SoupX PBMC Demonstration

Matthew Daniel Young

2018-05-16

Introduction

Before we get started with the specifics of example data sets and using the R package, it is worth understanding at a broad level what the problem this package aims to solve is and how it goes about doing it. Of course, the best way of doing this is by reading the pre-print, it's not long I promise. But if you can't be bothered doing that or just want a refresher, I'll try and recap the main points.

In droplet based, single cell RNA-seq experiments, there is always a certain amount of background mRNAs present in the dilution that gets distributed into the droplets with cells and sequenced along with them. The net effect of this is to produce a background contamination that represents expression not from the cell contained within a droplet, but the solution that contained the cells.

This collection of cell free mRNAs floating in the input solution (henceforth referred to as "the soup") is created from cells in the input solution being lysed. Because of this, the soup looks different for each input solution and strongly resembles the expression pattern obtained by summing all the individual cells.

The aim of this package is to provide a way to estimate the composition of this soup, what fraction of UMIs are derived from the soup in each droplet and estimate a decontaminated expression profile for each cell.

The method to do this consists of three parts:

1. Calculate the profile of the soup.
2. Estimate the cell specific contamination fraction.
3. Infer a corrected expression profile for each cell.

Generally, steps 1 and 3 are pretty simple and robust. The part of using this method that requires the most care and thought is step 2, i.e., working out how much background is present in each cell. This is parametrised as rho in the code, with rho=0 meaning no contamination and rho=1 meaning 100% of UMIs in a droplet are soup.

Soup specific genes

To estimate the contamination fraction, we need a set of genes that we know (usually through prior biological knowledge) are not expressed in a cell, so by measuring how much expression we observe we can infer the contamination fraction. That is, we need a set of genes that we know the only source of expression is the soup. The difficulty is that this set of genes is different for every cell.

Say we're using HBB,HBA2 and IGKC to estimate the contamination fraction. Let's now look at what happens in a few hypothetical cells:

Cell 1 - Is a red blood cell so expresses HBB and HBA2, but should not express IGKC. For this cell we want to use IGKC to estimate the contamination fraction but not HBB,HBA2.

Cell 2 - Is a B-Cell so should express IGKC, but not HBB or HBA2. For this cell we want to use HBB and HBA2 to estimate the contamination fraction, but not IGKC.

Cell 3 - Is an endothelial cell, so should not express any of HBB,HBA2 or IGKC. So we want to use all three to estimate the contamination fraction.

Basically we are trying to identify in each cell, a set of genes we know the cell does not express so we can estimate the contamination fraction using the expression we do see.

Now obviously the method doesn't know anything about the biology and we haven't told it what's a B cell, a RBC or anything else. There is nothing stopping you supplying that information if you do have it and that will of course give the best results.

But absent this information, the trick is to use the expression level of the cell to identify when not to use a gene to estimate the contamination fraction in a cell. This is why we want genes with a bimodal expression distribution across cells, because it tells us that when a cell expresses the gene, it expresses it a lot so we can easily identify these cells and not use that gene for the estimation in those cells. Given a set of genes that we suspect may be useful, the function `plotMarkerDistribution` can be used to visualise how this gene's expression is distributed across cells. To continue our example:

Cell 1 - The measured expression of HBB and HBA2 is 10 times what we'd expect if the droplet was filled with soup, so the method will not use either of these genes to calculate rho. On the other hand IGKC is about .05 times the value we'd get for pure soup, so that is used.

Cell 2 - HBB/HBA2 have values around .05 times the soup. IGKC is off the charts at 100 times what we'd expect in the soup. So the method concludes that this cell is expressing IGKC and so uses only HBB/HBA2 to estimate rho.

Cell 3 - All three are at around .05, so all are used to estimate rho.

To get a more accurate estimate, groups with a similar biological function are grouped together so they're either used or excluded as a group. This is why the parameter nonExpressedGeneList is given as a list. Each entry in the list is a group of genes that are grouped biologically. So in our example we would set it like:

```
nonExpressedGeneList = list(HEM = c("HBB", "HBA2"), IG = c("IGKC"))
```

in this example we'd probably want to include other IG genes and Haemoglobin genes even through they're not as high up our bimodal list, as they should correlate biologically. That is,

```
nonExpressedGeneList = list(HEM = c("HBB", "HBA2"), IG = c("IGKC", "IGHG1",
    "IGHG3"))
```

or something similar.

Getting started

You install this package like any other R package. The simplest way is to use the `devtools install_github` function as follows:

```
devtools::install_github("constantAmateur/SoupX")
```

Once installed, you can load the package in the usual way,

```
library(SoupX)
```

PBMC dataset

Like every other single cell tool out there, we are going to use one of the 10X PBMC data sets to demonstrate how to use this package. Specifically, we will use this PBMC dataset. The starting point is to download the raw and filtered cellranger output and extract them to a folder somewhere as follows.

```
mkdir SoupX_pbmc4k_demo
cd SoupX_pbmc4k_demo
wget http://cf.10xgenomics.com/samples/cell-exp/2.1.0/pbmc4k/pbmc4k_raw_gene_bc_matrices.tar.gz
wget http://cf.10xgenomics.com/samples/cell-exp/2.1.0/pbmc4k/pbmc4k_filtered_gene_bc_matrices.tar.gz
```

```
tar zxf pbmc4k_raw_gene_bc_matrices.tar.gz
tar zxf pbmc4k_filtered_gene_bc_matrices.tar.gz
cd ..
```

Loading the data

SoupX comes with a convenience function for loading 10X data processed using cellranger. We will use this to get started.

```
library(SoupX)
dataDirs = c("SoupX_pbmc4k_demo/")
scl = load10X(dataDirs)
```

```
## Loading data for 10X channel Channel1 from SoupX_pbmc4k_demo/
```

This will load the 10X data into a `SoupChannelList` object. This is just a list with some special properties. It has one sub-list per 10X "channel" and some additional entries giving global properties.

