ACE vignette

Jos B. Poell

2018-05-23

Contents

1 Introduction

1.1 Why use ACE?

You want to know the percentage of tumor cells in your sample(s) and you have (preferably whole-genome) NGS data. You want pretty copy number profiles of your samples. You want to know how many copies are present of a certain chromosomal segment, or even gene, or mutation!

1.2 How ACE works

ACE is a an absolute copy number estimator that scales copy number data to fit with integer copy numbers. For this it uses segmented data from the QDNAseq package, which in turn uses a number of dependencies. Note: make sure

QDNAseq fetches the bin annotations from the same genome build as the one used for aligning the sequencing data! On with ACE! In brief: ACE will run QDNAseq or use its output rds-file(s) of segmented data. It will subsequently run through all samples in the object(s), for which it will create individual subdirectories. For each sample, it will calculate how well the segments fit (the relative error) to integer copy numbers for each percentage of "tumor cells" (cells with divergent segments). Note that it does not estimate for a lower percentage than 5. ACE will output a graph with relative errors (all errors relative to the largest error). Said graph can be used to quickly identify the most likely fit. ACE selects all "minima" and saves the corresponding copy number plots. The "best fit" (lowest error) is not by definition the most likely fit! ACE will run models for a general tumor ploidy of 2N, but you can expand this to include any ploidy of your choosing. The program needs to make one assumption: the median bin segment value corresponds with the tumor's general ploidy. If the median bin segment value of a sample (the "standard") lies on a segment that happens to be 3N, but you only ran ACE on 2N, you can run that sample individually using the singlemodel function with argument ploidy = 3 (see section "examining single samples"). If the standard happens to be on a subclonal segment, you either have to manually change the standard, or use the squaremodel function (again, see below). The output of ACE is designed in such a way that it is "easy" to quickly analyze multiple samples. Bear in mind that it is absolutely necessary to manually select the most likely models! I have made a conscious decision not to let ACE "autopick" the best model. See below for a manual how to most efficiently pick the most likely models from your output. Let's get started!

2 Running ACE

2.1 Getting started

The ACE package includes segmented data derived from low-coverage whole genome sequencing, which will be used throughout this vignette. The mapped sequencing data has been processed through the QDNAseq package. Users of ACE, however, are most likely to start from their own bam-files, not the pre-processed segmented data. runACE, the core functionality of ACE, will run a default set of QDNAseq functions if bam-files are provided as the data source (make sure the genome builds correspond, I cannot stress this enough). If you wish to run the code below, make sure the file paths are correct. To get started, I recommend using a directory that only contains a few bam-files. The function runACE is designed to automatically analyze all samples in a directory. Input should be either segmented QDNAseq-objects or aligned bam-files. For details on all arguments, consult the runACE documentation. Let's get started!

```
userpath <- "D:/DATA/bam-files"
# if you do not want the output in the same directory, use the argument outputdir
runACE(userpath, filetype='bam', binsizes = c(100, 1000), ploidies = c(2,4), imagetype='png')
```

If you do not have aligned bam-files ready to go, you can use the data provided in the package:

```
data("copyNumbersSegmented")
userpath <- "D:/DATA/ACE"
saveRDS(file.path(userpath,"copyNumbersSegmented.rds"))
runACE(userpath, filetype='rds', ploidies = c(2,4), imagetype='png')
```

2.2 ACE output

2.2.1 rds-file

This is the segmented QDNAseq object; obviously not created when using rds-file as input. It can be used if you want to run ACE again with slightly different parameters. More importantly, you can use this file to examine individual samples in downstream analyses.

2.2.2 rds subdirectories

ACE creates a subdirectory for each rds-file. In case of bam-files as input, the subdirectories have the names of the binsizes.

2.2.3 ploidies subdirectories

For each analyzed tumor ploidy, ACE makes a subdirectory. In this case: 2N and 4N

2.2.4 summaries

summary_errors: error lists of all the models summary_likelyfits: copy number plots of the best fit and the last minimum of each sample, with the corresponding error list plots. I would recommend using the likelyfits for model selection. Summary files can become quite big / huge depending on sample size and bin size. See below how to deal with this.

2.2.5 likelyfits subdirectory

This subdirectory contains the individual copy number graphs of the likelyfits.

2.2.6 individual sample subdirectories

These subdirectories have a summary file with all the fits for the corresponding sample and the error list plot. Individual copy number graphs are available in the subdirectory "graphs".

2.2.7 fitpicker tables

This tab-delimited file can be used during selection of most likely models. Especially handy when analyzing a large number of samples. More instructions below.

2.3 Model selection

Having this massive pile of output can be daunting, how do you make sense of it all? First of all, if you have multiple bin sizes, you generally only need a single bin size for your model selection. I recommend using a relatively large bin size. File sizes are smaller and segmentation is often more robust. You can probably find the corresponding model in the smaller binsizes as well, if you prefer to use those for copy number graphs. Cellularities between corresponding models made with different bin sizes are usually very similar, but some fits may be "missed". The most likely fit of a tumor is generally 1) the fit with the smallest error, or 2) the fit at the highest cellularity, so those two are presented in the summary_likelyfits file. When you open this file, you see for each sample three plots in a row (in case of imagetype = 'pdf', it will make a new page from each plot). The first two of the three are copy number plots of the best fit (lowest relative error) and the last minimum, respectively. The third plot is the error list, which shows the relative error of the fit at each tested "cellularity" (tumor cell percentage). In most cases you will be able to pick a good fit from these graphs. The individual graph is then available in the likelyfits subdirectory. If none of the models fit well, or you think there might be a better fit possible, you can look at all fits in the summary file, available in the subdirectory of that sample. If there are still no good fits (e.g. the tumor has a ploidy of 3N and you did not include this ploidy), you have to do model fitting on this sample separately (see below).

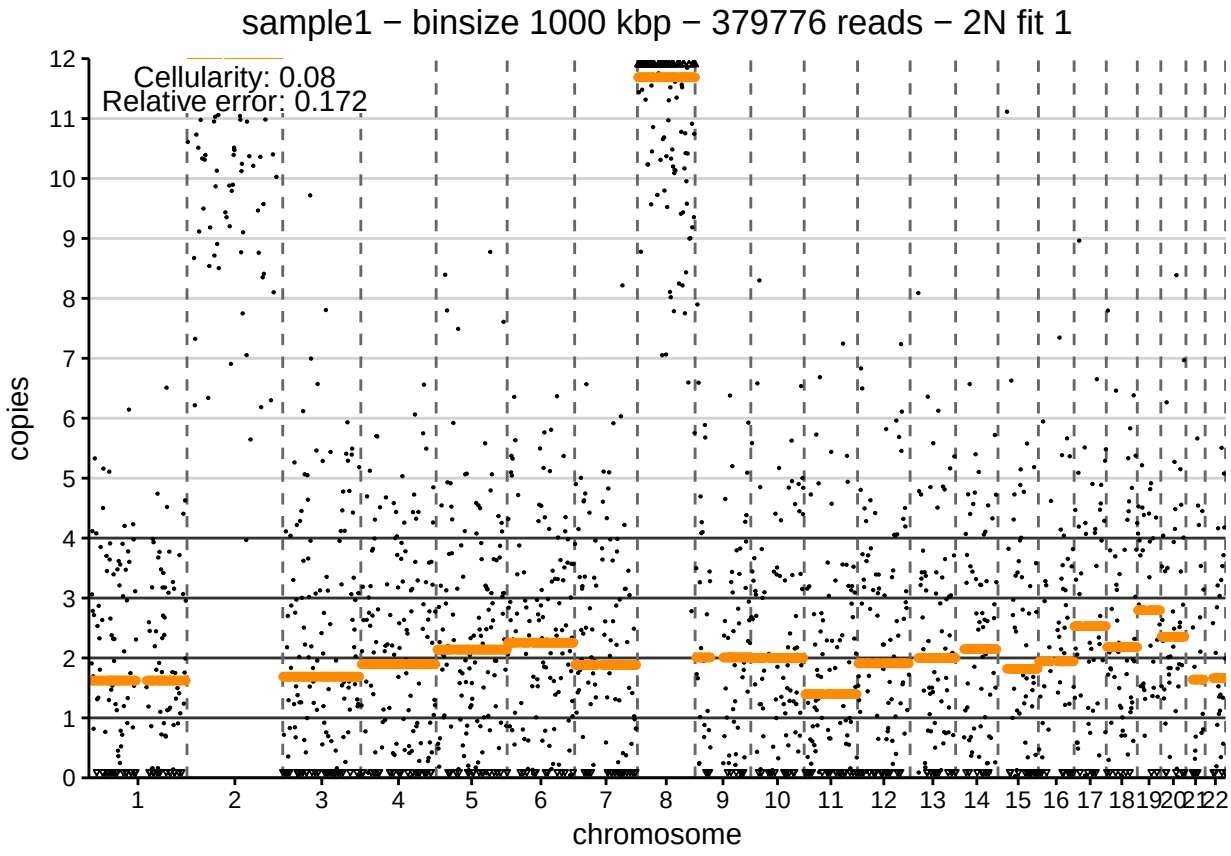
You might want to go through your samples a bit more systematically, especially if you have many samples. You can open the fitpicker.tsv file, go through the fits of the summary file and note in the likely_fit column of the picker table which fit you chose. Just leave open any samples of which you are not sure. The handy thing about the fitpicker files is that they have the list of your sample names and they have the list of cellularities.

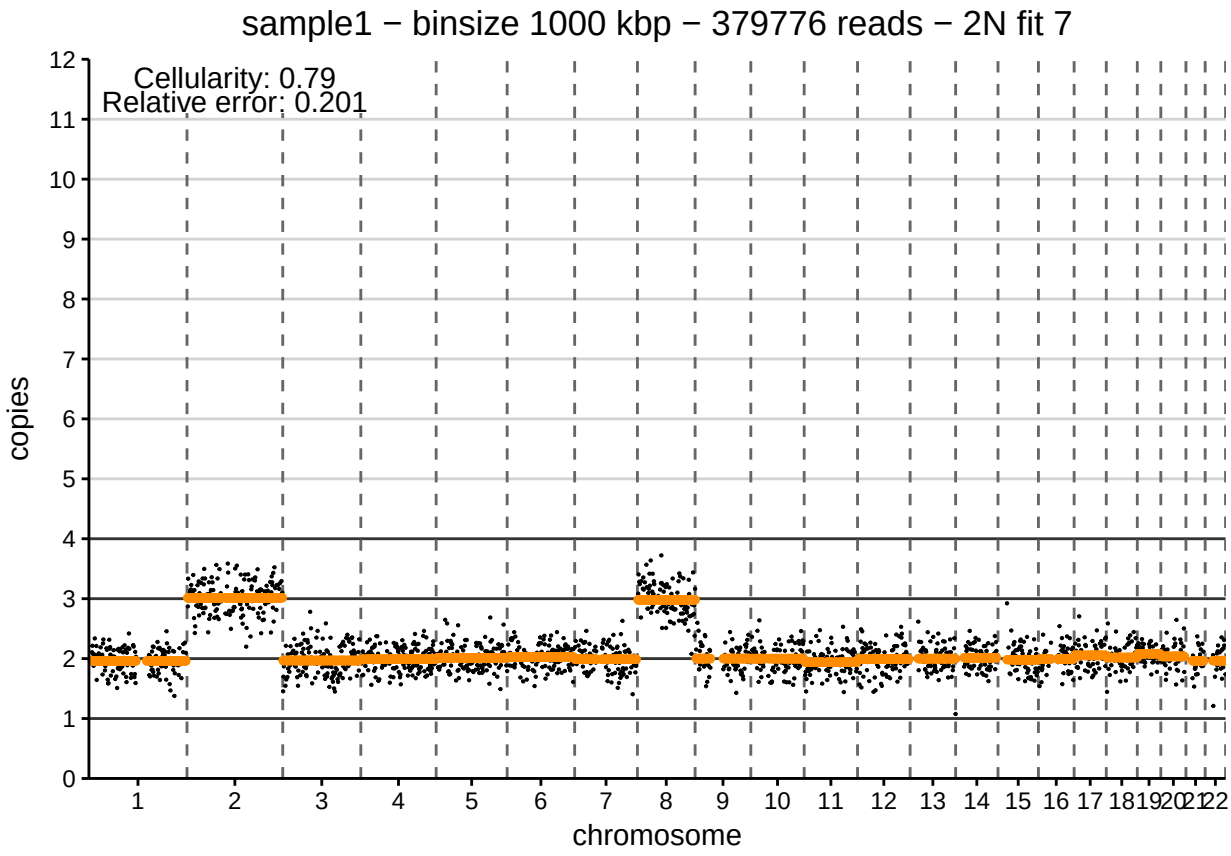
Another feature that might help going through a lot of samples, is using the "penalty" parameter in the runACE call. This parameter penalizes fits at lower cellularities. Doing so greatly improves the chance that the best fit is also the most likely fit, but comes at the cost of precision at high cellularities and comes at the cost of sensitivity at low cellularities (but only at the lowest end of the spectrum, let's say below 10%). More about this later.

