Iterative normalization for nCounter expression data

Arjun Bhattacharya

Why is normalization necessary?

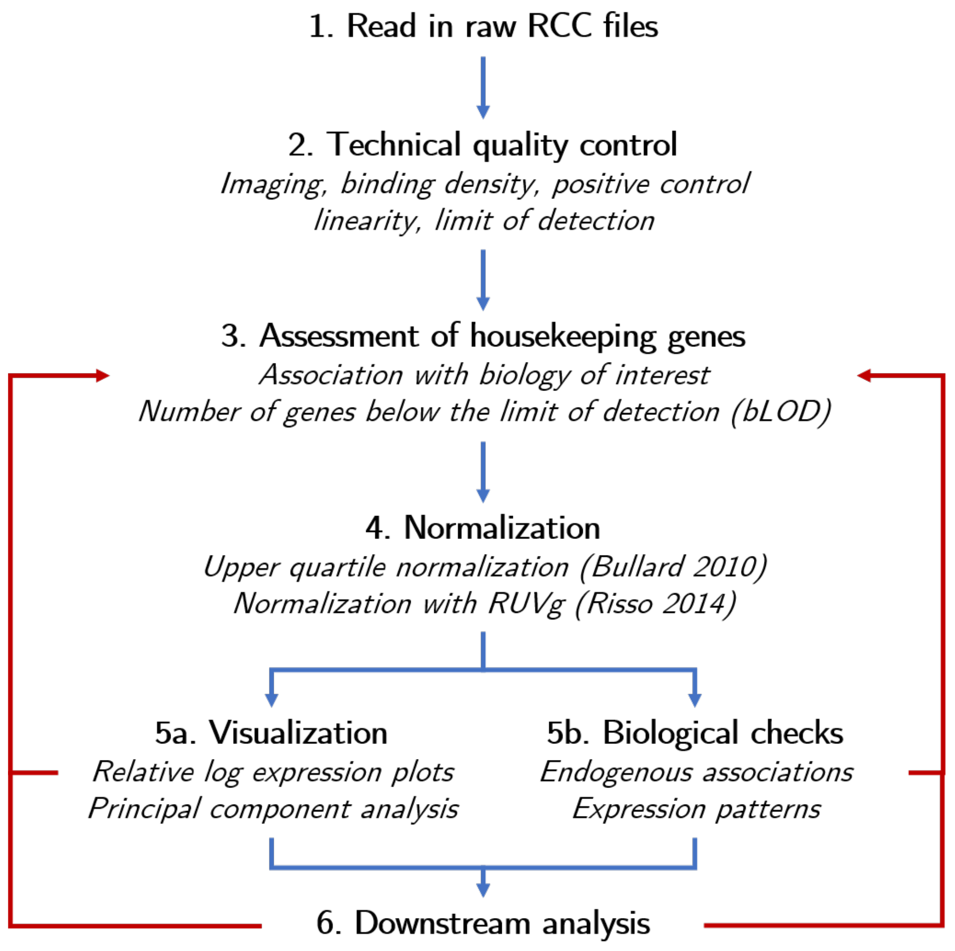
- Raw expression data comes with unwanted technical variation that confounds any statistical associations
- Proper normalization of gene expression data is necessary to draw proper statistical inference
- We can pinpoint that using nSolver's method of normalization leads to artifacts of technical variation persisting post-normalization

Systematic and iterative normalization

We use a more systematic and iterative approach that has three steps:

1. quality control using technical QC flags from nCounter and a few internally defined QC flags,
2. normalization using `DESeq2` and `RUVSeq`, and
3. iterative visualization and biological checks to determine the "quality" of the normalization.

Overview


1. Read in raw RCC files
2. Technical quality control
Imaging, binding density, positive control linearity, limit of detection
3. Assessment of housekeeping genes
Association with biology of interest
Number of genes below the limit of detection (bLOD)
4. Normalization
Upper quartile normalization (Bullard 2010)
Normalization with RUVg (Risso 2014)
5a. Visualization
Relative log expression plots
Principal component analysis
5b. Biological checks
Endogenous associations
Expression patterns
6. Downstream analysis

Reading and quality control

- The easiest, and least technical, way to read in the raw data in nSolver
- Use its functions to create a spreadsheet of raw counts
- You can also output a metadata file that contains the quality controls flags and other miscellaneous variables