Profiling the soup

Having loaded our data, the first thing to do is to estimate what the expression profile of the soup looks like. This is actually done for us automatically by the object construction function `SoupChannel` called by `load10X`. Generally, we'd never really want to explicitly make this call, but just so it's explicit that this is the first part of the method we will show how to do so here.

```
scl = load10X(dataDirs, keepDroplets = TRUE)
scl$channels$Channel1 = estimateSoup(scl$channels$Channel1)
```

Which modifies the `Channel1` `SoupChannel` object to add estimates of the soup expression profile to each channel entry. In our case we only have one channel, but it is worth thinking about the case of multiple channels to understand why the SoupX data and functions are structured as they are and because most experiments will involve multiple samples.

Note that we had to reload the `scl` object to do this. By default, when the soup is estimated the table of droplets `tod` is dropped to reduce the memory requirements. Generally, we don't need the full table of droplets once we have determined what the soup looks like.

Visual sanity checks

Often times really what you want is to get a rough sense of whether the expression of a gene (or group of genes) in a set of cells is derived from the soup or not. At this stage we already have enough information to do just this. Before proceeding, we will briefly discuss how to do this.

Say that we are interested in the expression of the gene IGKC, a key component immunoglobulins (i.e., antibodies) highly expressed by B-cells. Suppose we have used some other method to produce a reduced dimension representation of our data (PCA, tSNE, UMAP or whatever). In this case I have run Seurat in a standard way and produced a tSNE map of the data.

The tSNE coordinates for the PBMC data has been included with the package. For the exact details as to how it was calculated look at `?PBMC_DR`. Let's load this data

```
data(PBMC_DR)
```

Now we can quickly visualise which cells express IGKC by extracting the counts for it from the `SoupChannelList` object.

```
library(ggplot2)
PBMC_DR$IGKC = scl$toc["IGKC", rownames(PBMC_DR)]
gg = ggplot(PBMC_DR, aes(RD1, RD2)) + geom_point(aes(colour = IGKC > 0))
plot(gg)
```

Wow! We know from prior annotation that the cells in the cluster at the bottom are B-cells so should express IGKC. But the cluster on the right is a T-cell population. Taken at face value, we appear to have identified a scattered population of T-cells that are producing antibodies! Start preparing the nature paper!

Before we get too carried away though, perhaps it's worth checking if the expression of IGKC in these scattered cells is more than we would expect by chance from the soup. To really answer this properly, we need to know how much contamination is present in each cell, which will be the focus of the next sections. But we can get a rough idea just by calculating how many counts we would expect for IGKC in each cell, by assuming that cell contained nothing but soup. The function `soupMarkerMap` allows you to visualise the ratio of observed counts for a gene (or set of genes) to this expectation value. Let's try it out,

```
gg = plotMarkerMap(scl, "IGKC", PBMC_DR)
plot(gg)
```

We pass the function three things: the `SoupChanneList` containing information about each channel and it's soup profile, the gene we are interested in and the tSNE co-ordinates of each gene. SoupX does not have any of its own functions for generating tSNE (or any other reduced dimension) co-ordinates, so it is up to us to generate them using something else (Seurat was used in this case).

Looking at the resulting plot, we see that the cells in the B-cell cluster have a reddish colour, indicating that they are expressed far more than we would expect by chance, even if the cell was nothing but soup. Our paradigm changing, antibody producing T-cells do not fare so well. With a few exceptions, they all have a decidedly bluish hue, indicating that is completely plausible that the expression of IGKC in these cells is due to contamination from the soup.

We have made these plots assuming each droplet is pure soup, which is obviously not true. Nevertheless, this can still be a useful quick and easy sanity check to perform.

Estimating the contamination fraction

The most difficult part of correcting for background contamination is accurately estimating how much contamination is present in each cell. In order to do this, we need to find a set of genes that we are as certain will not be expressed in each cell. See the section above on "Soup Specific Genes" for an example which may make this clearer.

For some experiments, such as solid tissue studies where red cell lysis buffer has been used, it is obvious what genes to use for this purpose. In the case of bloody solid tissue, haemoglobin genes will be a ubiquitous contaminant and are not actually produced by any cell other than red blood cells in most contexts. If this is the case, you can skip the next section and proceed straight to estimating contamination.