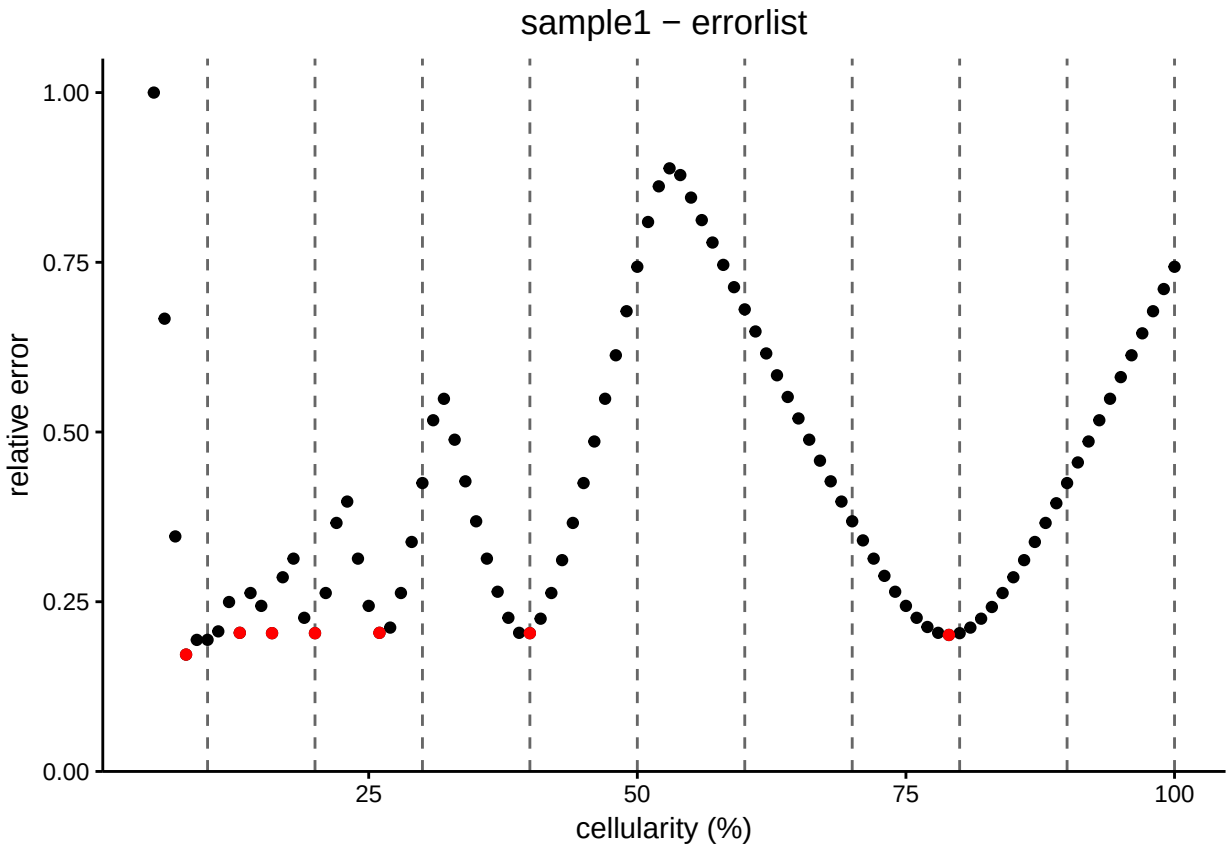
In the two examples given below, it is quite obvious what the most likely appropriate models are: for sample 1 it is the "lastminimum" with a general ploidy of 2N and cellularity of 0.79, and for sample 2 it is the "bestfit" with a general ploidy of 2N and cellularity of 0.38. Examining the error plots of these samples draws a great picture of how obvious the right model can be for the trained eye, while it can be deceivingly tricky to rely on a simple computer algorithm to make the pick.

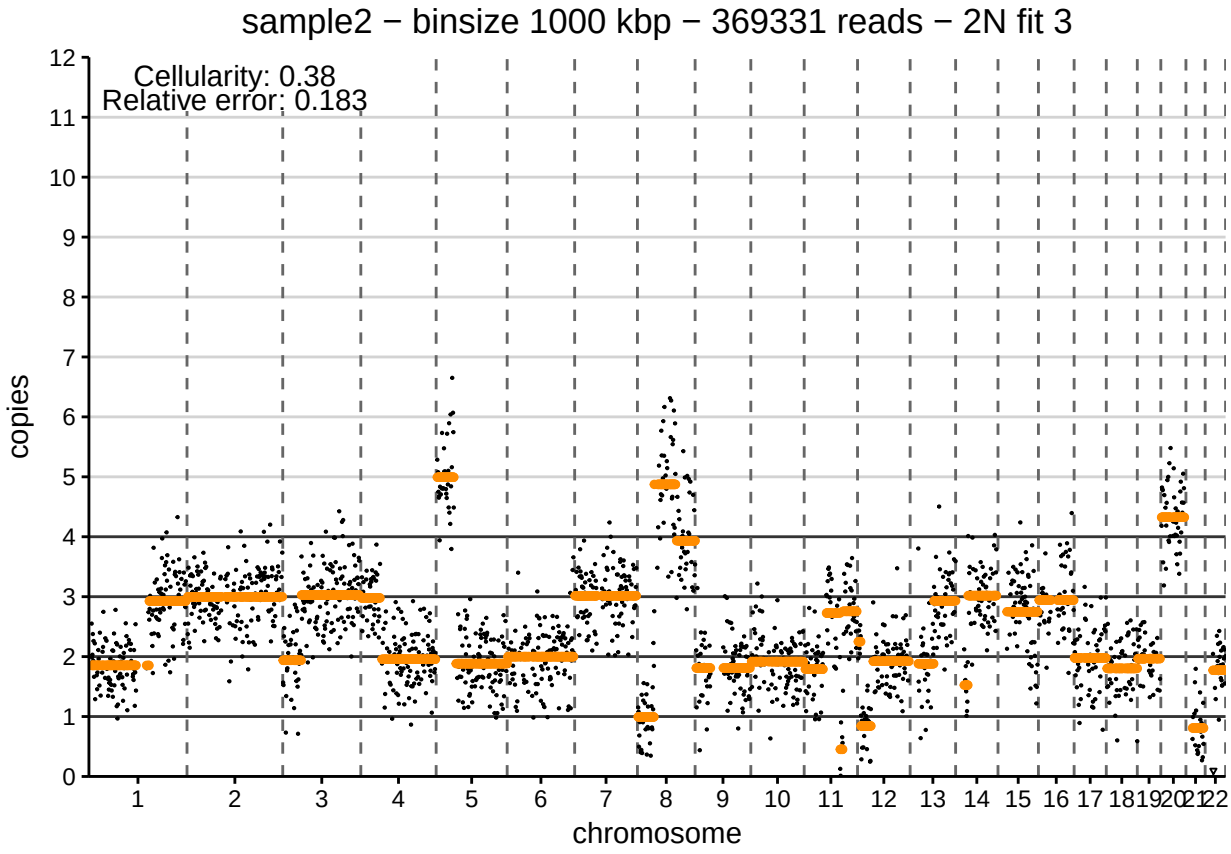
To reproduce some of the output created by runACE (which is only written to file), I have included the following code. The functions used will be explained in more detail later:

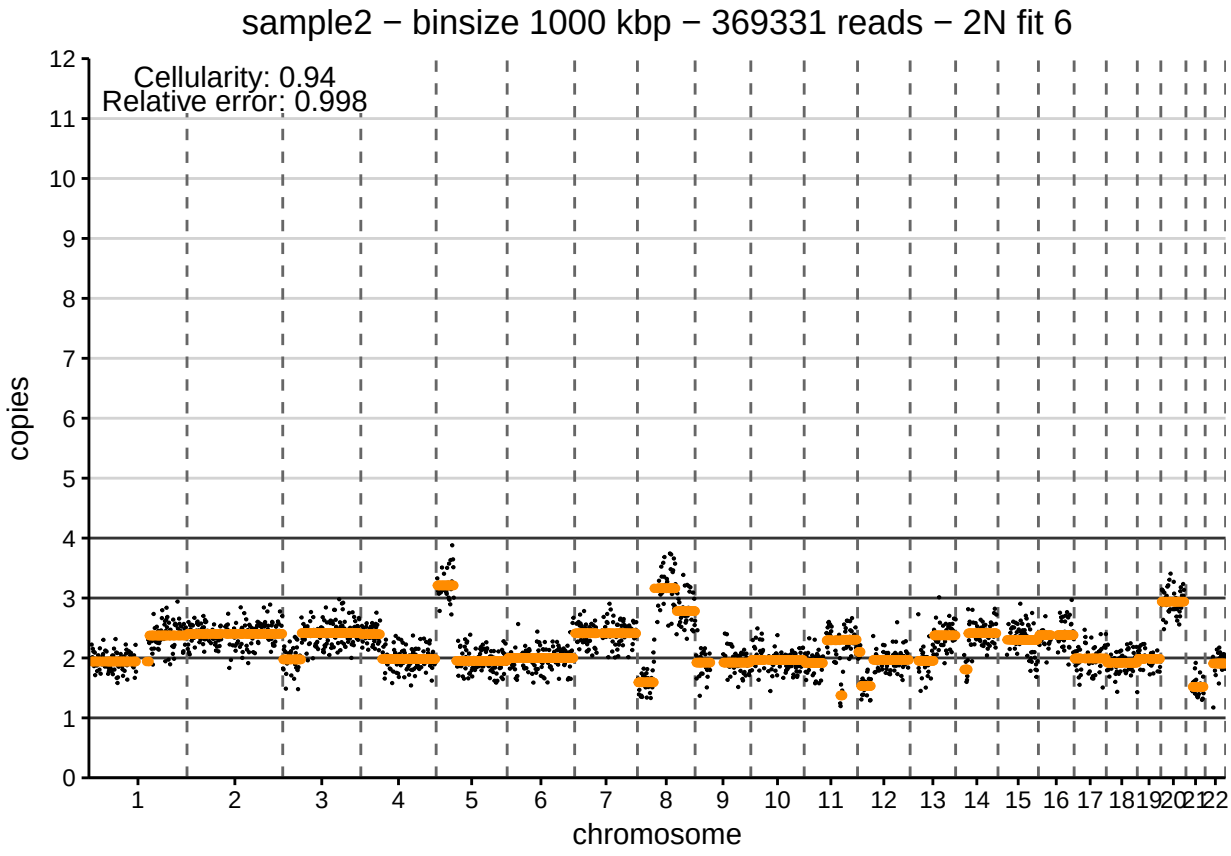
```
data("copyNumbersSegmented")
object <- copyNumbersSegmented
model1 <- singlemodel(object, QDNAseqobjectsample = 1)
bestfit1 <- model1$minima[tail(which(model1$rerror==min(model1$rerror)), 1)]
besterror1 <- min(model1$rerror)
lastfit1 <- tail(model1$minima, 1)
lasterror1 <- tail(model1$rerror, 1)
plot1 <- singleplot(object, QDNAseqobjectsample = 1, cellularity = bestfit1,
                    error = besterror1, standard = model1$standard,
                    title = "sample1 - binsize 1000 kbp - 379776 reads - 2N fit 1")
plot2 <- singleplot(object, QDNAseqobjectsample = 1, cellularity = lastfit1,
                    error = lasterror1, standard = model1$standard,
                    title = "sample1 - binsize 1000 kbp - 379776 reads - 2N fit 7")
plot3 <- model1$errorplot + ggtitle("sample1 - errorlist") +
  theme(plot.title = element_text(hjust = 0.5))
model2 <- singlemodel(object, QDNAseqobjectsample = 2)
plot4 <- singleplot(object, QDNAseqobjectsample = 2, 0.38, 0.183,
                    title = "sample2 - binsize 1000 kbp - 369331 reads - 2N fit 3")
plot5 <- singleplot(object, QDNAseqobjectsample = 2, 0.94, 0.998,
                    title = "sample2 - binsize 1000 kbp - 369331 reads - 2N fit 6")
plot6 <- model2$errorplot + ggtitle("sample2 - errorlist") +
  theme(plot.title = element_text(hjust = 0.5))
plot1
plot2
plot3
plot4
plot5
plot6
```

