SoupX PBMC Demonstration

Matthew Daniel Young

2022-05-24

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 produce a corrected count table with the soup based expression removed.

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 matrix.

In previous versions of SoupX, the estimation of the contamination fraction (step 2) was the part that caused the most difficulty for the user. The contamination fraction 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.

From version 1.3.0 onwards, an automated routine for estimating the contamination fraction is provided, which should be suitable is most circumstances. However, this vignette will still spend a lot of effort explaining how to calculate the contamination fraction "manually". This is because there are still circumstances where manually estimating `rho` is preferable or the only option and it is important to understanding how the method works and how it can fail.

While it is possible to run SoupX without clustering information, you will get far better results if some basic clustering is provided. Therefore, it is **strongly** recommended that you provide some clustering information to SoupX. If you are using 10X data mapped with cellranger, the default clustering produced by cellranger is automatically loaded and used. The results are not strongly sensitive to the clustering used. Seurat with default parameters will also yield similar results.

Quickstart

If you have some 10X data which has been mapped with cellranger, the typical SoupX work flow would be.

```
install.packages("SoupX")
library(SoupX)
# Load data and estimate soup profile
sc = load10X("Path/to/cellranger/outs/folder/")
# Estimate rho
sc = autoEstCont(sc)
# Clean the data
out = adjustCounts(sc)
```

which would produce a new matrix that has had the contaminating reads removed. This can then be used in any downstream analysis in place of the original matrix. Note that by default `adjustCounts` will return non-integer counts. If you require integers for downstream processing either pass out through `round` or set `roundToInt=TRUE` when running `adjustCounts`.

Getting started

You install this package like any other R package. The simplest way is to use the CRAN version by running,

```
install.packages("SoupX")
```

If you want to use the latest experimental features, you can install the development version from github using the devtools `install_github` function as follows:

```
devtools::install_github("constantAmateur/SoupX", ref = "devel")
```

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 temporary folder as follows.

```
tmpDir = tempdir(check = TRUE)
download.file("https://cf.10xgenomics.com/samples/cell-exp/2.1.0/pbmc4k/pbmc4k_raw_gene_bc_matrices.tar
    destfile = file.path(tmpDir, "tod.tar.gz"))
download.file("https://cf.10xgenomics.com/samples/cell-exp/2.1.0/pbmc4k/pbmc4k_filtered_gene_bc_matrices
    destfile = file.path(tmpDir, "toc.tar.gz"))
untar(file.path(tmpDir, "tod.tar.gz"), exdir = tmpDir)
untar(file.path(tmpDir, "toc.tar.gz"), exdir = tmpDir)
```

Loading the data

SoupX comes with a convenience function for loading 10X data processed using cellranger. If you downloaded the data as above you can use it to get started by running,

```
sc = load10X(tmpDir)
```

This will load the 10X data into a `SoupChannel` object. This is just a list with some special properties, storing all the information associated with a single 10X channel. A `SoupChannel` object can also be created manually by supplying a table of droplets and a table of counts. Assuming you have followed the above code to download the PBMC data, you could manually construct a `SoupChannel` by running,

```
toc = Seurat::Read10X(file.path(tmpDir, "filtered_gene_bc_matrices", "GRCh38"))
tod = Seurat::Read10X(file.path(tmpDir, "raw_gene_bc_matrices", "GRCh38"))
sc = SoupChannel(tod, toc)
```

To avoid downloading or including large data files, this vignette will use a pre-loaded and processed object `PBMC_sc`.

```
data(PBMC_sc)
sc = PBMC_sc
sc
```

```
## Channel with 33694 genes and 2170 cells
```

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`. Usually, the default estimation is fine, but it can be done explicitly by setting `calcSoupProfile=FALSE` as follows

```
sc = SoupChannel(tod, toc, calcSoupProfile = FALSE)
sc = estimateSoup(sc)
```

Note that we cannot perform this operation using our pre-saved `PBMC_sc` data as the table of droplets is dropped once the soup profile has been generated to save memory. Generally, we don't need the full table of droplets once we have determined what the soup looks like.

Usually the only reason to not have `estimateSoup` run automatically is if you want to change the default parameters or have some other way of calculating the soup profile. One case where you may want to do the latter is if you only have the table of counts available and not the empty droplets. In this case you can proceed by running

```
library(Matrix)
toc = sc$toc
scNoDrops = SoupChannel(toc, toc, calcSoupProfile = FALSE)
# Calculate soup profile
soupProf = data.frame(row.names = rownames(toc), est = rowSums(toc)/sum(toc), counts = rowSums(toc))
scNoDrops = setSoupProfile(scNoDrops, soupProf)
```

In this case the `setSoupProfile` command is used instead of `estimateSoup` and directly adds the custom estimation of the soup profile to the `SoupChannel` object. Note that we have loaded the `Matrix` library to help us manipulate the sparse matrix `toc`.

Adding extra meta data to the SoupChannel object

We have some extra meta data that it is essential we include in our `SoupChannel` object. In general you can add any meta data by providing a `data.frame` with row names equal to the column names of the `toc` when building the `SoupChannel` object.