Picking soup specific genes

However, some times it is not obvious in advance which genes are highly specific to just one population of cells. This is the case with our PBMC data, which is not a solid tissue biopsy and so it is not clear which gene sets to use to estimate the contamination. To aid our selection, SoupX provides a series of (hopefully) useful functions,

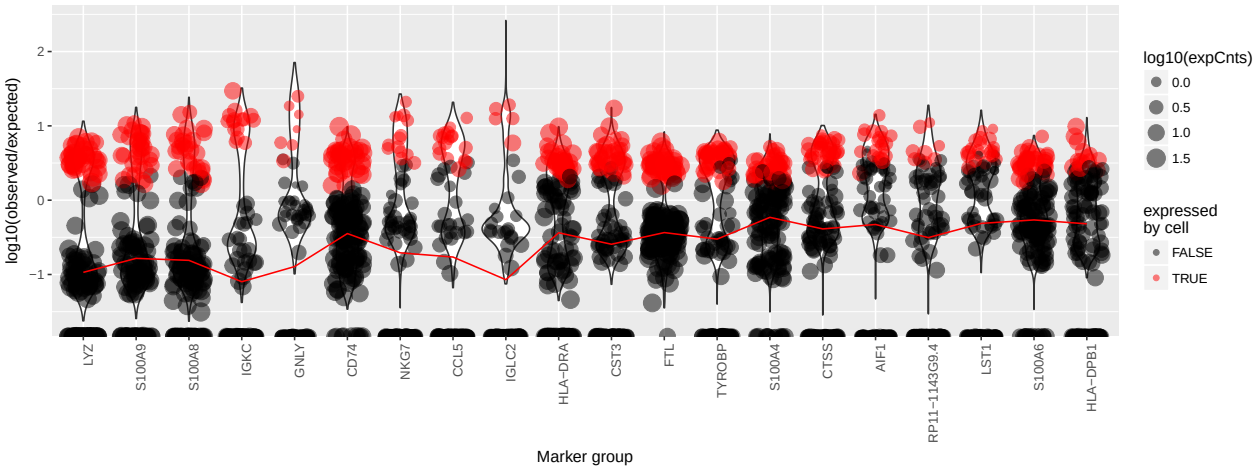
```
scl = inferNonExpressedGenes(scl)
```

```
## Inferring non-expressed genes for channel %sChannel1
```

Running `inferNonExpressedGenes` adds a table to each channel containing genes that are estimated to have highly bimodal expression patterns and a reasonable number of cells in which they are lowly expressed. The reason bimodal genes are useful is that such genes are either highly expressed in a cell (the upper mode of the distribution) and easy to identify and not use for contamination estimation, or expressed due only to the soup. Next we plot the distribution of expression across cells for the first 20 such genes,

```
tstGenes = rownames(scl$channels$Channel1$nonExpressedGenes)[seq(20)]
gg = plotMarkerDistribution(scl, "Channel1", tstGenes)
plot(gg)
```

```
## Warning: Removed 38508 rows containing non-finite values (stat_ydensity).
```



Here I manually extract the top 20 candidates from the newly created `nonExpressedGenes` table for the first (and in our case only) channel. I do this explicitly to show how to extract and interact with the table, the function defaults for `plotMarkerDistribution` will extract the top 20 from this table by default. That is,

```
gg = plotMarkerDistribution(scl, "Channel1")
```

Gives the same plot. Also notice that I explicitly specified the first channel, `Channel1`, rather than just passing the global `SoupChannelList` object `scl`. This is because each channel can and will have different genes that are good markers. So the decision of what genes to use to estimate the contamination must be made on a channel by channel basis. We will find that B-cell specific genes are useful for estimating the contamination in this channel. If we had another channel with only T-cells, these markers would be of no use.

The plot shows the distribution of log10 ratios of observed counts to expected if the cell contained nothing but soup. A guess at which cells definitely express each gene is made and those that are deemed to express it are marked in red. The red line shows the global estimate (i.e., assuming the same contamination fraction for all cells) of the contamination fraction using just that gene.

Looking at this plot, we observe that there are two immunoglobulin genes from the constant region (IGKC and IGLC2) present and they give a consistent estimate of the contamination fraction of around 10% (-1 on

the log10 scale). As we know that it is reasonable to assume that immunoglobulin genes are expressed only in B-cells, we will decide to use their expression in non B-cells to estimate the contamination fraction.

But there's no reason to just use the genes `inferNonExpressedGenes` flagged for us. So let's define a list of all the constant immunoglobulin genes,