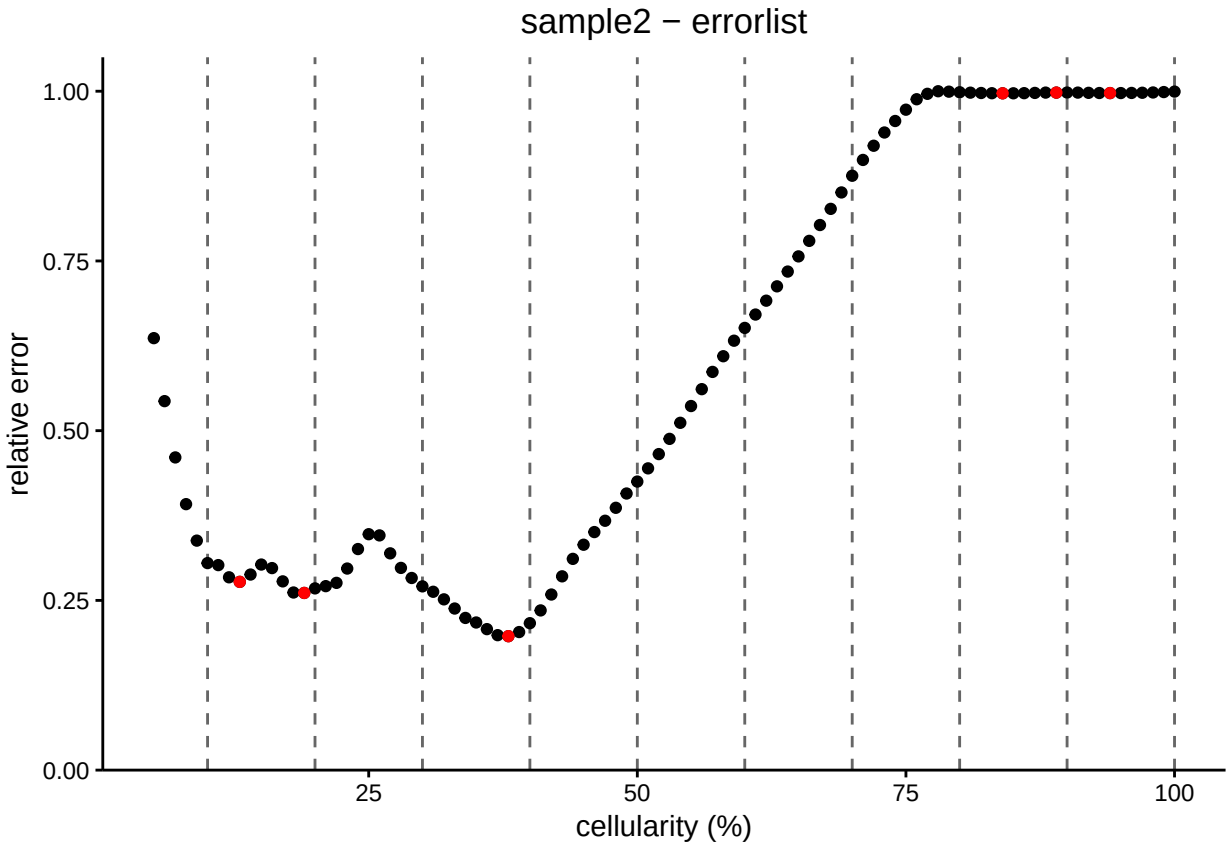

sample1 − binsize 1000 kbp − 379776 reads − 2N fit 1
Cellularity: 0.08
Relative error: 0.172
copies
chromosome


sample1 − binsize 1000 kbp − 379776 reads − 2N fit 7
Cellularity: 0.79
Relative error: 0.201
copies
chromosome


sample1 − errorlist
relative error
1.00
0.75
0.50
0.25
0.00
25
50
75
100
cellularity (%)


sample2 − binsize 1000 kbp − 369331 reads − 2N fit 3
Cellularity: 0.38
Relative error: 0.183
copies
chromosome


sample2 − binsize 1000 kbp − 369331 reads − 2N fit 6
Cellularity: 0.94
Relative error: 0.998
copies
chromosome


sample2 − errorlist
relative error
cellularity (%)

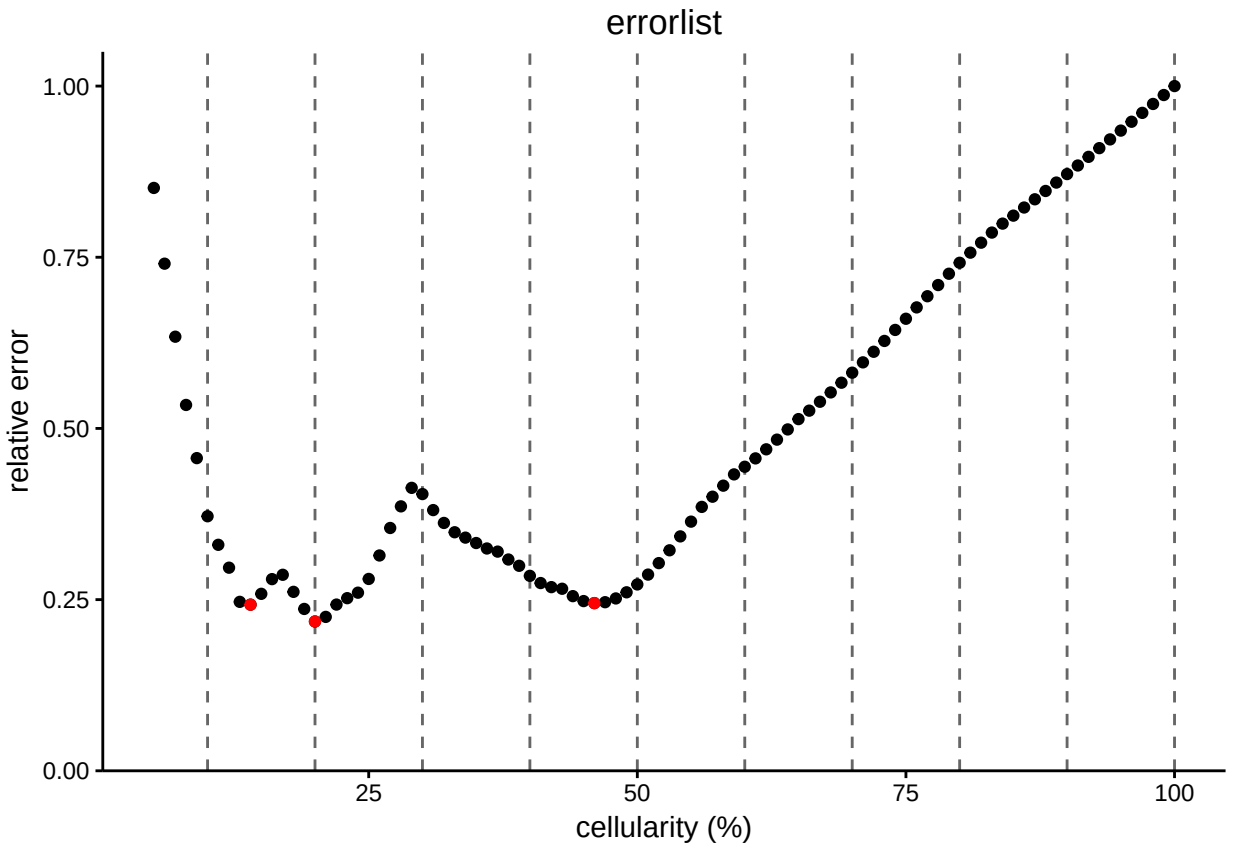
So this should allow you to find the "easy" fits, what about those tricky ones?

2.4 Examining single samples

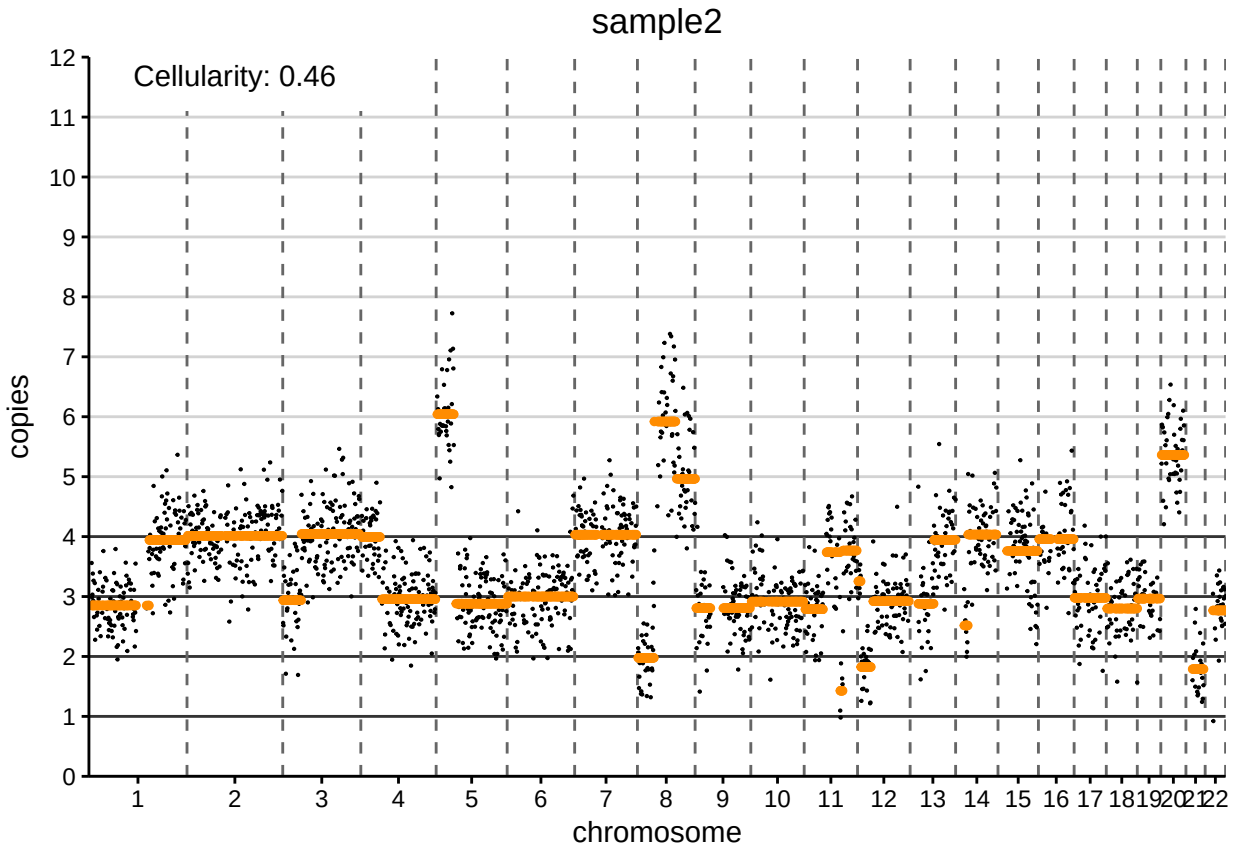
There are several reasons why you might want to zoom in on a single sample. Like mentioned before, it is necessary when the automatically generated fits are no good. Also, you might just be interested in a single sample from your object. Or perhaps you don't want to create all the output, but just see the plots on your graphics device or put the segment data in a dataframe. Here is what to do!