However, there are some bits of meta data that are so essential that they have their own special loading functions. The most essential is clustering information. Without it, SoupX will still work, but you won't be able to automatically estimate the contamination fraction and the correction step will be far less effective. Metadata associated with our PBMC dataset is also bundled with SoupX. We can use it to add clustering data by running,

```
data(PBMC_metaData)
sc = setClusters(sc, setNames(PBMC_metaData$Cluster, rownames(PBMC_metaData)))
```

It can also be very useful to be able to visualise our data by providing some kind of dimension reduction for the data. We can do this by running,

```
sc = setDR(sc, PBMC_metaData[colnames(sc$toc), c("RD1", "RD2")])
```

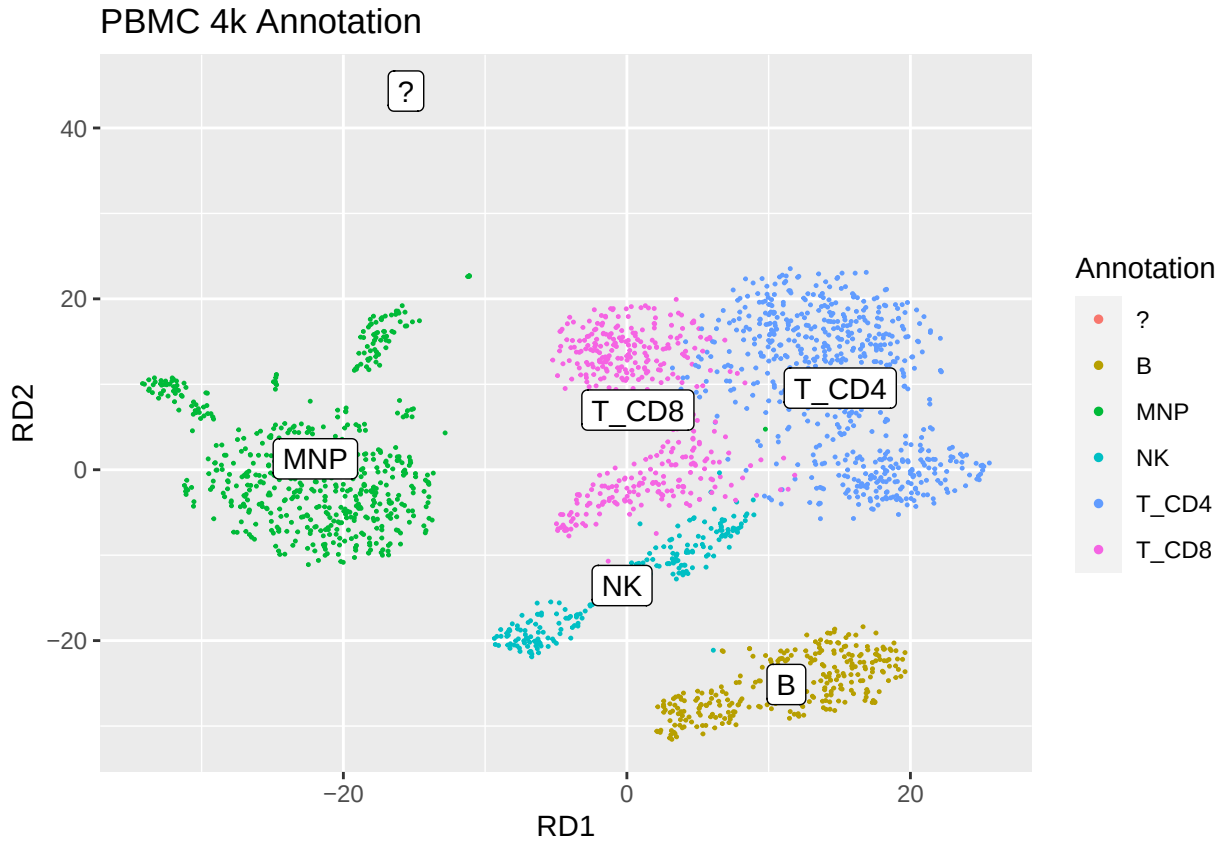
This is usually not needed when using the `load10X` function as the cellranger produced values are automatically loaded.

Visual sanity checks

It is often the case that 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.

Let's start by getting a general overview of our PBMC data by plotting it with the provided annotation.

```
library(ggplot2)
dd = PBMC_metaData[colnames(sc$toc), ]
mids = aggregate(cbind(RD1, RD2) ~ Annotation, data = dd, FUN = mean)
gg = ggplot(dd, aes(RD1, RD2)) + geom_point(aes(colour = Annotation), size = 0.2) +
    geom_label(data = mids, aes(label = Annotation)) + ggtitle("PBMC 4k Annotation") +
    guides(colour = guide_legend(override.aes = list(size = 1)))
plot(gg)
```