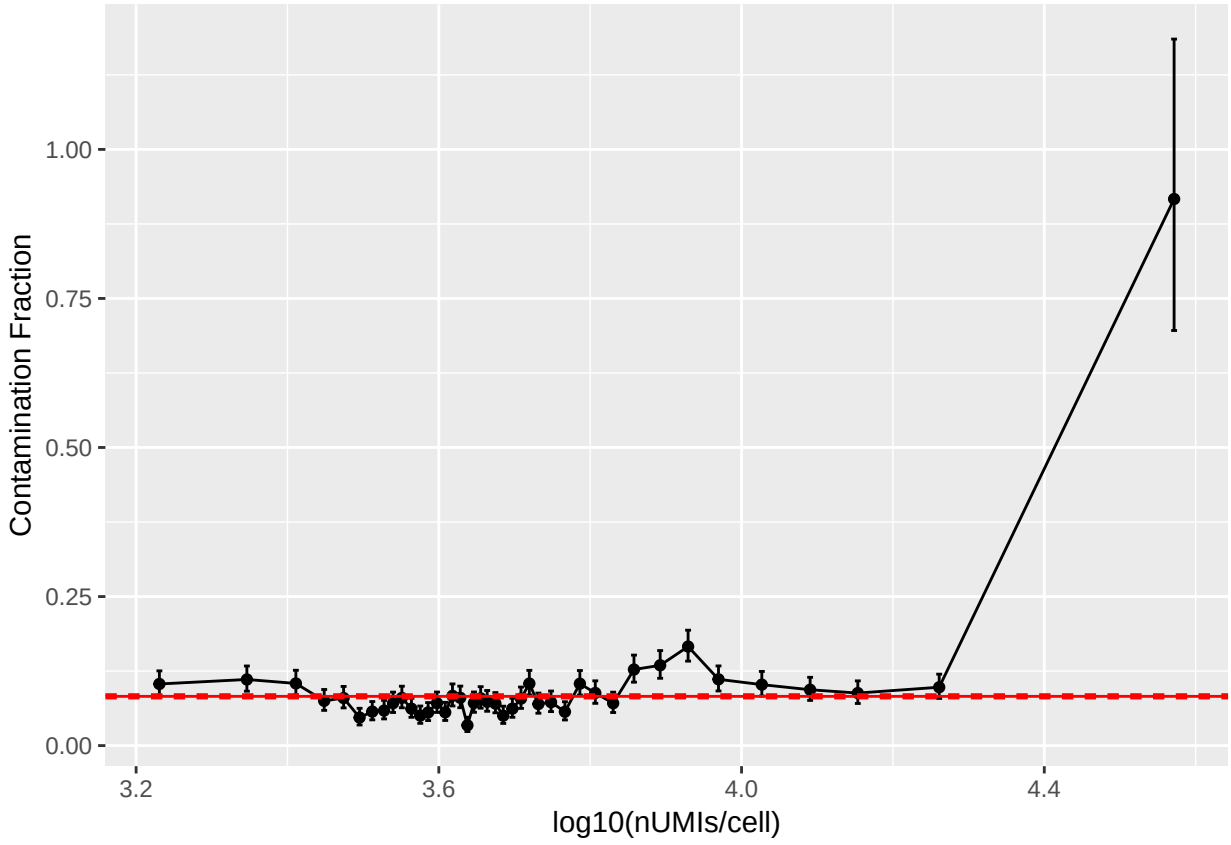
```
igGenes = c("IGHA1", "IGHA2", "IGHG1", "IGHG2", "IGHG3", "IGHG4", "IGHD", "IGHE",
    "IGHM", "IGLC1", "IGLC2", "IGLC3", "IGLC4", "IGLC5", "IGLC6", "IGLC7", "IGKC")
```

it doesn't matter if some of these are not expressed in our data, they will then just not contribute to the estimate.

Estimating the contamination fraction

Having decided on a set of genes with which to estimate the contamination, we perform the estimation as follows,

```
scl = calculateContaminationFraction(scl, "Channel1", list(IG = igGenes))
gg = plotChannelContamination(scl, "Channel1")
plot(gg)
```



The function `calculateContaminationFraction` uses the `igGenes` to estimate the contamination in cells binned together by number of UMIs. We then use `plotChannelContamination` to plot these estimates and the global estimate in red. As there is no strong trend in this data, other than the outlier at high nUMIs/cell that is likely caused by mistakenly including a B-cell in the estimation, we will use the global estimate of the contamination for the correction procedure.

Before we move on to correction, it is worth understanding what `calculateContaminationFraction` is doing in a bit more detail. As before, we feed it our `scl` object and tell it we want to operate on channel `Channel1`

as contamination estimation and correction has to be done on the level of individual channels. The second thing to notice is that we pass the `igGenes` in the rather mysterious format `list(IG=igGenes)`.

There is a good reason for this. In this case we only have one set of genes, IG genes we expect to be expressed only by B-cells, that we are using for the estimation. But ideally, we would have multiple sets of biologically related genes, each useful in different cellular contexts. For instance, if we had lots of bloodly contamination, we would want to use haemoglobin genes to estimate the contamination in anything that isn't a red blood cell. That is,

```
hgGenes = c("HBA1", "HBA2", "HBB", "HBD", "HBE1", "HBG1", "HBG2", "HBM", "HBQ1",
    "HBZ")
scl = calculateContaminationFraction(scl, "Channel1", list(IG = igGenes, HG = hgGenes))
```

That is, the second parameter of `calculateContaminationFraction` should be passed a list of gene families, each of which can be used to estimate the contamination in some subset of the cells. We don't bother using `hgGenes` in the present context as haemoglobulin expression is basically zero.

```
library(Matrix)
rowSums(scl$channels$Channel1$toc[hgGenes, ])
```

```
## HBA1 HBA2  HBB  HBD HBE1 HBG1 HBG2  HBM HBQ1  HBZ
##   69    9    0    0    0    0    3    0    4    0
```

The other thing we have glossed over is that each of these gene families is useful in estimating contamination in a subset of cells only. Specifically, those cells that we can be confident that a particular gene family is not expressed. How does `calculateContaminationFraction` know which cells to use and which to ignore? By default it performs the same statistical test used by `plotMarkerDistribution` or `plotMarkerMap` to exclude any cell that is unambiguously expressing a gene family. This works well at excluding the obvious cells, but can sometimes let through a few cells that are not completely clear cut. The result of this is an over estimate of the contamination. Usually this is rare enough that it is confined to a few bins (the rightmost bin on the above plot being a perfect example), but it is far from ideal.