The primary tools for examining single samples are the functions singlemodel and singleplot. Let's say you don't like the fit of sample2, because you're superconvinced that it is mainly a 3N tumor.

```
data("copyNumbersSegmented")
object <- copyNumbersSegmented
# Since you're convinced the mode is 3N, you can run the singlemodel function to
# fit at ploidy = 3
model <- singlemodel(object, QDNAseqobjectsample = 2, ploidy = 3)
ls(model)
## [1] "errorlist" "errorplot" "method"    "minima"    "penalty"   "ploidy"
## [7] "rerror"    "standard"
model$minima
## [1] 0.14 0.20 0.46
model$rerror
## [1] 0.2426503 0.2180412 0.2449059
model$errorplot
```



```
# Examining the errors can give you a feel for the fits. Experience tells
# us the last fit is probably the right one, so let's check out the copy number
# plot. Specify the same parameters, now including the cellularity derived from
# the model.
singleplot(object, QDNAseqobjectsample = 2, cellularity = 0.46, ploidy = 3)
```

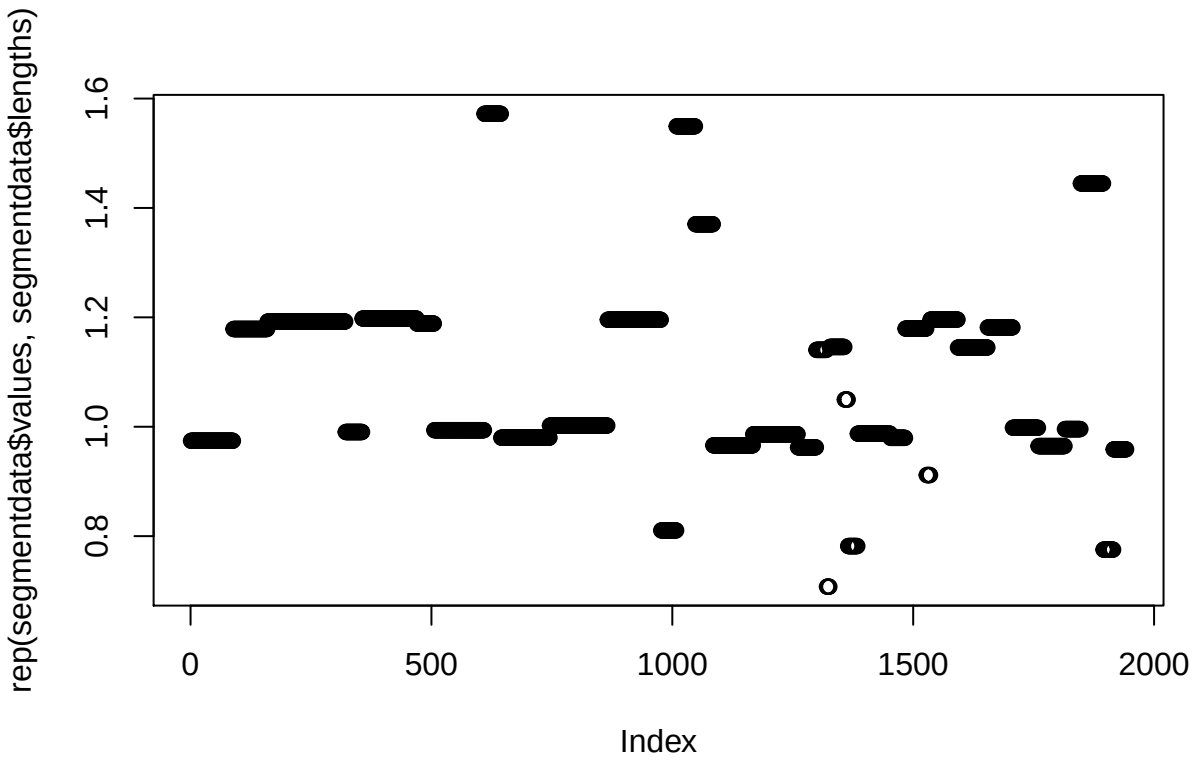



```
# That is actually a very nice fit, let's run with it!
# You can now save the plot however you like.
```

Model fitting as performed by singlemodel, but also runACE, starts with setting the median segment value of a sample to the specified integer ploidy (default = 2). This usually works very well, but in some cases the median segment value may lie on a subclonal segment. Using the argument "standard" you can specify the segment value that should correspond with the ploidy.

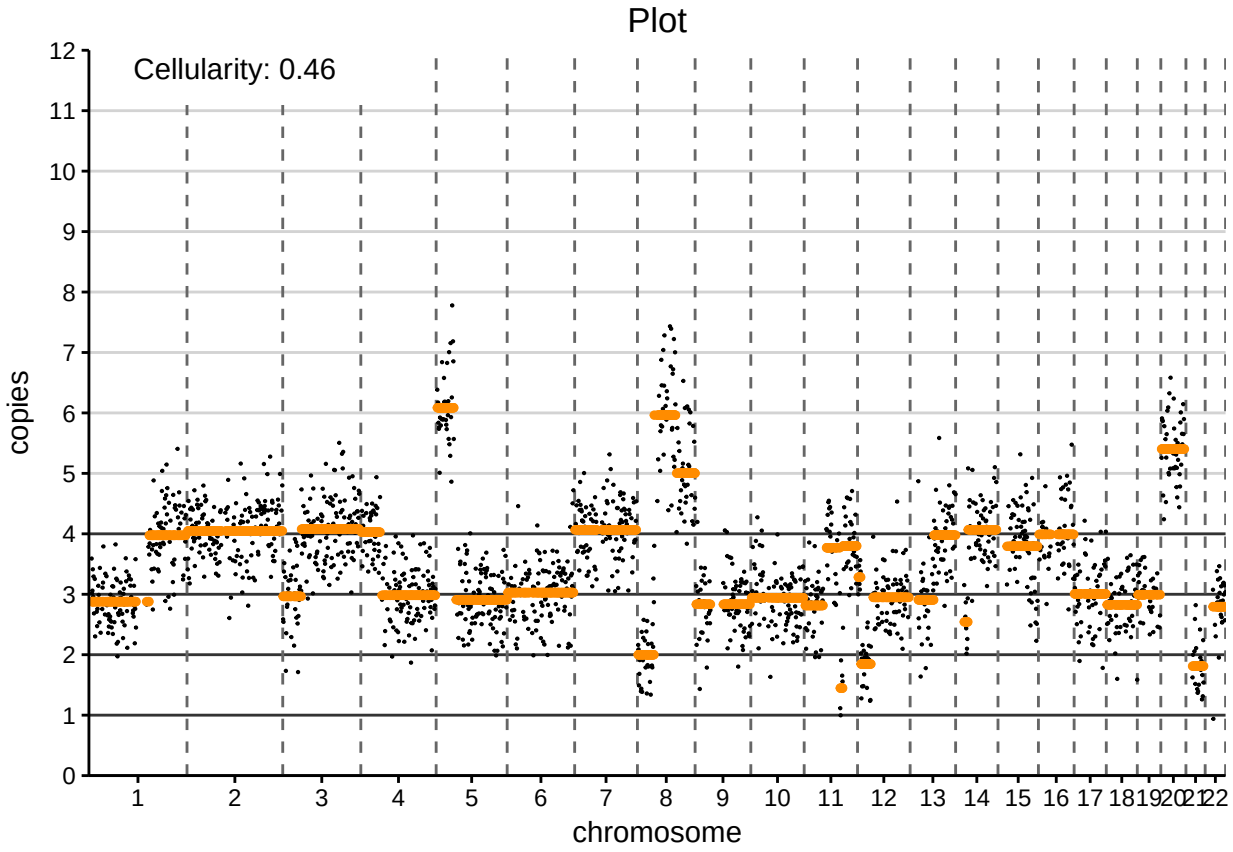
```
# To use data from QDNAseq-objects, ACE parses it into data frames referred to
# as "templates". Because we will look at sample2 several times, we can just
# create a variable with this data frame.
template <- objectsampletotemplate(object, index = 2)
head(template)
##   bin chr   start   end copynumbers segments
## 1   1   1       1 1e+06          NA       NA
## 2   2   1 1000001 2e+06          NA       NA
## 3   3   1 2000001 3e+06          NA       NA
## 4   4   1 3000001 4e+06          NA       NA
## 5   5   1 4000001 5e+06          NA       NA
## 6   6   1 5000001 6e+06          NA       NA
head(na.exclude(template))
```

```
##    bin chr    start     end copynumbers  segments
## 7    7   1  6000001 7.0e+06   1.1070897 0.9743061
## 8    8   1  7000001 8.0e+06   0.9997716 0.9743061
## 9    9   1  8000001 9.0e+06   1.0081438 0.9743061
## 10  10   1  9000001 1.0e+07   1.0739445 0.9743061
## 11  11   1 10000001 1.1e+07   0.9892931 0.9743061
## 12  12   1 11000001 1.2e+07   1.0517785 0.9743061
# The template has the raw data from QDNAseq
median(na.exclude(template$segments))
## [1] 1.002168
# That number looks familiar ... but suppose I am not happy with it?
# You could find the values of all segments by doing
unique(na.exclude(template$segments))
##  [1] 0.9743061 1.1786904 1.1922945 0.9903714 1.1978319 1.1886100 0.9937348
##  [8] 1.5721273 0.9799809 1.0021684 1.1956154 0.8105095 1.5493462 1.3703144
## [15] 0.9659909 0.9863428 0.9623434 1.1405094 0.7075958 1.1459254 1.0497091
## [22] 0.7818852 0.9873491 0.9796466 1.1794238 0.9116813 1.1959847 1.1448412
## [29] 1.1816252 0.9981159 0.9646637 0.9957557 1.4448269 0.7753297 0.9587029
# Personally I like the rle function, because it also shows you the "length" of
# a segment
segmentdata <- rle(as.vector(na.exclude(template$segments)))
plot(rep(segmentdata$values, segmentdata$lengths))
```




```
# Yes, it can be that easy to make something resembling a copy number plot :-)
# Let's say we are sure that the segment with value 0.8105095 is 2N, we can use
# that number as new standard. First we need to find a good model that fits with
```