Reading in RCC files in R

I'd collect the RCC filenames in a spreadsheet first and write a loop that reads in the file and does the technical quality control. This loop reads in the i-th filename in the vector `files.RCC` and throws the raw counts into the i-th column of the raw expression matrix `raw_expression`. Then, we store QC flags in the data frame `pData`:

```r
source('nanostring_functions.R')
for (i in 1:length(files.RCC)){
    rcc = readRcc(files.RCC[i])
    raw = rcc$Code_Summary
    raw_expression[,i+2] = as.numeric(raw$Count)
    colnames(raw_expression)[i+2] = strsplit(files.RCC[i],
                                             '_')[[1]][1]
    pData[i,2:7] = as.vector(rcc$Sample_Attributes)
    pData$imagingQC[i] = imagingQC(rcc)
    pData$bindingDensityQC[i] = bindingDensityQC(rcc,.05,2.25)
    pData$limitOfDetectionQC[i] = limitOfDetectionQC(rcc)
    pData$positiveLinearityQC[i] = positiveLinQC(rcc)

}
```

Technical quality control flags

I've user-defined the nSolver quality controls in `R`, as they've been identified in the nSolver manual. These are defined in the source file `nanostring_functions.R`. Don't forget to load that in first!

1. imaging quality flag: `imagingQC(rcc = rccInput)`,
2. binding density flag: `bindingDensityQC(rcc = rccInput,high,low)`,
3. limit of detection flag: `limitOfDetectionQC(rcc)`, and
4. linearity of positive controls flag: `positiveLinQC(rcc)`.

Housekeeping control checks

To eventually use RUVSeq, a traditional assumption is that the housekeeping genes are *not* associated with some kind of biology of interest. In many cases, you're not going to have the outcome of interest defined already. But it's good to check if there's differential expression across important biologies of interest, like ER subtype. We can run a really quick negative binomial regression for all the housekeeping genes against ER subtype and flag housekeepers that are potentially differentially expressed across ER subtype:

```r
hk_raw = raw[cIdx,]
pval = vector(length = nrow(hk_raw))
require(MASS)
for (i in 1:nrow(hk_raw)){
  reg = glm.nb(as.numeric(hk_raw[i,]) ~ as.factor(pData$Group))
  pval[i] = coef(summary(reg))[2,4]
}
```

Below limit of detection

It's also important to mark down the number of endogenous and houskeeping genes that are below the limit of detection. You can define the limit of detection however you want (i.e. the mean of the negative controls or the mean of the negative controls with a buffer of 2 standard deviations). This example code defines it as the mean of the negative controls minus 1 standard deviation:

```
neg_raw = raw[negative_controls,]
lod = colMeans(neg_raw) - apply(neg_raw,2,sd)
num_endogenous_blod = colSums(raw[raw$Code.Class == 'Endogenous',-c(1:2)] < lod)
num_hk_blod = colSums(raw[raw$Code.Class == 'Housekeeping',-c(1:2)] < lod)
```

Try to flag samples that have high numbers of endogenous and houskeeping genes below the limit of detection.

Normalization

Now, once we've done all our quality controls and flagged questionable samples, we can move on to normalization. Don't make any judgement calls about samples right now; we've flagged potentionally problematic samples and we'll can look at them after normalization.