If we set `excludeMethod = 'thresh'`, we can instead tell `calculateContaminationFraction` to exclude any cell that has a ratio of observed to observed (under the pure soup assumption) that exceeds `exCut`.

Of course, the best thing to do is to have some biological knowledge what each cell is and use this to exclude them. If this is available (or you have another, cleverer method for deciding which cells to exclude), you can pass a matrix indicating which cells (columns) to use which gene families (rows) to estimate the contamination.

I have deliberately picked a "hard" data-set to demonstrate the considerations involved. In practice, there is often one gene family that is extremely specific (e.g. red blood cells and haemoglobin) and basically any sensible cut-off will produce good results.

Cell level contamination fraction

Having estimated the contamination fraction in bins and globally, we now need to decide what value to assign to each cell. You are free to do whatever you think best, but a convenience function `interpolateCellContamination` implements the most common choices. By default, each cell's contamination will be linearly interpolated from the binned estimates we produced above. As there were too many outliers and no strong trend, we will instead use the global estimate for all cells.

```
scl = interpolateCellContamination(scl, "Channel1", useGlobal = TRUE)
```

This will create a new entry in `scl$Channel1` named `rhos` which contains the estimate of the contamination for each cell.

```
head(scl$channels$Channel1$rhos)
```

```
## Channel1___AAACCTGAGAAGGCCT Channel1___AAACCTGAGACAGACC
##                   0.08240242                   0.08240242
## Channel1___AAACCTGAGATAGTCA Channel1___AAACCTGAGCGCCTCA
##                   0.08240242                   0.08240242
## Channel1___AAACCTGAGGCATGGT Channel1___AAACCTGCAAGGTTCT
##                   0.08240242                   0.08240242
```

Correcting expression profile

We have now calculated the contamination fraction for each cell and would like to use this to remove the contamination. SoupX provides two ways of doing this, each with different advantages. The first is to produce an expression matrix, where columns and cells, rows are genes and the entry represents the best estimate of the soup corrected expression fraction for that gene/cell combination. That is, columns all sum to 1. The second is to produce a modified table of counts, where SoupX attempts to remove all the counts that are likely to be soup in origin.

The second option, implemented in the `adjustCounts` function, has the advantage that it maintains the count properties of the data. This allows for downstream tools that require or work best with count data, such as monocle, to be used. Because it explicitly removes counts, it is also easiest to interpret the resulting distribution of expression (see below).

The first option, implemented in the `strainCells` function, has the advantage that it modifies the expression for all genes, not just those that are likely to be heavily contaminated. Because it can in effect remove fractions of counts it can produce a more accurate adjustment of the data. However, the removal of only part of a count for most genes makes the interpretation of patterns of gene expression more challenging. It also requires a data transformation that destroys the count nature of the data, preventing the use of negative binomial models downstream. In practice, this is often not an issue as the transformation is one performed by many popular downstream analysis packages anyway (e.g., both scanpy and monocle).

Both functions add a new matrix to our `scl` object. `strainCells` creates `scl$strainedExp` and `adjustCounts` creates `scl$atoc`.

```
scl = strainCells(scl)
scl = adjustCounts(scl)
```

```
## Calculating probability of each gene being soup

## Calculating probability of the next count being soup

## Filtering table of counts

## Most removed genes for channel Channel1 are:

##
##    S100A9      CD74    S100A8       LYZ     SAP18    NDUFA1    C4orf3
## 0.4364055 0.4258065 0.3806452 0.3709677 0.3608295 0.3582949 0.3534562
##  C11orf31   MORF4L1     UBXN1     SRSF2     PRR13   ANAPC16      CNBP
## 0.3502304 0.3479263 0.3472350 0.3470046 0.3465438 0.3460829 0.3460829
##    SEC61G      CUTA     SEP15    HNRNPC     NEDD8      UBL5     VAMP2
## 0.3451613 0.3444700 0.3442396 0.3435484 0.3435484 0.3435484 0.3433180
##      ARF6     ERP29   COX7A2L   HNRNPDL      SSR2    TMBIM6     EIF3G
## 0.3426267 0.3417051 0.3414747 0.3405530 0.3403226 0.3400922 0.3391705
##    NDUFA3     SPCS1      BRK1   N4BP2L2      SOD1       MIF    COX6A1
## 0.3391705 0.3382488 0.3377880 0.3375576 0.3375576 0.3361751 0.3359447
##    NDUFS5     EIF5A     ATP5J    ATP5A1     CIRBP       FUS       SON
```

```
## 0.3359447 0.3357143 0.3350230 0.3341014 0.3338710 0.3336406 0.3336406
##    TMEM59     ACTR3      GUK1   NDUFA11   ARL6IP4     FKBP8  C19orf43
## 0.3334101 0.3327189 0.3327189 0.3324885 0.3322581 0.3322581 0.3320276
##       JTB    UQCR10       UXT   NDUFB11     USMG5  ATP6V1G1     SEPT7
## 0.3320276 0.3320276 0.3313364 0.3311060 0.3311060 0.3308756 0.3308756
##    EIF4G2     GTF3A     PSMB1   RSL24D1  ATP6V0E1  C11orf58     GPSM3
## 0.3301843 0.3299539 0.3299539 0.3299539 0.3297235 0.3297235 0.3297235
##      ISCU     RBM39   TRMT112    DAZAP2      ARF1    COX7A2    MYL12B
## 0.3297235 0.3297235 0.3297235 0.3292627 0.3288018 0.3288018 0.3288018
##     EIF3M    UBE2D2    PPP1CA      TPM3      GPX4     ATP5O     MZT2B
## 0.3285714 0.3285714 0.3283410 0.3283410 0.3281106 0.3278802 0.3278802
##     COX7B     PNISR   NDUFA13     PARK7      SSR4     SRSF3     ARPC5
## 0.3276498 0.3274194 0.3269585 0.3269585 0.3269585 0.3264977 0.3262673
##  C19orf53     CAPZB   HNRNPA3    RPS27L     SMDT1    PTGES3       RAN
## 0.3262673 0.3260369 0.3260369 0.3260369 0.3258065 0.3255760 0.3255760
##    ATP5J2   HNRNPA0     ATP5D   SLC25A3     UQCRH     CSDE1   TMEM258
## 0.3251152 0.3251152 0.3248848 0.3248848 0.3248848 0.3246544 0.3241935
##      CD53     PCBP2
## 0.3232719 0.3232719
```