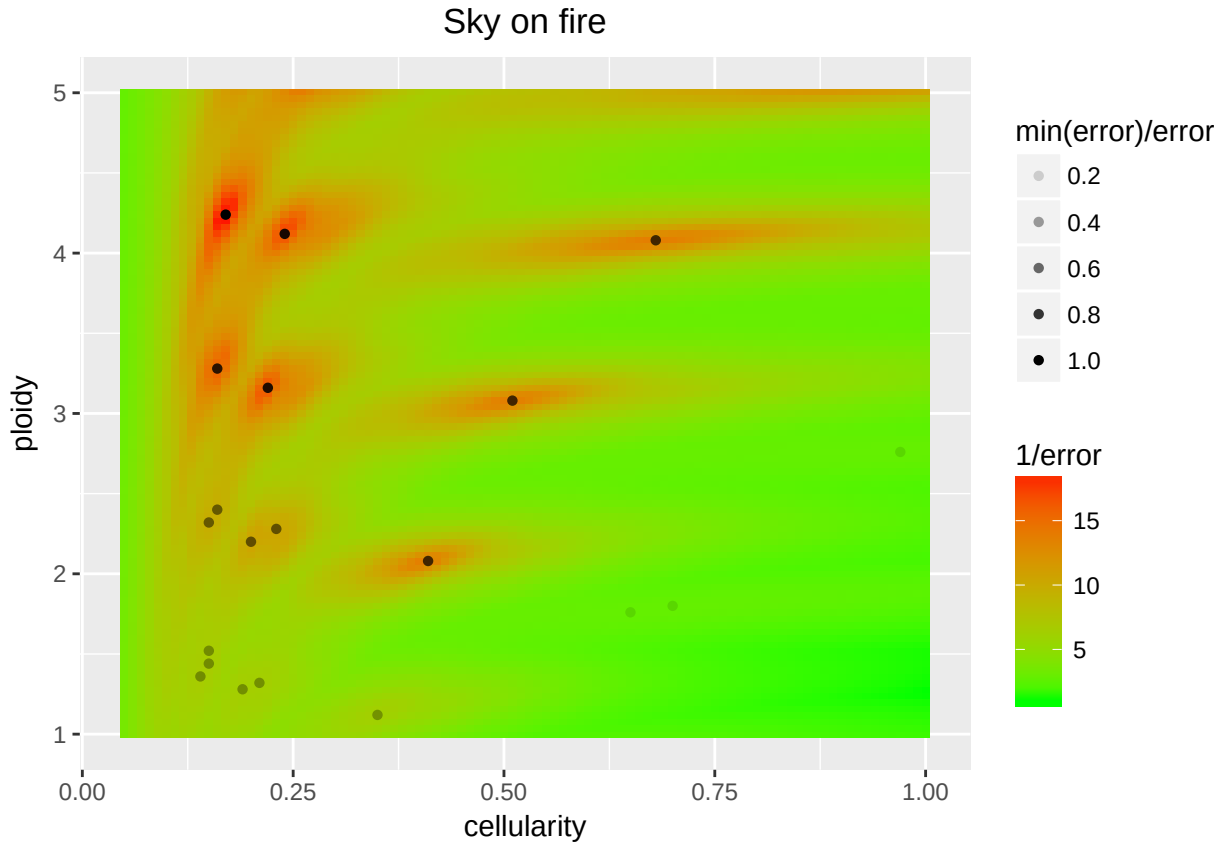
```
# this hypothesis
model <- singlemodel(template, ploidy = 2, standard = 0.8105095)
data.frame(model$minima, model$rerror)
##   model.minima model.rerror
## 1         0.16    0.1548715
## 2         0.19    0.1166904
## 3         0.23    0.1092674
## 4         0.32    0.2156666
## 5         0.46    0.1359516
## 6         0.65    0.5761402
singleplot(template, cellularity = 0.46, ploidy = 2, standard = 0.8105095)
```




```
# Choosing a lower standard shifts the absolute copy numbers up
# As a result we ended up with the exact same model as the 3N fit
# Note that I can directly use the template for the singlemodel and singleplot functions
```

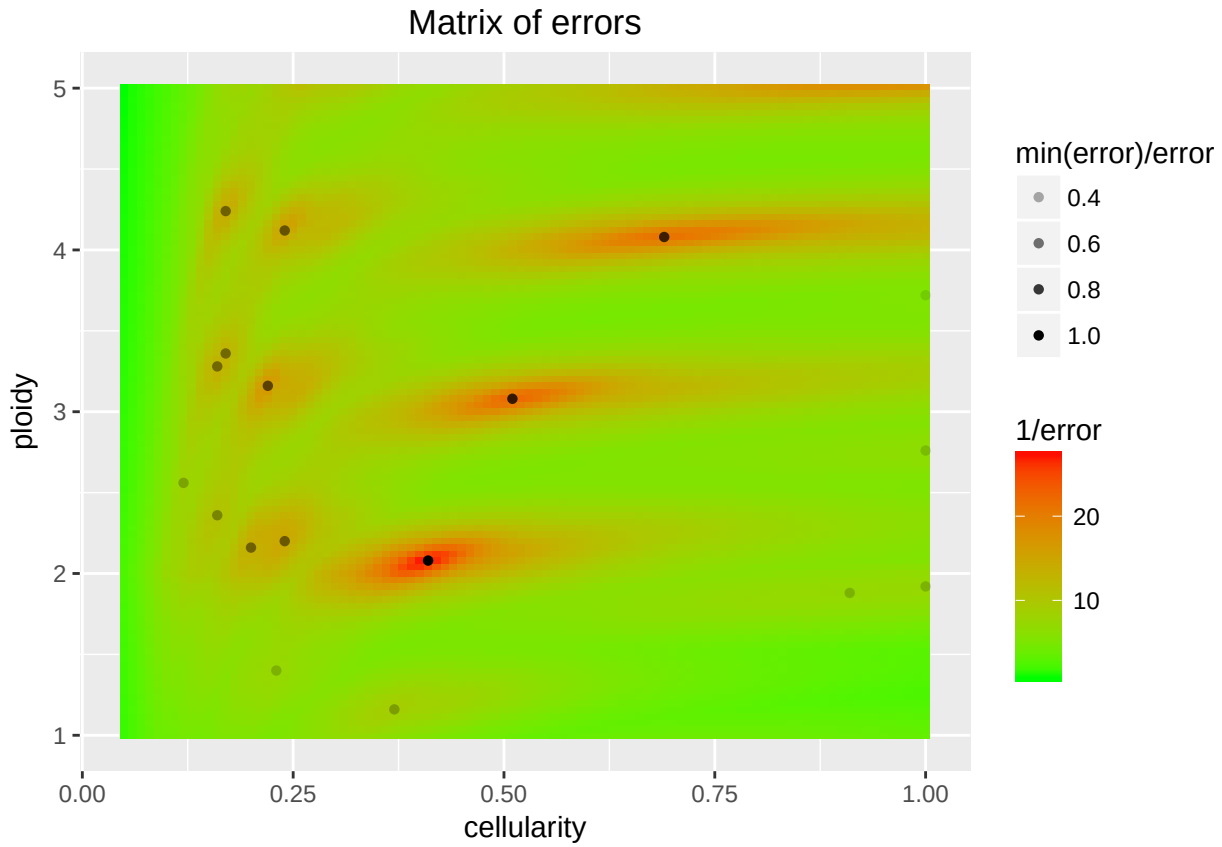
Instead of tinkering with the standard and ploidy, you can also forget about the standard, and fit any possible ploidy to segment value 1 (make sure your segmented data is normalized to 1). This is what the function squaremodel allows you to do. Instead of the boring error plots, you now get a colorful view of the relative error as a function of both ploidy and cellularity! Awesome! User beware though. You'll find ACE can make great fits at interesting ploidies, but they often don't make sense from a biological perspective. On top of the penalty for low cellularities, you can consider to use a penalty for ploidies (penploidy) that diverge a lot from two. All that trickiness urges me to advise against the use of this function as the primary means of finding fits. That, and the fact that the function takes more time to compute. Instead, use it to troubleshoot difficult samples or create visually appealing output to impress your colleagues. The function has some options to specify the range of ploidy (ptop and pbottom) and the resolution on the y-axis (prows).

```
# Let's continue with the template we made for sample 2, and just see what happens...
# Since the output of this one is pretty big, I'm saving it to a variable
sqmodel <- squaremodel(template)
ls(sqmodel)
## [1] "errordf"     "errormatrix" "matrixplot"  "method"      "minimadf"
## [6] "minimatrix"  "penalty"     "penploidy"
# Yes, you get a lot of bang for your buck. You get back the parameters it used,
# but also the errors of all combinations tested in both a matrix and a
# dataframe, and where minima are found. But the fun comes in form of the
# matrixplot.
sqmodel$matrixplot + ggtitle("Sky on fire")
```

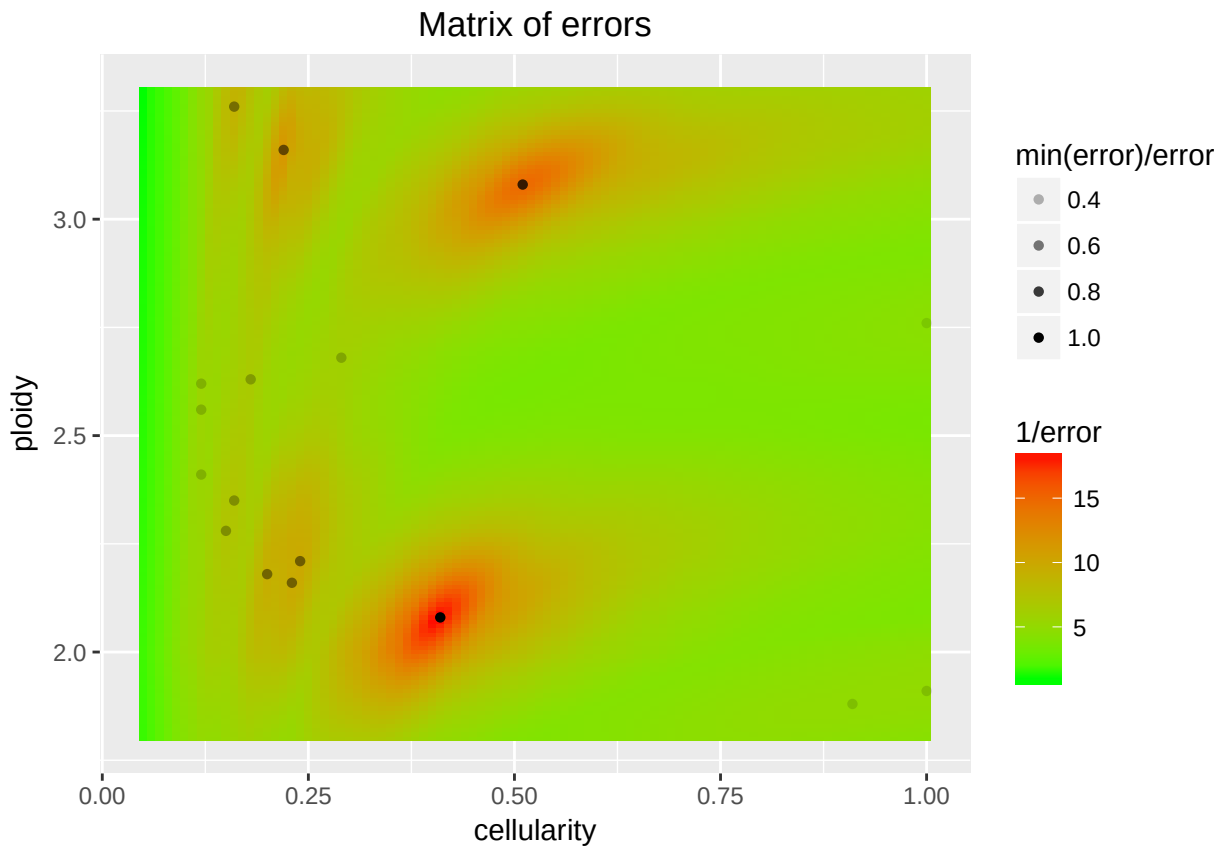



```
# You can find your minima of interest in the minimadf
# Note that the minima are sorted by their relative error
head(sqmodel$minimadf, 10)
##      ploidy cellularity      error minimum
## 1837   4.24        0.17 0.05322983    TRUE
## 2132   4.12        0.24 0.05869922    TRUE
## 4434   3.16        0.22 0.06093845    TRUE
## 4140   3.28        0.16 0.06431760    TRUE
## 2272   4.08        0.68 0.07255574    TRUE
## 4655   3.08        0.51 0.07257179    TRUE
## 7045   2.08        0.41 0.07296175    TRUE
## 6736   2.20        0.20 0.09457471    TRUE
## 6547   2.28        0.23 0.09645399    TRUE
## 6252   2.40        0.16 0.10250067    TRUE
```

