","", .x) |
|  |  |
|  | res |
|  | }) |
|  |  |
|  | #Leaderboard |
|  | leaderboard %>% arrange(desc(score)) %>% select(score, fileName, ratio\_area) |

**Appendix**

If we instead of the total sum of squares criterion involving \(F\_j(c,r)\) mentioned in the text solve the related criterion \[  
\sum\_{j=1}^m f\_j(c,r)^2, \quad\text{where}\quad f\_j(c,r) =  
||c-a\_j||\_2^2 – r^2,  
\] then a much simpler solution emerges. Coope (1993) explains that this alternative criterion geometrically corresponds to minimizing the product

\[  
\text{(distance to the closest point on the circle)}\times \text{(distance to  
the furthest away point point on the circle)}  
\]

over the measurement point. In order to obtain the solution write the residuals \(f\_j\) as \(f\_j(c,r) = c^T c – 2 c^T a\_j + a\_j^T a\_j – r^2\) and perform a change of variables from \((c\_1, c\_2, r)'\) to \[  
y =  
\left[  
\begin{matrix}  
2 c\_1 \\  
2 c\_2 \\  
r^2 – c^T c \\  
\end{matrix}  
\right]  
\quad \text{and let} \quad  
b\_j =  
\left[  
\begin{matrix}  
a\_{j1} \\  
a\_{j2} \\  
1  
\end{matrix}  
\right].  
\] The minimization problem then becomes \[  
\min\_{y \in \mathbb{R}^3} \sum\_{j=1}^m \left\{ a\_j^T a\_j – b\_j^T y \right\},  
\] which can be written as a linear least square (LLS) expression \[  
\min\_y ||By – d||\_2^2,  
\] where \(B\) is a \(3\times m\) matrix with the \(b\_j\)-vectors as columns and \(d=||a\_j||\_2^2\). This expression is then easily solved using the standard least squares machinery.

##Fast linear least squares problem as described in Coope (1993)

fitCircle\_lls <- function(freehandCircle) {

a <- as.matrix(where(freehandCircle > 0))

b <- cbind(a,1)

B <- b

d <- a[,1]^2 + a[,2]^2

y <- solve(t(B) %\*% B) %\*% t(B) %\*% d

x <- 1/2\*y[1:2]

r <- as.numeric(sqrt(y[3] + t(x) %\*% x))

return(c(x=x[1], y=x[2], radius=r))

}

##Fit using linear least squares procedure of Coole (1993)

fit\_lls <- fitCircle\_lls(freehandCircleThinBorder)

##Compare TLS and LLS fit

rbind(lls=fit\_lls,tls=fit\_tls)

## x y radius

## lls 894.3666 707.3901 518.4295

## tls 894.3885 707.3191 518.4119

In other words: the results are nearly identical.