Investigating changes in expression

Before proceeding let's have a look at what this has done. We can get a sense for what has been the most strongly decreased by looking at the fraction of cells that were non-zero now set to zero after correction.

```
cntSoggy = rowSums(scl$toc > 0)
cntStrained = rowSums(scl$strainedExp > 0)
mostZeroed = tail(sort((cntSoggy - cntStrained)/cntSoggy), n = 10)
mostZeroed
```

```
##         LTB        JUNB      MT-ND3         FTL        IGKC     HLA-DRA
## 0.006816835 0.007008217 0.010460251 0.018750000 0.020460358 0.022553897
##        CD74      S100A8      S100A9         LYZ
## 0.098994975 0.113066267 0.156281240 0.234976610
```

Notice that a number of the genes on this list are highly specific markers of one cell type or group of cells (CD74/HLA-DRA antigen presenting cells, IGKC B-cells) and others came up on our list of potential cell specific genes. Notice also the presence of the mitochondrial gene MT-ND3. Let's see what happened to these genes under the other correction method.

```
cntAdjusted = rowSums(scl$atoc > 0)
((cntSoggy - cntAdjusted)/cntSoggy)[names(mostZeroed)]
```

```
##       LTB      JUNB    MT-ND3       FTL      IGKC   HLA-DRA      CD74
## 0.3153527 0.2522958 0.1420270 0.2287037 0.5899403 0.3966833 0.4643216
##    S100A8    S100A9       LYZ
## 0.6184949 0.6311230 0.5793451
```

Which illustrates one of the differences between the two methods, that the count adjustment sets far more entries to zero, rather than just reducing them a bit. It is not that the `strainedExp` matrix fails to decrease these genes in cells it should. It is just that it does not have enough evidence to remove all expression from these cells and so decreases them instead. To make this abundantly clear, let's ask the less strict question, "in what fraction of cells does expression of these genes decrease?".

```
# Need to convert table of counts to a normalised expression matrix
soggyExp = t(t(scl$toc)/scl$nUMIs)
```

```
rowSums(scl$strainedExp[names(mostZeroed), ] < soggyExp[names(mostZeroed), ])/rowSums(scl$toc[names(mos
    ] > 0)
```

```
##       LTB      JUNB    MT-ND3       FTL      IGKC   HLA-DRA      CD74
## 0.6511559 0.7544708 0.9590888 0.7557870 0.6291560 0.5807629 0.7497487
##    S100A8    S100A9       LYZ
## 0.6944964 0.7120960 0.6254048
```

So the expression decreases in even more cells than the count adjustment method sets to zero. You might be thinking that since we are **subtracting** soup expression, all expression values should decrease. Clearly this is not the case as the above vector shows. The reason for this is that the expression matrix is normalised so that expression sums to 1 for each cell. So any gene which only decreases by a little bit when the soup is subtracted will actually increase once we re-normalise everything to sum to 1.

Let's now look at which genes are most commonly decreased.

```
tail(sort(rowSums(scl$strainedExp < soggyExp)), n = 10)
```

```
##  MT-ND1    RPS2  MT-CO1  MT-ND3 MT-ATP6  MT-ND2  MT-CYB  MT-ND4  MT-CO2
##    4043    4103    4107    4126    4137    4171    4183    4192    4247
##  MT-CO3
##    4259
```

Wow! All mitochondrial genes bar one. It is not always the case that mitochondria genes are decreased by the SoupX correction, but it tends to be the case. This is because the soup is made up of mRNAs from lysed cells that tend to produce more mitochondrial genes than the stable cells captured in the experiment. As such, the observed mitochondrial fraction is often higher than it should be and must be corrected downwards.