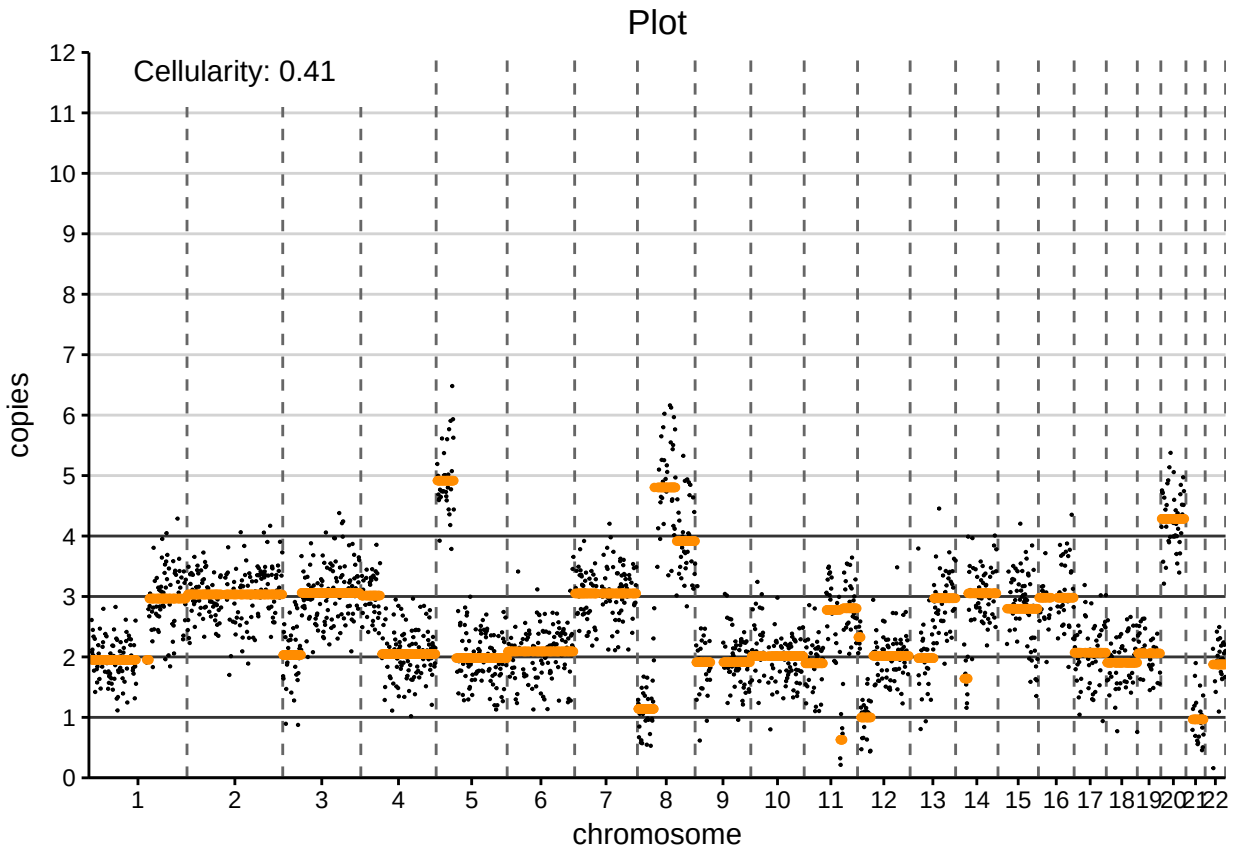
```
# Guess I was warned about this ... squaremodel is a bit fithappy; time to put
# the thumbscrews on the fit
squaremodel(template, penalty = 0.5, penploidy = 0.5)$matrixplot
```




```
# Much better. Additionally you can play around with the range and resolution
sqmodel <- squaremodel(template, prows = 150, ptop = 3.3, pbottom = 1.8,
                       penalty = 0.5, penploidy = 0.5)
sqmodel$matrixplot
```




```
# Going for higher ploidy resolution is not recommended! Not only will it take
# much longer to run, it will start distinguishing minima that correspond with
# basically the same model.
head(sqmodel$minimadf, 10)
##       ploidy cellularity      error minimum
## 11749   2.08        0.41 0.05489686    TRUE
## 2159    3.08        0.51 0.06794331    TRUE
## 1362    3.16        0.22 0.08851955    TRUE
## 10484   2.21        0.24 0.10129842    TRUE
## 10963   2.16        0.23 0.10266600    TRUE
## 10768   2.18        0.20 0.10691679    TRUE
## 396     3.26        0.16 0.11225476    TRUE
## 9132    2.35        0.16 0.13869234    TRUE
## 9803    2.28        0.15 0.14000904    TRUE
## 6446    2.63        0.18 0.15140320    TRUE
singleplot(template, cellularity = 0.41, ploidy = 2.08, standard = 1)
```




```
# Perhaps this fit is a little bit better than our original 2N fit,
# but the difference is negligible.
# Don't forget: standard for squaremodel is by definition 1, but you have to
# specify this in the singleplot function call!
```

This gives you all the tools to find the right fit, and customize your data and plots!

3 Advanced functions

3.1 getadjustedsegments

We have made our pick for the most likely fit. Besides plotting, we can also get the segment information in a data frame. Plus we can get some clue whether a segment is truly subclonal by examining its distance to the closest integer copy number.

```
# Let's use the template dataframe we already created in the previous section
# for sample2. For cellularity and ploidy I'll use the original 2N fit
segmentdf <- getadjustedsegments(template, cellularity = 0.38)
head(segmentdf)
##   Chromosome     Start       End Num_Bins Segment_Mean Segment_Mean2
## 1          1   6000001 1.53e+08       88     1.853674      1.861052
## 2          1 153000001 2.47e+08       71     2.927053      2.939692
## 3          2   1000001 2.43e+08      162     2.998498      3.006641
## 4          3         1 4.70e+07       35     1.938045      1.950609
## 5          3  47000001 1.95e+08      113     3.027579      3.038532
```

```
## 6          4   1000001 4.90e+07       36     2.979148      2.985244
##   Segment_SE Copies P_log10
## 1 0.04181002      2  -3.051
## 2 0.06503895      3  -0.451
## 3 0.03647978      3  -0.068
## 4 0.08913917      2  -0.237
## 5 0.04951716      3  -0.360
## 6 0.06557257      3  -0.085
```

The first five columns are basically what you need as input for ABSOLUTE when it comes to segment data (although for ABSOLUTE you will need to run the function with argument log=TRUE). Again, keep in mind that the genomic locations of the segments are associated with a certain genome build. The first segment mean is the adjusted value calculated from the QDNAseq value. The second segment mean is manually calculated from the adjusted copy number values of the individual bins. I do not know why they are different, but luckily the difference is pretty marginal. Those individual bin values allow us to calculate the standard error of a segment value. This error can be useful to make statements about the subclonality of a segment. I have calculated a P-value, which is the chance that, if the segment had a real value that is the closest integer copy number, it would end up with a segment mean as extreme as seen in Segment_Mean2. This P-value must be interpreted with EXTREME caution, because certain biases can easily create very low P-values.

3.2 linkmutationdata

Imagine we also have mutation data available for these samples. You can use this function to append the segment data at the genomic locations of the mutations. That's already pretty cool, but this function also calculates how many mutant copies it thinks there are, using the mutation frequency. Programs I use have these frequencies represented as percentages, not fractions. If the mutant copies don't make sense, try multiplying with 100.

```
# Luck has it, we actually have mutation data for these samples. Back luck has
# it, quality was very low, so calculated mutant copies are not very precise.
# We use the segment data frame created in the previous section.
# Mutation data can be provided as a file or as a data frame. The result will be
# printed to file or returned as a supplemented data frame respectively.
# I will create the mutation data frame manually here.
Gene <- c("CASP8", "CDKN2A", "TP53")
Chromosome <- c(2, 9, 17)
Position <- c(202149589, 21971186, 7574003)
Frequency <- c(47.46, 36.28, 43.48)
mutationdf <- data.frame(Gene, Chromosome, Position, Frequency)
linkmutationdata(mutationdf, segmentdf, cellularity = 0.38,
                 chrindex = 2, posindex = 3, freqindex = 4)
##     Gene Chromosome  Position Frequency Copynumbers Mutant_copies
## 1  CASP8          2 202149589     47.46    3.006641      2.975646
## 2 CDKN2A          9  21971186     36.28    1.820867      1.844484
## 3   TP53         17   7574003     43.48    1.993213      2.285470
```

As mentioned, the mutation data for both samples was appalling, so don't be discouraged :-) You'll do better. Keep in mind that the program assumes the mutations only occur in tumor cells. Be sure to filter out your SNPs!

3.3 analyzegenomiclocations

Perhaps we just want to know for one or a few specific locations how many copies are in the tumor. Or we quickly want to look up a single mutation without creating a bunch of output (files). That's what this function is for. Again, we need our adjusted segment data from the getadjustedsegments function. Then it is a simple matter of entering the genomic location:

```
analyzegenomiclocations(segmentdf, Chromosome = 1, Position = 26365569)
##   Chromosome Position Copynumbers
## 1          1 26365569    1.861052
```

Multiple locations can be entered by providing vectors. Make sure they are the same length!

```
chr <- c(1,2,3)
pos <- c(2000000,4000000,6000000)
analyzegenomiclocations(segmentdf, Chromosome = chr, Position = pos)
##   Chromosome Position Copynumbers
## 1          1    2e+06          NA
## 2          2    4e+06    3.006641
## 3          3    6e+06    1.950609
```

It will return NA if there is no copy number info for the given position.