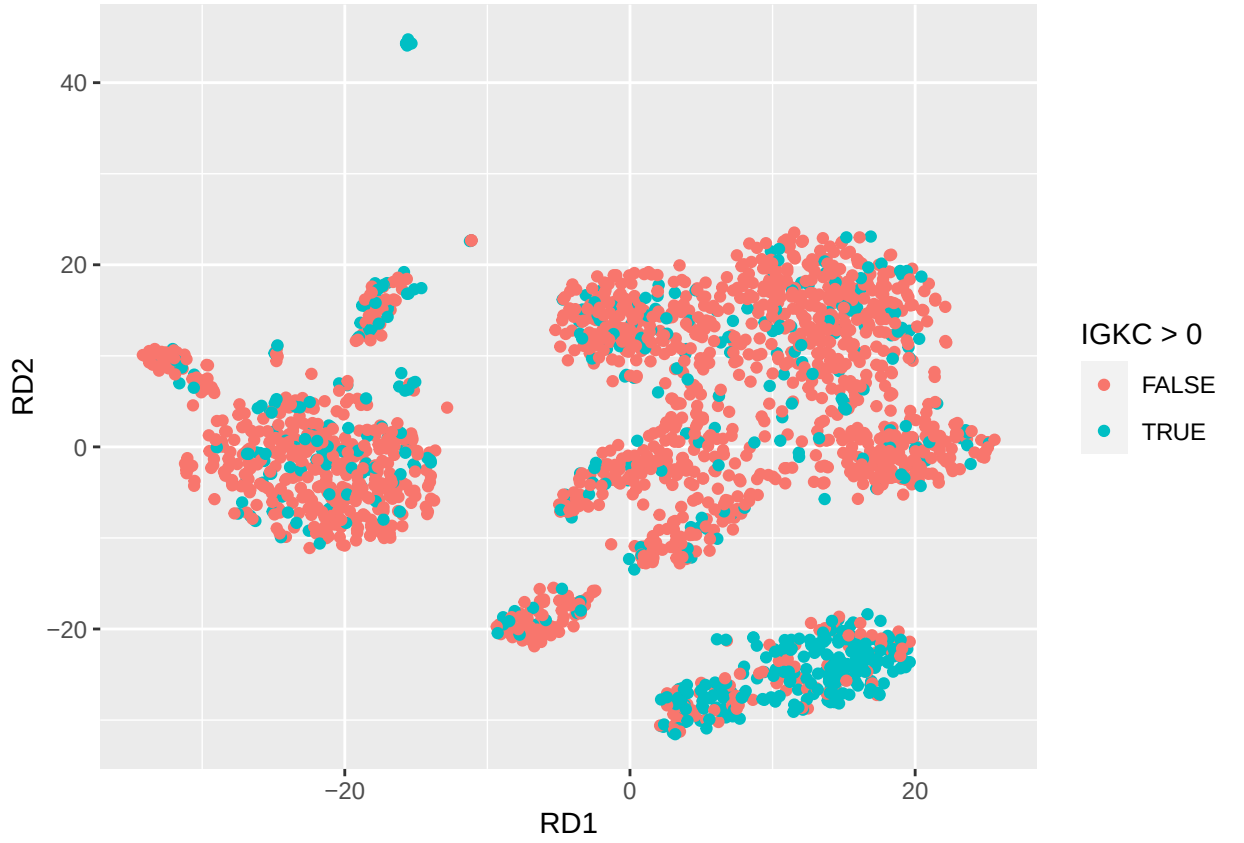


SoupX does not have any of its own functions for generating tSNE (or any other reduced dimension) co-ordinates, so it is up to us to generate them using something else. In this case I have run Seurat in a standard way and produced a tSNE map of the data (see `?PBMC`).

Suppose that we are interested in the expression of the gene *IGKC*, a key component immunoglobulins (i.e.,

antibodies) highly expressed by B-cells. We can quickly visualise which cells express *IGKC* by extracting the counts for it from the `SoupChannel` object.

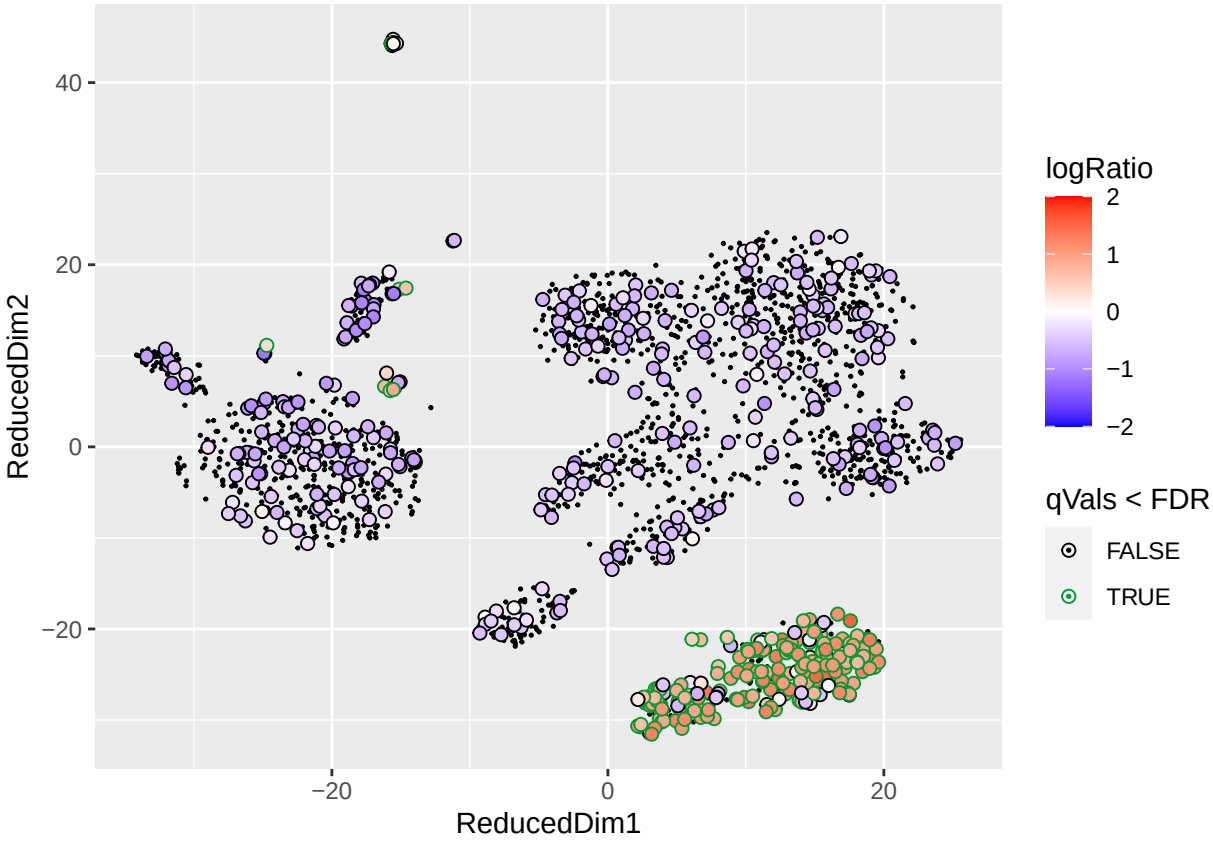
```
dd$IGKC = sc$toc["IGKC", ]
gg = ggplot(dd, aes(RD1, RD2)) + geom_point(aes(colour = IGKC > 0))
plot(gg)
```