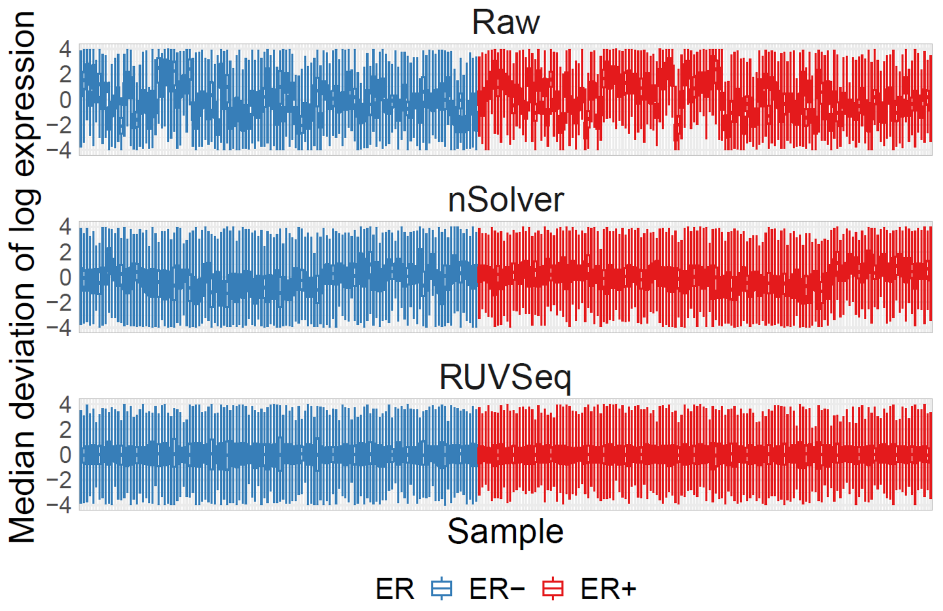
All you need for the `RUV.total()` function is the `raw` matrix of expressions, an associated `pData` data frame that has all the QC and LOD flags, and an `fData` data frame that needs one column that defines the gene names and one column that defined the `Code.Class` (i.e., endogenous, positive control, negative control, housekeeping):

```
RUV.total <- function(raw,pData,fData,k,exclude = NULL){

  library(RUVSeq)
  library(DESeq2)
  library(limma)
  library(matrixStats)

  fData = fData[rownames(raw),]
  int = intersect(rownames(raw),rownames(fData))
  fData = fData[int,]
  raw = raw[int,]

  ## USE DESEQ2 FORMULATION TO INTEGRATE RAW EXPRESSION
  ## PDATA AND FDATA
  set <- newSeqExpressionSet(as.matrix(round(raw)),
                             phenoData=pData,
                             featureData=fData)

  cIdx <- rownames(set)[fData(set)$Class == "Housekeeping"]
  cIdx = cIdx[!(cIdx %in% exclude)]

  ## UPPER QUANTILE NORMALIZATION (BULLARD 2010)
  set <- betweenLaneNormalization(set, which="upper")

  ## RUVg USING HOUSKEEPING GENES
  set <- RUVg(set, cIdx, k=k)
  dds <- DESeqDataSetFromMatrix(counts(set),colData=pData(set),design=~1)
  rowData(dds) <- fData

  ## SIZE FACTOR ESTIMATIONS
  dds <- estimateSizeFactors(dds)
  dds <- estimateDispersionsGeneEst(dds)
  cts <- counts(dds, normalized=TRUE)
  disp <- pmax((rowVars(cts) - rowMeans(cts)),0)/rowMeans(cts)^2
  mcols(dds)$dispGeneEst <- disp
  dds <- estimateDispersionsFit(dds, fitType="mean")

  ## TRANSFORMATION TO THE LOG SPACE WITH A VARIANCE STABILIZING TRANSFORMATION
  vsd <- varianceStabilizingTransformation(dds, blind=FALSE)
  mat <- assay(vsd)

  ## REMOVE THE UNWANTED VARIATION ESTIMATED BY RUVg
  covars <- as.matrix(colData(dds)[,grep("W",colnames(colData(dds))),drop=FALSE])
  mat <- removeBatchEffect(mat, covariates=covars)
  assay(vsd) <- mat
  return(vsd = vsd)

}

norm.dat = RUV.total(raw,pdata,fData,k = 1)
```

Run this for a range of `k` (i.e. `k = 1:5`) and see what the normalized data looks on the backend.

Checks post-normalization

- Always make relative log-expression plots using `EDASeq::plotRLE()` or define your function. It'll give you something that looks like this:



Systemative deviations from the median line (like we see in the middle panel with nSolver-normalized data) are not what we want!

- Make PCA plots plotting the first two principal components, colored by a host of different technical and biological variables. Just run a PCA on the expression data and extract the first two PCs with `prcomp(assay(norm.dat))$x[,1:2]` and color by different variables in your `pData` data frame.
- Look at expression heatmaps and see if things look funky, i.e. immune genes don't cluster together or

PAM50 subtype genes don't cluster. Proliferative genes often tend to be lowly correlated.

- Prioritze endgenous checks and biological associations, like eQTLs or other associations that must be present, like *FOXA1* differential expression across ER subtype.
- If your final analysis is inconsistent, go back to selecting the set of housekeepers, fiddle with the limit of detection, choose a different `k`, and redo normalization.

Remember: normalization is a part of the research process. It's a step in answering your research question, not a catch-all nuisance step that fixes your data in one fell swoope.