This illustrates one of the key advantages of the expression matrix correction, that it can appropriately increase/decrease expression of genes without having to set them to zero. As most genes have just one or two counts, this fine-tuned adjustment cannot be made by removing counts. Which we see if we look at how often the mitochondrial genes are adjusted to zero

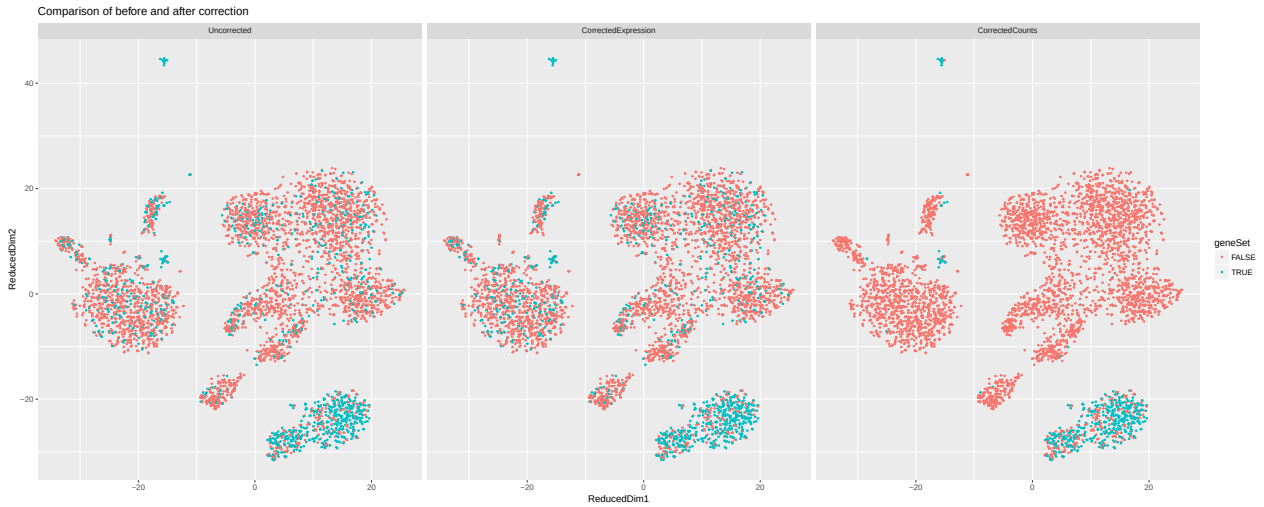
```
(cntSoggy - cntAdjusted)[grep("^MT-", names(cntAdjusted))]
```

```
##  MT-ND1  MT-ND2  MT-CO1  MT-CO2 MT-ATP8 MT-ATP6  MT-CO3  MT-ND3 MT-ND4L
##     988     479     131     259      28     731     234     611    1374
##  MT-ND4  MT-ND5  MT-ND6  MT-CYB
##     414     997     156     821
```

Visualising expression distribution

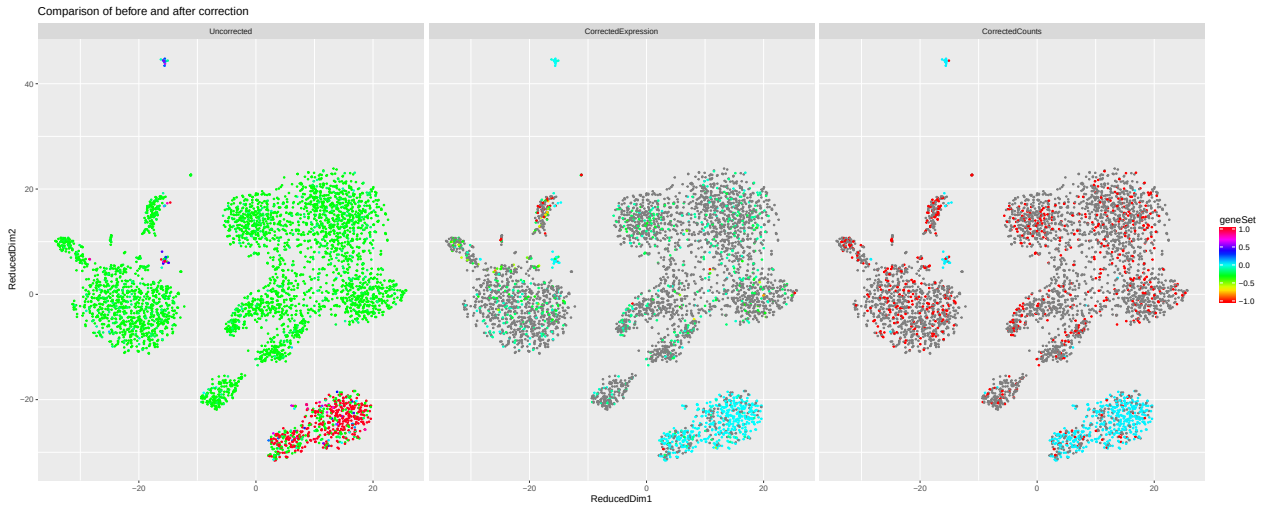
Way back at the start, we did a quick visualisation to look at how the ratio of IGKC expression to pure soup was distributed. Now that we've corrected our data, we can see how that compares to our corrected data. The function `plotChangeMap` can help us with this. By default it plots which cells express a gene and which don't.

```
plotChangeMap(scl, "IGKC", PBMC_DR)
```



which shows us just what we've discussed at length previously, that the expression correction decreases expression rather than removing it entirely (although if you look closely enough you can see that expression has been removed from a number of points). We can also plot the ratio to the uncorrected expression

```
plotChangeMap(scl, "IGKC", PBMC_DR, dataType = "ratio")
```