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(sc, "IGKC")
plot(gg)
```



There is no need to pass the tSNE coordinates to this function as we stored them in the `sc` object when we ran `setDR` above. 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 background. Our paradigm changing, antibody producing T-cells do not fare so well. 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. Those cells that are shown as dots have zero expression for *IGKC*.

We have made these plots assuming each droplet contains nothing but background contamination, which is obviously not true. Nevertheless, this can still be a useful quick and easy sanity check to perform.

Estimating the contamination fraction

Probably the most difficult part of using SoupX is accurately estimating the level of background contamination (represented as `rho`) in each channel. There are two ways to do this: using the automatic `autoEstCont` method, or manually providing a list of "non expressed genes". This vignette will demonstrate both methods, but we anticipate that the automatic method will be used in most circumstances. Before that we will describe the idea that underpins both approaches; identifying genes that are not expressed by some cells in our data and the expression that we observe for these genes in these cells must be due to contamination.

This is the most challenging part of the method to understand and we have included a lot of detail here. But successfully applying SoupX does not depend on understanding all these details. The key thing to understand is that the contamination fraction estimate is the fraction of your data that will be discarded. If this value is set too low, your "corrected" data will potentially still be highly contaminated. If you set it too high, you will discard real data, although there are good reasons to want to do this at times (see section below). If the contamination fraction is in the right ball park, SoupX will remove most of the contamination. It will generally not matter if this number if off by a few percent.

Note that all modes of determining the contamination fraction add an entry titled `fit` to the `SoupChannel`

object which contains details of how the final estimate was reached.

Manually specifying the contamination fraction

It is worth considering simply manually fixing the contamination fraction at a certain value. This seems like a bad thing to do intuitively, but there are actually good reasons you might want to. When the contamination fraction is set too high, true expression will be removed from your data. However, this is done in such a way that the counts that are most specific to a subset of cells (i.e., good marker genes) will be the absolute last thing to be removed. Because of this, it can be a sensible thing to set a high contamination fraction for a set of experiments and be confident that the vast majority of the contamination has been removed.

Even when you have a good estimate of the contamination fraction, you may want to set the value used artificially higher. SoupX has been designed to be conservative in the sense that it errs on the side of retaining true expression at the cost of letting some contamination to creep through. Our tests show that a well estimated contamination fraction will remove 80-90% of the contamination (i.e. the soup is reduced by an order of magnitude). For most applications this is sufficient. However, in cases where complete removal of contamination is essential, it can be worthwhile to increase the contamination fraction used by SoupX to remove a greater portion of the contamination.

Our experiments indicate that adding 5% extra removes 90-95% of the soup, 10% gets rid of 95-98% and 20% removes 99% or more.

Explicitly setting the contamination fraction can be done by running,