You can also provide a vector with frequencies to calculate mutant copies. Note: to make this calculation, you will also have to provide the cellularity (the same number as you used to create the segmentdf dataframe).

```
freq <- c(38,19,0)
analyzegenomiclocations(segmentdf=segmentdf, cellularity = 0.38,
                        Chromosome = chr, Position = pos, Frequency = freq)
##   Chromosome Position Frequency Copynumbers Mutant_copies
## 1          1    2e+06        38          NA            NA
## 2          2    4e+06        19    3.006641      1.191262
## 3          3    6e+06         0    1.950609      0.000000
```

3.4 postanalysisloop

This function was created to automate the above functions in case of larger data sets. You need to have picked your best fits, and recorded the variables cellularity, ploidy, and standard. The last two have defaults, namely 2 and 1. If you don't specify them, make sure the defaults are actually correct. Short description of the default functionality (for the full run-down, consult the function documentation). The function loops through a QDNAseq-object. For each sample, it will try to find the model variables by looking through the models-file's first column. Using the variables, it will calculate adjusted segments and link the mutation data. It will also print new plots for the models, which are returned by the function in a list. Output of this function is always written to disk. You can adjust the code below to run it locally.

```
# Set the correct path
userpath <- "D:/DATA/ACE"
# This function needs a models-file, which should like like this
sample <- c("sample1", "sample2")
cellularity <- c(0.79, 0.38)
ploidy <- c(2, 2)
standard <- c(1, 1)
models <- data.frame(sample, cellularity, ploidy, standard)
write.table(models, file.path(userpath, "models.tsv"), quote = FALSE,
            sep = "\t", na = "", row.names = FALSE)
# Let's make sure we have some mutation data to analyze
# For simplicity, I will just use the same mutation data for both samples
write.table(mutationdf, file.path(userpath, "sample1_mutations.tsv"),
            quote = FALSE, sep = "\t", na = "", row.names = FALSE)
write.table(mutationdf, file.path(userpath, "sample2_mutations.tsv"),
            quote = FALSE, sep = "\t", na = "", row.names = FALSE)
# Let's go!
postanalysisloop(object, inputdir = userpath, postfix = "_mutations",
```

```
                chrindex = 2, posindex = 3, freqindex = 4,
                outputdir = file.path(userpath,"output_loop"), imagetype = 'png')
# note that mutation data is optional!
```

That should cover most of it. The sections below cover some advanced and / or situational issues

4 Advanced use

4.1 Considerations for larger data sets

The model-fitting functionality of ACE is very fast, but some other steps in the process may be hampered by having lots of input. Here are some tips and considerations: If you already have the (segmented!) rds-file(s), use that instead of bam-files. When using bam-files, the function runs the samples through QDNAseq, which has to download bin annotations for all bin sizes. Then it has to bin the samples, normalize the bins, and subsequently segment the bins. Obviously you can save a lot of time and space by reducing the number of bin sizes analyzed, and even more so, using relatively large bin sizes. Try using 500 or 1000 kbp!

The imagetype is perhaps not so important for speed, but it is important for file size and convenience. PDF gives you vector art quality, but its size is directly related to the number of data points. Again, small bin sizes blow up your file size and cause the resulting PDFs to take a long time to load. For anything with binsize 100 kbp and smaller, you might want to go with png.

The summary files can especially become very large. The program might crash if you use png and you have too many samples in your object or directory. For this reason I have created the printsummaries argument. You can set it to FALSE if you don't want any summary files, or you can set it to 2 if you only want the summary of error lists.

4.2 Error methods

To be a bit blunt, the error method is just a means to an end. That said, it is possible that the default error method "RMSE" (root mean squared error) does not give the desired result. Imagine two segments: a segment with 100 bins and adjusted segment value 2.2 and a segment with 50 bins and adjusted segment value 2.4. Both segments will best fit to 2N. In case of RMSE, the error of the segments is (roughly) calculated as follows. sqrt((100x(2.2-2)^2 + 50x(2.4-2)^2)/2) = sqrt((4 + 8)/2). You can see the segment of 50 bins constributes more to the error than the segment of 100 bins, because RMSE penalizes quadratically heavier for larger deviations. Is this what you want to do? That depends. You can argue the opposite: I find it more important that my long segments close to the absolute copy numbers are penalized for deviating, whereas my smaller segments can easily be subclonal and should be penalized relatively less. In that case you can try "SMRE" which as the name implies does it the other way around (takes the square of the mean rooted error, wait, is that even a thing?). If you don't feel all that adventurous, you can go for the mean absolute error, "MAE", which indeed does not do any squaring or rooting, but just averages all errors. Nice and simple. From my experience: I still like to start with "RMSE", because it seems best at giving a few good fits. Sometimes you find your segments a little overextended, in which case your actual cellularity is probably just a few percentage points higher. In this case, you can quickly run the singlemodel function with method = "MAE", or just try to get the perfect fit by manually adjusting the cellularity. Remember: means to an end.

4.3 Penalizing lower cellularities

This parameter was added to penalize fits at lower cellularities. A prime example why you would want to do this can be seen in sample1 of this walkthrough. The penalty parameter corrects each error by dividing it with the cellularity to the power of the penalty. In case of 0, it divides by 1 and has no effect (i.e. no penalty). In case of 1, it divides by the cellularity, meaning an error at a cellularity of 0.05 becomes 20 times bigger. This will generally be too stringent, especially

if you are analyzing samples that actually have a small fraction of aberrant cells. I like to use 0.5 which penalizes with the inverse of the square root of the cellularity.

4.4 Chromosomal subsets

- Generally, ACE will use data from 22 autosomes. Since QDNAseq is also available for mouse data, runACE should work for mice as well, since they have fewer chromosomes.
- The singlemodel and squaremodel functions have an argument to exclude chromosomes. If chromosomes such as sex chromosomes and mitochondrial DNA have no segmented data (as is the default of QDNAseq), they will not influence these functions. If they have segmented data, but you do not want to use them for model fitting, you have to exclude them: exclude = c("X", "Y", "MT"). You can obviously also use the argument "exclude" to exclude specific autosomes.
- If the argument "onlyautosomes" is available, setting this to FALSE will include data from all other chromosomes.
- If the argument "chrsubset" is available, you can use this to specify which chromosomes should be included in the analysis. In plotting functions, you cannot "skip" chromosomes. For instance, chrsubset = c(3:7, 10:12) will plot chromosomes 3 through 12.

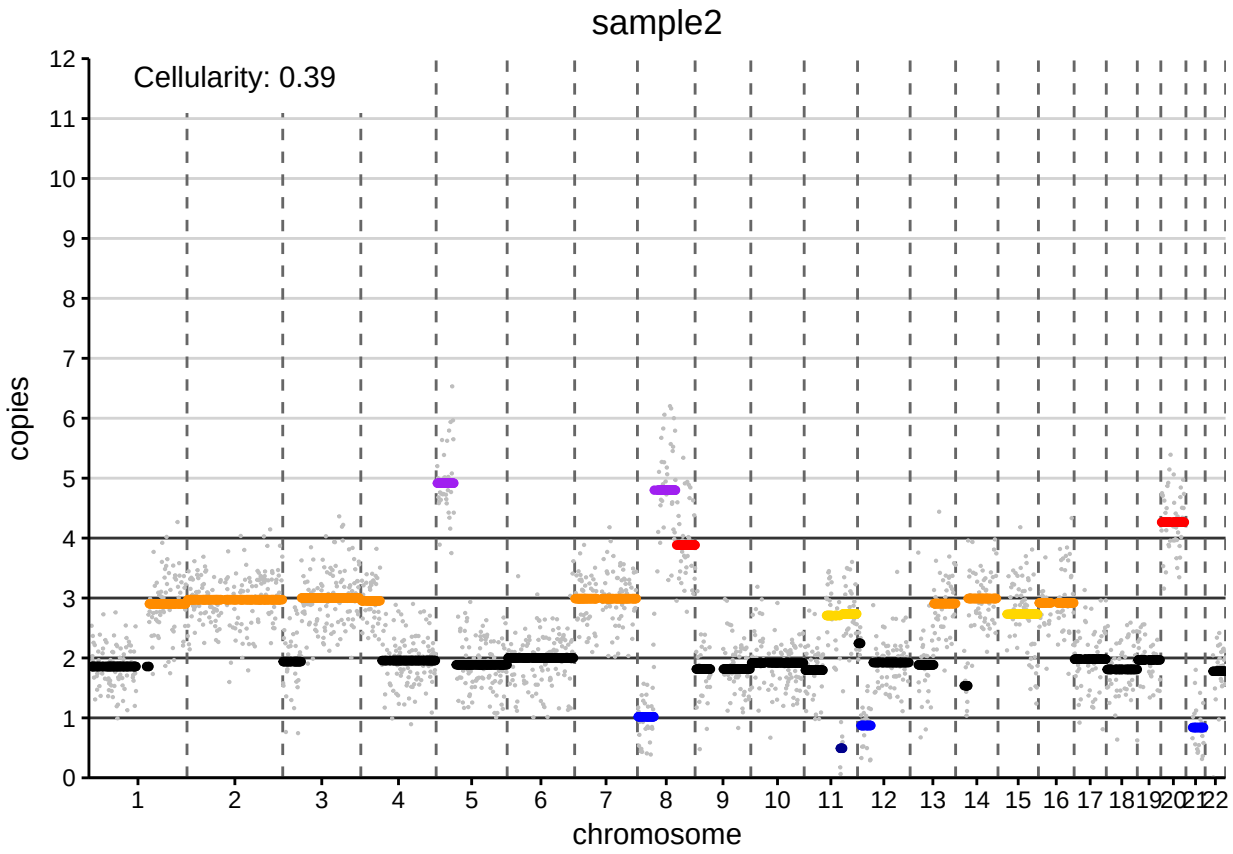
5 Additional Functionality (accessory functions)

Detailed information and examples for below functions can be found in their respective documentation

5.0.1 ACEcall

ACE was not created to perform "calling" of segments. That said, ACEcall can help visualizing gains and losses.

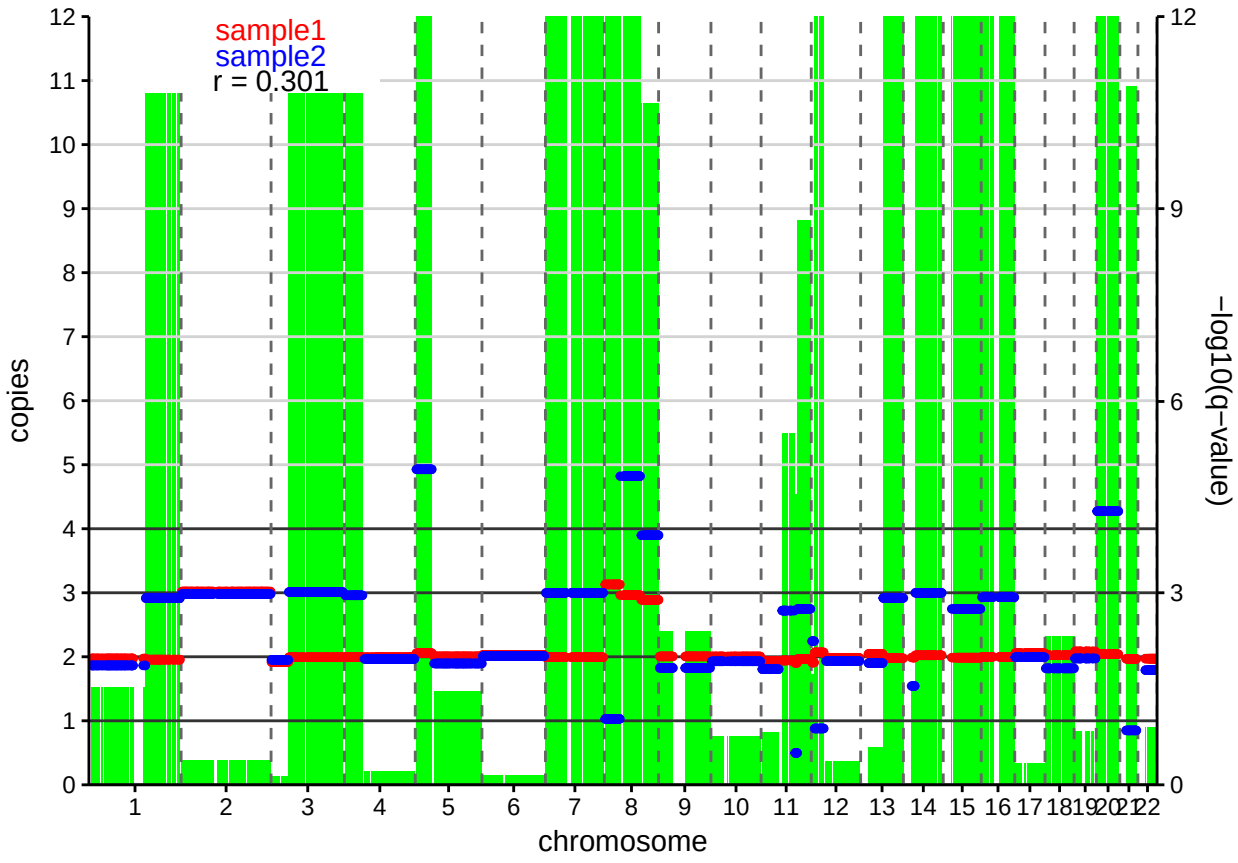
```
ACEcall(object, 2, cellularity = 0.39)$calledplot
```



5.0.2 twosamplecompare

Sometimes it is useful to compare two copy number profiles. You can compare a tumor sample with a matched normal, but you can also compare two samples from the same clonal origin that were separated in space or time, and see if changes have occurred. Additionally, it returns calculations on correlation of segments between the two samples.