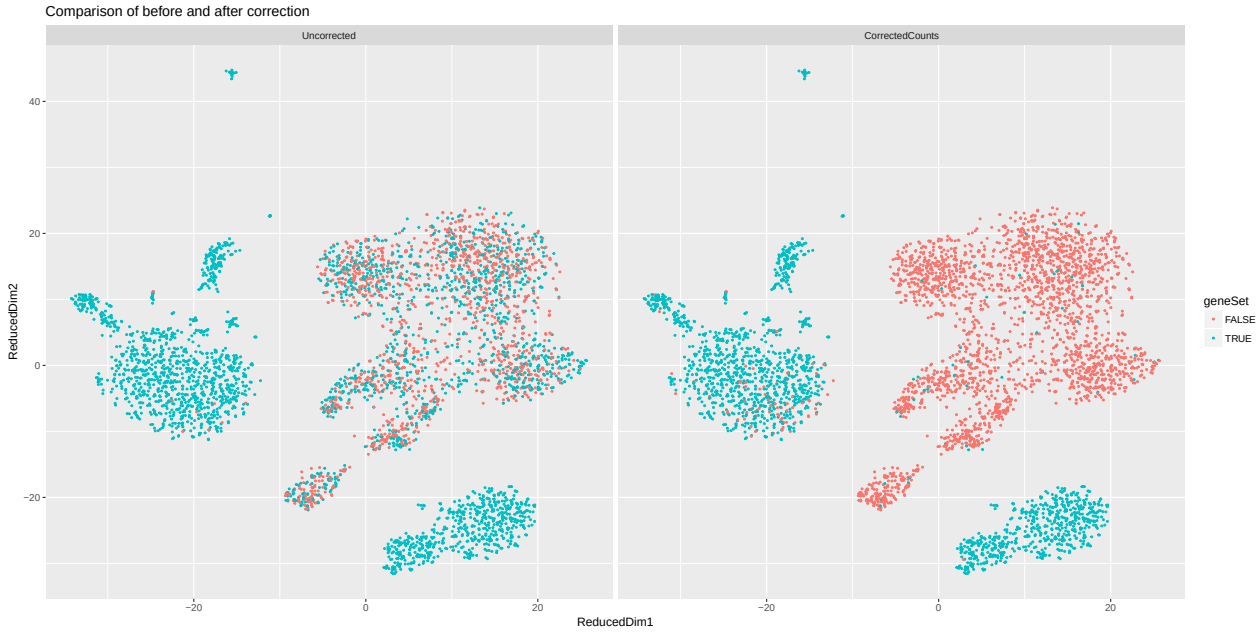
Other than showing that I'm terrible at choosing colour schemes, this shows that the expression has been decreased in the places we expected.

The take away from this is that if you are just interested in seeing "which cells express X", the corrected count maps are the easiest to interpret. Let's take a look at the expression of some other genes.

```
plotChangeMap(scl, "LYZ", PBMC_DR, includePanels = c("Uncorrected", "CorrectedCounts"))
```


```
plotChangeMap(scl, "CD74", PBMC_DR, includePanels = c("Uncorrected", "CorrectedCounts"))
```


```
plotChangeMap(scl, "HLA-DRA", PBMC_DR, includePanels = c("Uncorrected", "CorrectedCounts"))
```




```
plotChangeMap(scl, "IL32", PBMC_DR, includePanels = c("Uncorrected", "CorrectedCounts"))
```


```
plotChangeMap(scl, "TRAC", PBMC_DR, includePanels = c("Uncorrected", "CorrectedCounts"))
```


```
plotChangeMap(scl, "CD3D", PBMC_DR, includePanels = c("Uncorrected", "CorrectedCounts"))
```


```
plotChangeMap(scl, "S100A9", PBMC_DR, includePanels = c("Uncorrected", "CorrectedCounts"))
```


```
plotChangeMap(scl, "S100A8", PBMC_DR, includePanels = c("Uncorrected", "CorrectedCounts"))
```


```
plotChangeMap(scl, "LTB", PBMC_DR, includePanels = c("Uncorrected", "CorrectedCounts"))
```


```
plotChangeMap(scl, "NKG7", PBMC_DR, includePanels = c("Uncorrected", "CorrectedCounts"))
```


```
plotChangeMap(scl, "GNLY", PBMC_DR, includePanels = c("Uncorrected", "CorrectedCounts"))
```


```
plotChangeMap(scl, "CD4", PBMC_DR, includePanels = c("Uncorrected", "CorrectedCounts"))
```


```
plotChangeMap(scl, "CD8A", PBMC_DR, includePanels = c("Uncorrected", "CorrectedCounts"))
```

Clearly the interpretation of which cells are expressing these genes changes quite dramatically when we correct for soup contamination. I have included plots of CD4 and CD8 to show that genes that are not highly expressed in the soup are essentially unchanged by the soup correction.

The change in pattern will be interesting for many other genes, feel free to explore for yourself. In general, the changes tend to be largest for genes that are highly expressed but only in a specific context.

Integrating with downstream tools

Of course, the next thing you'll want to do is to load this corrected expression matrix into some downstream analysis tool and further analyse the data.

If you are using the count correction method, you can just use the adjusted table of counts as input for downstream tools in the same way as you would have if you weren't correcting for background contamination.

To aid integrating the corrected expression matrix with the popular tool Seurat, we can use the `createCleanedSeurat` function, which will create a log-normalised Seurat object.

```
srat = createCleanedSeurat(scl)
srat
```

```
## An object of class seurat in project SeuratProject
##  33694 genes across 4340 samples.
```