```
sc = setContaminationFraction(sc, 0.2)
```

to set the contamination fraction to 20% for all cells.

Genes to estimate the contamination fraction

To estimate the contamination fraction, we need a set of genes that we know are not expressed in a set of cells, 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 are not expressed by cells of a certain type, so that in these cells the only source of expression is the soup. The difficulty is in identifying these sets of genes and the cells in which they can be assumed to be not expressed.

Note that the purpose of this set of genes is to estimate the contamination fraction and nothing else. These genes play no special role in the actual removal of background associated counts. They are categorically **not** a list of genes to be removed or anything of that sort.

Furthermore, if no good set of genes can be provided, and/or the automatic method fails, it is reasonable to consider setting the contamination fraction manually to something and seeing how your results are effected. A contamination rate of around 0.1 is appropriate for many datasets, but of course every experiment is different.

To make this concrete, let us consider an example. The genes *HBB*,*HBA2* are both haemoglobin genes and so should only be expressed in red blood cells and nowhere else. *IGKC* is an antibody gene produced only by B cells. Suppose we're estimating the contamination then using two sets of genes: HB genes (*HBB* and *HBA2*) and IG genes (*IGKC*). 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 these genes in each cell to identify when not to use a gene to estimate the contamination fraction. This is why the best genes for estimating the contamination fraction are those that are highly expressed in the cells that do use them (like HB or IG genes). Then we can be confident that observing a low level of expression of a set of genes in a cell is due to background contamination, not a low level of mRNA production by the cell.

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 prevent accidentally including cells that genuinely express one of the estimation genes, SoupX will by default exclude any cluster where even one gene has evidence that it expresses a gene. So in the example above, SoupX would not use HB genes to estimate the contamination rate in Cell 1, or any of the cells belonging to the same cluster as Cell 1. This very conservative behaviour is to prevent over-estimation of the contamination fraction.

Clustering is beyond the scope of SoupX, so must be supplied by the user. For 10X data mapped using cellranger, SoupX will automatically pull the graph based clustering produced by cellranger and use that by default.

As indicated above, 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(HB = 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 particularly highly expressed in general, as they should correlate biologically. That is,

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

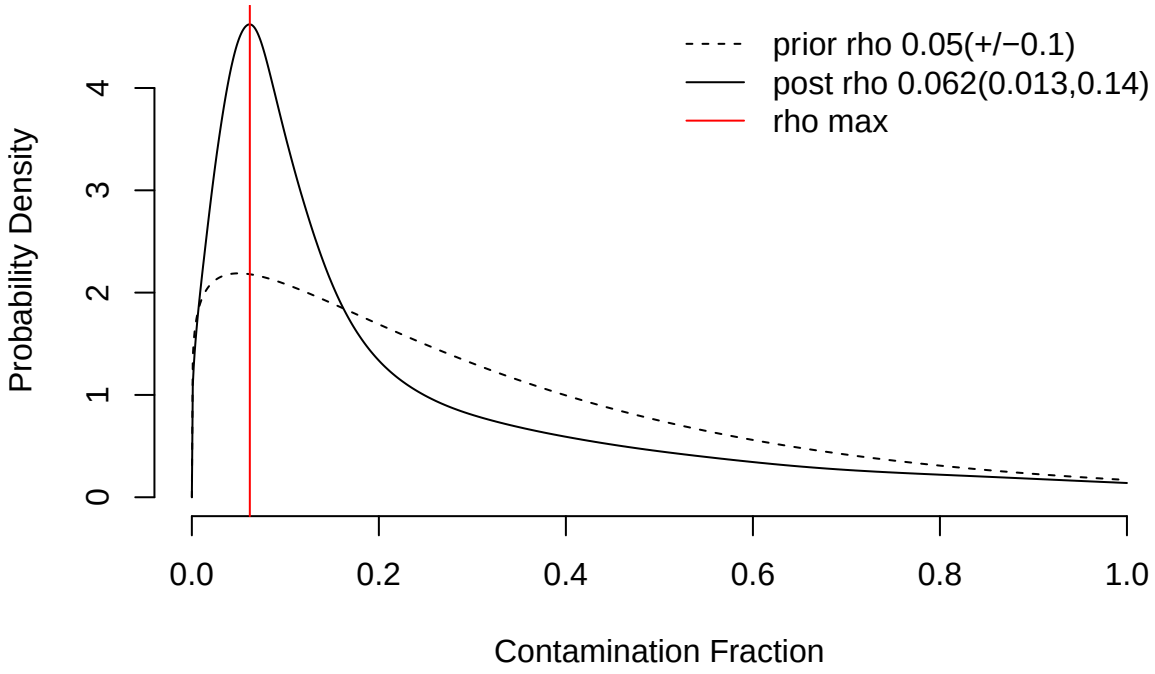
or something similar.

The automated method

Estimating the contamination fraction using the automated is as simple as running:

```
sc = autoEstCont(sc)
```

```
## 786 genes passed tf-idf cut-off and 401 soup quantile filter.  Taking the top 100.
## Using 854 independent estimates of rho.
## Estimated global rho of 0.06
```



This will produce a mysterious looking plot with two distributions and a red line. To understand this plot, we need to understand a little bit about what `autoEstCont` is doing. The basic idea is that it tries to aggregate evidence from many plausible estimators of `rho` and assigns the true contamination fraction to the one that occurs the most often. The reason this works is that incorrect estimates should have no preferred value, while true estimates should cluster around the same value. The solid curve shows something like the frequency of different estimates of `rho`, with a red line indicating its peak, which gives the estimate of `rho`. If you are using the default values for `priorRho` and `priorRhoStdDev` (which you probably should be) you can ignore the dashed line.

For those wanting a more detailed explanation, here is what happens. First the function tries to identify genes that are very specific to one cluster of cells (using `quickMarkers`). The determination of how specific is "very specific" is based on the gene's tf-idf value for the cluster it is specific to. See the `quickMarkers` help or this for an explanation of what this means. The default of `tfidfMin=1` demands that genes by reasonably specific, so if you are getting a low number of genes for estimation you can consider decreasing this value. This list is further reduced by keeping only genes that are "highly expressed" in the soup (as these give more accurate estimates of `rho`), where highly expressed is controlled by `soupQuantile`. The default value sounds strict, but in practice many genes with tf-idf over 1 tend to pass it.

Each of these genes is used to independently estimate `rho` in the usual way. That is, the clusters for which the gene can be confidently said to not be expressed (as determined by `estimateNonExpressingCells`) are used to estimate `rho`. We could then just create a histogram of these estimates and pick the most common value. This would work reasonably well, but we can do better by considering that each estimate has uncertainty associated with it. So instead we calculate the posterior distribution of the contamination fraction for each gene/cluster pair and determine the best estimate of `rho` by finding the peak of the average across all these distributions. This is what is shown as the solid curve in the above plot.

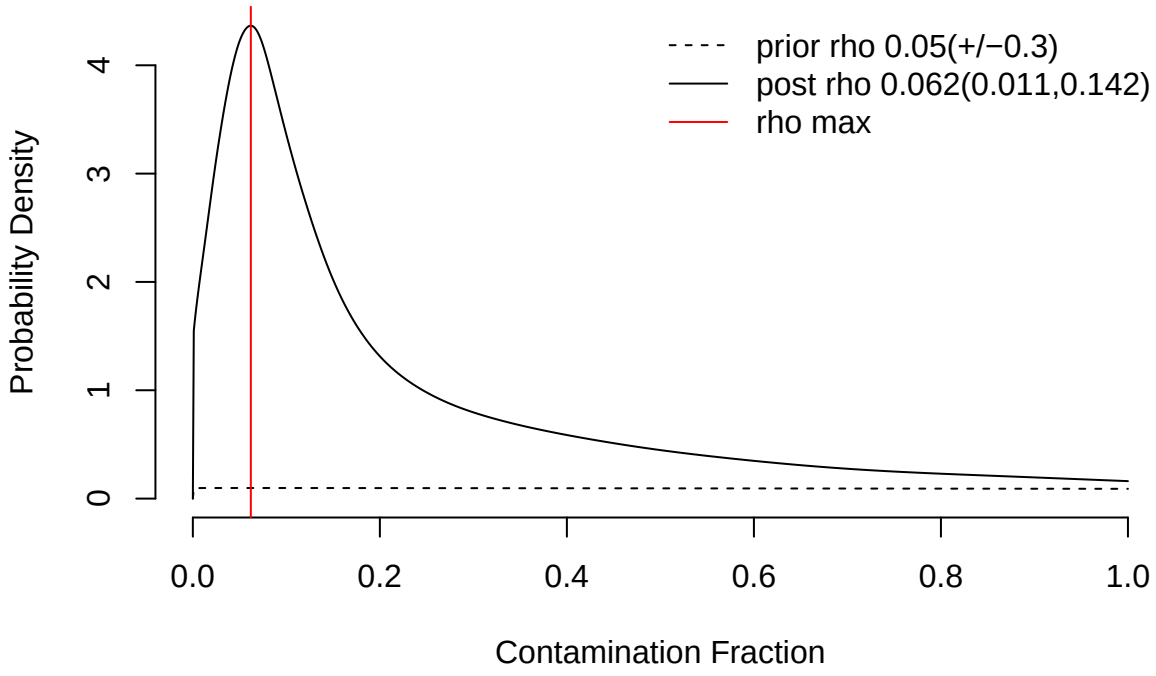
The posterior distribution is calculated using a Poisson likelihood with a gamma distribution prior, parametrised by its mean `priorRho` and standard deviation `priorRhoStdDev`. The dotted line in the above

plot shows the prior distribution. The default parameters have been calibrated to be fairly non-specific with a slight preference towards values of rho in the 0% to 10% range which is most commonly seen for fresh (i.e. not nuclear) single cell experiments.

The default values place only a very weak constraint, as can be seen by setting a uniform prior

```
sc = autoEstCont(sc, priorRhoStdDev = 0.3)
```

```
## 786 genes passed tf-idf cut-off and 401 soup quantile filter.  Taking the top 100.
## Using 854 independent estimates of rho.
## Estimated global rho of 0.06
```



which gives the same answer up to two significant figures. Of course you can break things if you start setting strong, badly motivated priors, so please don't do this.

The manual way

The alternative to the automatic method is to manually specify which sets of genes to use to estimate the contamination fraction. These genes need to be such that we are as certain they will not be expressed in each cell. See the section above on "Genes to estimate the contamination fraction" 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. In general it is up to the user to pick sensible genes, but there are a few things that can be done to aid in this selection process. Firstly, the genes that are the most useful are those expressed most highly in the background. We can check which genes these are by running:

```
head(sc$soupProfile[order(sc$soupProfile$est, decreasing = TRUE), ], n = 20)
```