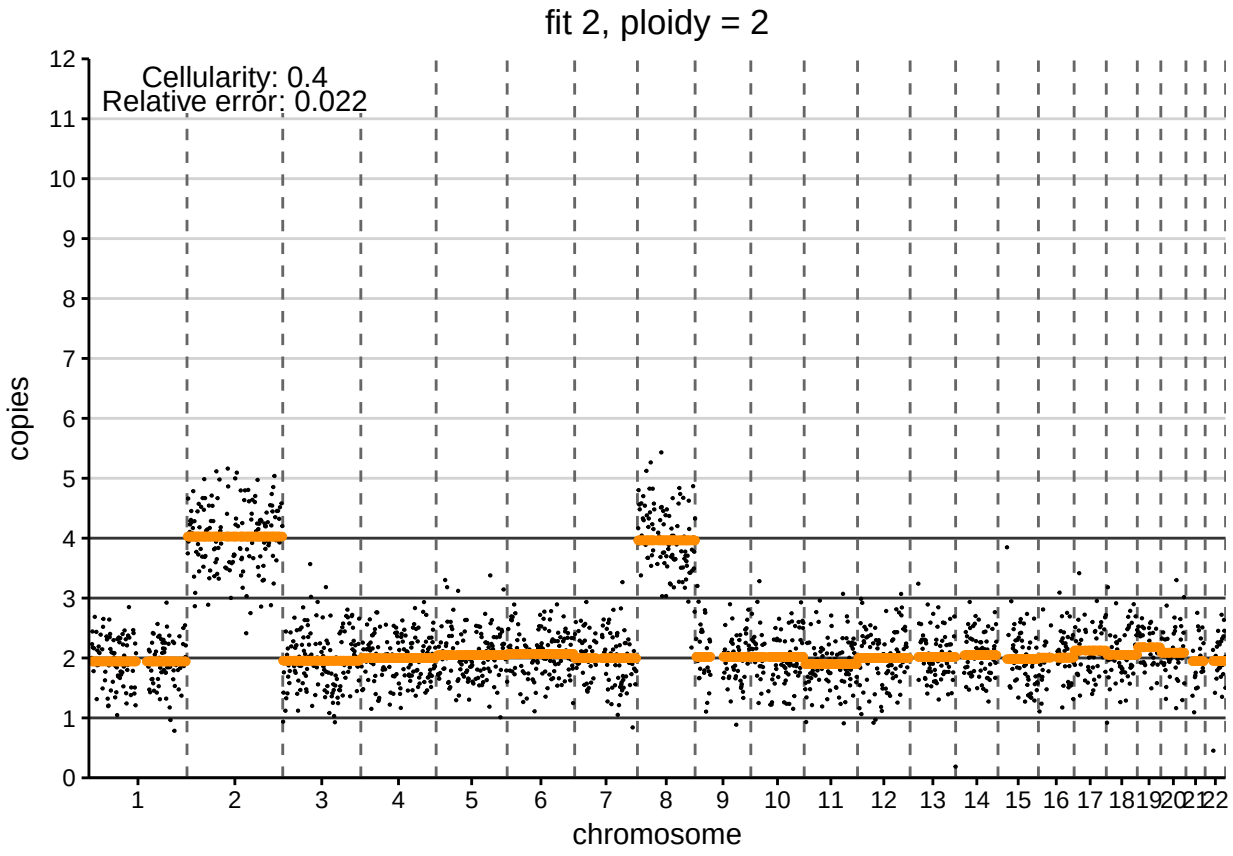
```
# I don't think these two samples are very much related
tsc <- twosamplecompare(object, index1 = 1, index2 = 2,
                        cellularity1 = 0.79, cellularity2 = 0.39)
tsc$compareplot
```



5.0.3 squaremodelsummary

If you want to quickly visualize the copy number plots that correspond with the best fits from a squaremodel call, this function makes a summary. Its default is to print the output to file, but this can be suppressed. It will return the plots as a list.

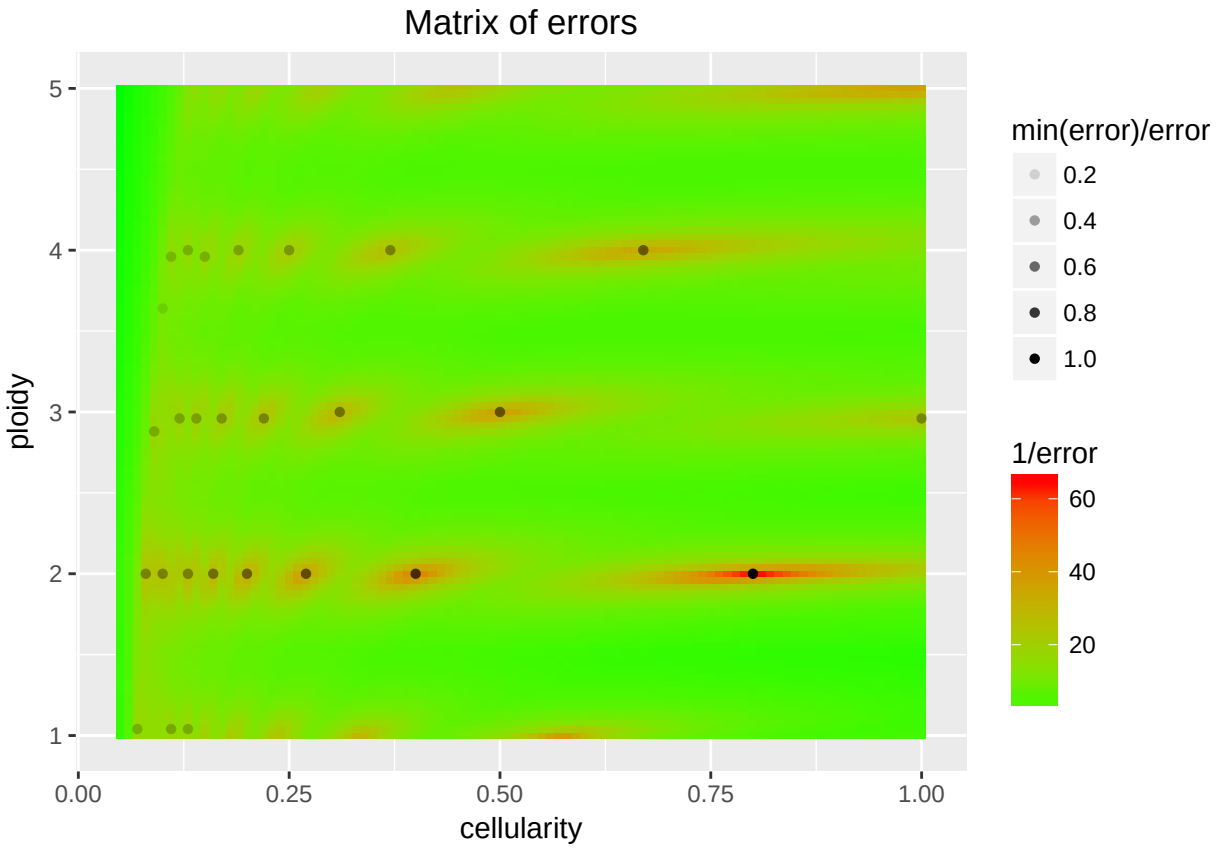
```
# Let's see some more fits for sample1!
sms <- squaremodelsummary(object, QDNAseqobjectsample = 1,
                          squaremodel(object, 1, penalty = 0.5, penploidy = 0.5),
                          printplots = FALSE)
sms[[3]]
```



5.0.4 loopsquaremodel

This function makes squaremodels for all samples in a QDNAseq-object! It can be viewed as the squaremodel equivalent of ploidyplotloop. In contrast though, it is possible to run this function without writing anything to file. It will return a list with the squaremodels. The object summary is a single file with all matrixplots. When printplots = TRUE, the squaremodel summaries are saved to file.

```
# I like squaremodels
lsm <- loopsquaremodel(object, printplots = FALSE, printobjectsummary = FALSE,
                       penalty = 0.5, penploidy = 0.5)
class(lsm)
## [1] "list"
length(lsm)
## [1] 2
class(lsm[[1]])
## [1] "list"
ls(lsm[[1]])
## [1] "errordf"     "errormatrix" "matrixplot"  "method"      "minimadf"
## [6] "minimatrix"  "penalty"     "penploidy"   "samplename"
lsm[[1]]$samplename
## [1] "sample1"
lsm[[1]]$matrixplot
```



The following functions fall outside the scope of this vignette. For those I would like to refer to the function documentation in R:

correlationmatrix

segmentstotemplate

compresstemplate

templatefromequalsegments

6 Information

6.1 Contact

ACE was developed by Jos B. Poell at the VU Medical Center in the department of otolaryngology and head and neck surgery in collaboration with the department of pathology. Source code is available through GitHub: https://github.com/tgac-vumc/ACE Questions regarding ACE can be sent to j.poell@vumc.nl or rh.brakenhoff@vumc.nl

6.2 License

ACE is licensed under GPL

6.3 Reference

There is currently no literature reference for ACE. Please refer to Poell et al. followed by the web site. The package, this document, and the web site will be updated as soon as a literature reference is available.

6.4 Session information

```
sessionInfo()
## R version 3.4.3 (2017-11-30)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 16.04.3 LTS
##
## Matrix products: default
## BLAS: /usr/lib/libblas/libblas.so.3.6.0
## LAPACK: /usr/lib/lapack/liblapack.so.3.6.0
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] parallel  stats     graphics  grDevices utils     datasets  methods
## [8] base
##
## other attached packages:
## [1] ggplot2_2.2.1       QDNAseq_1.10.0      Biobase_2.34.0
## [4] BiocGenerics_0.20.0 ACE_1.0.0           BiocStyle_2.2.1
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_0.12.16          pillar_1.2.1          plyr_1.8.4
##  [4] compiler_3.4.3        GenomeInfoDb_1.10.3   XVector_0.14.1
##  [7] R.methodsS3_1.7.1     R.utils_2.6.0         bitops_1.0-6
## [10] tools_3.4.3           zlibbioc_1.20.0       digest_0.6.15
## [13] tibble_1.4.2          gtable_0.2.0          evaluate_0.10.1
## [16] rlang_0.2.0           yaml_2.1.18           stringr_1.3.0
## [19] knitr_1.20            Biostrings_2.42.1     S4Vectors_0.12.2
## [22] IRanges_2.8.2         stats4_3.4.3          rprojroot_1.3-2
## [25] grid_3.4.3            CGHbase_1.34.0        impute_1.48.0
## [28] marray_1.52.0         DNAcopy_1.48.0        CGHcall_2.36.0
## [31] BiocParallel_1.8.2    rmarkdown_1.9         limma_3.30.13
## [34] magrittr_1.5          scales_0.5.0          backports_1.1.2
## [37] Rsamtools_1.26.2      matrixStats_0.53.1    htmltools_0.3.6
## [40] GenomicRanges_1.26.4 colorspace_1.3-2      labeling_0.3
## [43] stringi_1.1.7         lazyeval_0.2.1        munsell_0.4.3
## [46] RCurl_1.95-4.10       R.oo_1.21.0
```