```
##                est counts
## MALAT1 0.032491616  88000
## B2M    0.019416325  52587
## TMSB4X 0.016583278  44914
## EEF1A1 0.012944217  35058
## RPL21  0.010354856  28045
## RPS27  0.010097877  27349
## RPL13  0.009351309  25327
## RPL13A 0.008643877  23411
## RPL10  0.008105920  21954
## RPLP1  0.007909124  21421
## RPL34  0.007891401  21373
## RPS12  0.007672083  20779
## RPS18  0.007560208  20476
## RPS3A  0.007439842  20150
## RPL32  0.007433934  20134
## RPS6   0.007406612  20060
## RPS27A 0.007243415  19618
## RPS2   0.007188770  19470
## RPL41  0.006997143  18951
## RPL11  0.006277897  17003
```

Unfortunately most of the most highly expressed genes in this case are ubiquitously expressed (*RPL*/*RPS* genes or mitochondrial genes). So we need some further criteria to aid our selection process.

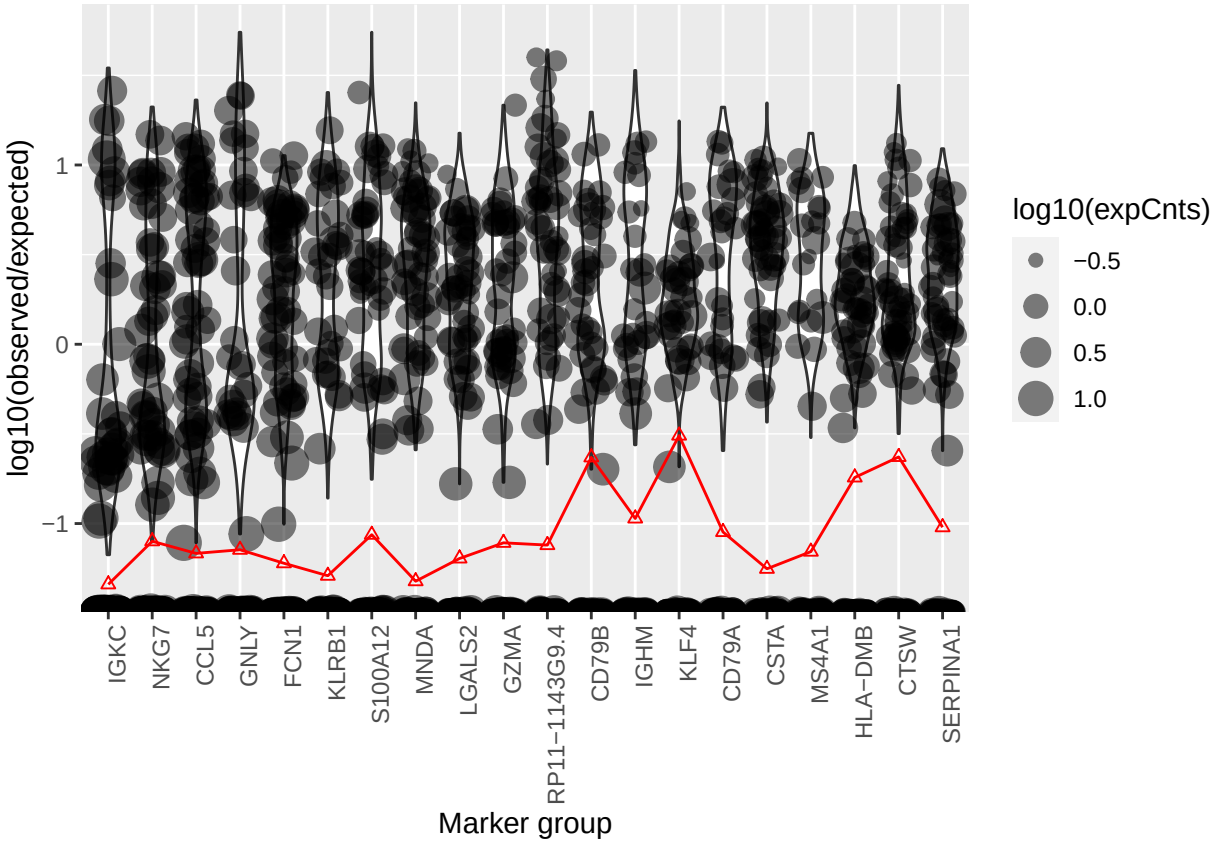
The function `plotMarkerDistribution` is used to visualise the distribution of expression (relative to what would be expected were each cell pure background) across all cells in the data set. When no geneset is provided, the function will try and guess which genes might be useful.

```
plotMarkerDistribution(sc)
```

```
## No gene lists provided, attempting to find and plot cluster marker genes.
```

```
## Found 786 marker genes
```

```
## Warning: Removed 31674 rows containing non-finite values (stat_ydensity).
```



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 the background contamination is calculated. 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. This "guessing" is done using the `quickMarkers` function to find marker genes of each cluster (see "Automatic method" section). As such, it will fail if no clusters have been provided.

Note that this is a heuristic set of genes that is intended to help develop your biological intuition. It absolutely **must not** be used to automatically select a set of genes to estimate the background contamination fraction. For this reason, the function will not return a list of genes. **If you select the top N genes from this list and use those to estimate the contamination, you will over-estimate the contamination fraction!**

Note too that 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.

Looking at this plot, we observe that there are two immunoglobulin genes from the constant region (*IGKC* and *IGHM*) 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 `quickMarkers` 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 non-expressing cells Having decided on a set of genes with which to estimate the contamination, we next need to decide which cells genuinely express these genes and should not be used for estimating the contamination, and which do not and should. This is done as follows,

```
useToEst = estimateNonExpressingCells(sc, nonExpressedGeneList = list(IG = igGenes),
    clusters = FALSE)
```

```
## No clusters found or supplied, using every cell as its own cluster.
```

Which produces a matrix indicating which cells (rows) should use which sets of genes (columns) to estimate the contamination. You will notice that the function returned a warning about cluster information not being provided. As discussed above, SoupX tries to be conservative and prevents estimation both from cells with high expression of a gene set (`igGenes` in this case) and any cell that falls in the same cluster. When no clustering information is given, it cannot do this so defaults to just excluding those cells that are obviously not suitable. We can visualise which cells have been marked to use for estimation,

```
plotMarkerMap(sc, geneSet = igGenes, useToEst = useToEst)
```

You'll notice that above we set `clusters=FALSE` which stops SoupX from using clustering information. We provided this information earlier when we ran `setClusters`. Let's see how things change if we let clustering information be used.

```
useToEst = estimateNonExpressingCells(sc, nonExpressedGeneList = list(IG = igGenes))
plotMarkerMap(sc, geneSet = igGenes, useToEst = useToEst)
```

As you can see the set of cells to be used for estimation with the `igGenes` set has decreased. In this case it makes not much difference, but in general it is better to provide clustering and be conservative.

It is worth noting one final thing about the specification of `nonExpressedGeneList`. It seems odd that we have specified `nonExpressedGeneList = list(IG=igGenes)` instead of just `nonExpressedGeneList = igGenes`. This is because `nonExpressedGeneList` expects sets of genes that are biologically related and expected to be present or not present as a set (e.g. IG genes, HB genes).

Calculating the contamination fraction At this point all the hard work has been done. To estimate the contamination fraction you need only pass your set of genes and which cells in which to use those sets of genes to `calculateContaminationFraction`.

```
sc = calculateContaminationFraction(sc, list(IG = igGenes), useToEst = useToEst)
```

```
## Estimated global contamination fraction of 7.69%
```

This function will modify the `metaData` table of `sc` object to add a table giving the contamination fraction estimate. This approach gives a contamination fraction very close to the automatic method.

```
head(sc$metaData)
```

```
##                   nUMIs clusters        RD1        RD2        rho     rhoLow
## GCGAGAAGTTCTGGTA  6569        2  17.189524 -1.0840559 0.07692854 0.07201939
## TGAGCCGAGACAGGCT  3594        0 -19.479848  0.2849510 0.07692854 0.07201939
## TATCTCAAGCTGGAAC  7152        2  22.135188  0.5986279 0.07692854 0.07201939
## TCAGGTATCACCCTCA  3334        5  -2.778464 -3.0432322 0.07692854 0.07201939
## CACAGTAGTATCACCA  3077        1  13.246665 12.7795112 0.07692854 0.07201939
## CGTTAGATCGTACGGC  4242        1  16.470599 10.5222708 0.07692854 0.07201939
##                     rhoHigh
```

```
## GCGAGAAGTTCTGGTA 0.08205567
## TGAGCCGAGACAGGCT 0.08205567
## TATCTCAAGCTGGAAC 0.08205567
## TCAGGTATCACCCTCA 0.08205567
## CACAGTAGTATCACCA 0.08205567
## CGTTAGATCGTACGGC 0.08205567
```

Correcting expression profile

We have now calculated or set the contamination fraction for each cell and would like to use this to remove the contamination from the original count matrix. As with estimating the contamination, this procedure is made much more robust by providing clustering information. This is because there is much more power to separate true expression from contaminating expression when counts are aggregated into clusters. Furthermore, the process of redistributing corrected counts from the cluster level to individual cells automatically corrects for variation in the cell specific contamination rate (see the paper for details).

We have already loaded clustering information into our `sc` object with `setClusters`, which will be used by default. So we can just run.

```
out = adjustCounts(sc)
```

```
## Warning in sparseMatrix(i = out@i[w] + 1, j = out@j[w] + 1, x = out@x[w], :
## 'giveCsparse' has been deprecated; setting 'repr = "T"' for you
```

```
## Expanding counts from 12 clusters to 2170 cells.
```

The recommended mode of operation will produce a non-integer (although still sparse) matrix where the original counts have been corrected for background expression. See the help, code, and paper for details of how this is done.

You should not change the method parameter unless you have a strong reason to do so. When you need integer counts for downstream analyses, setting `roundToInt=TRUE`, stochastically rounds up with probability equal to the fraction part of the number. For example, if a cell has 1.2 corrected counts it will be assigned a value of 1 80% of the time and 2 20% of the time.

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

```
##   HLA-DRA       PF4      CST3      GNLY      PPBP     IGLC2      IGKC       LYZ
## 0.3955175 0.4117647 0.4210526 0.4329897 0.4509804 0.5550239 0.5650624 0.5793991
##    S100A8    S100A9
## 0.5947761 0.6151294
```

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*.

If on the other hand we focus on genes for which there is a quantitative difference,

```
tail(sort(rowSums(sc$toc > out)/rowSums(sc$toc > 0)), n = 20)
```

```
##      S100B      PRMT2     MT-ND1     MT-ND2     MT-CO1     MT-CO2    MT-ATP8
##          1          1          1          1          1          1          1
```

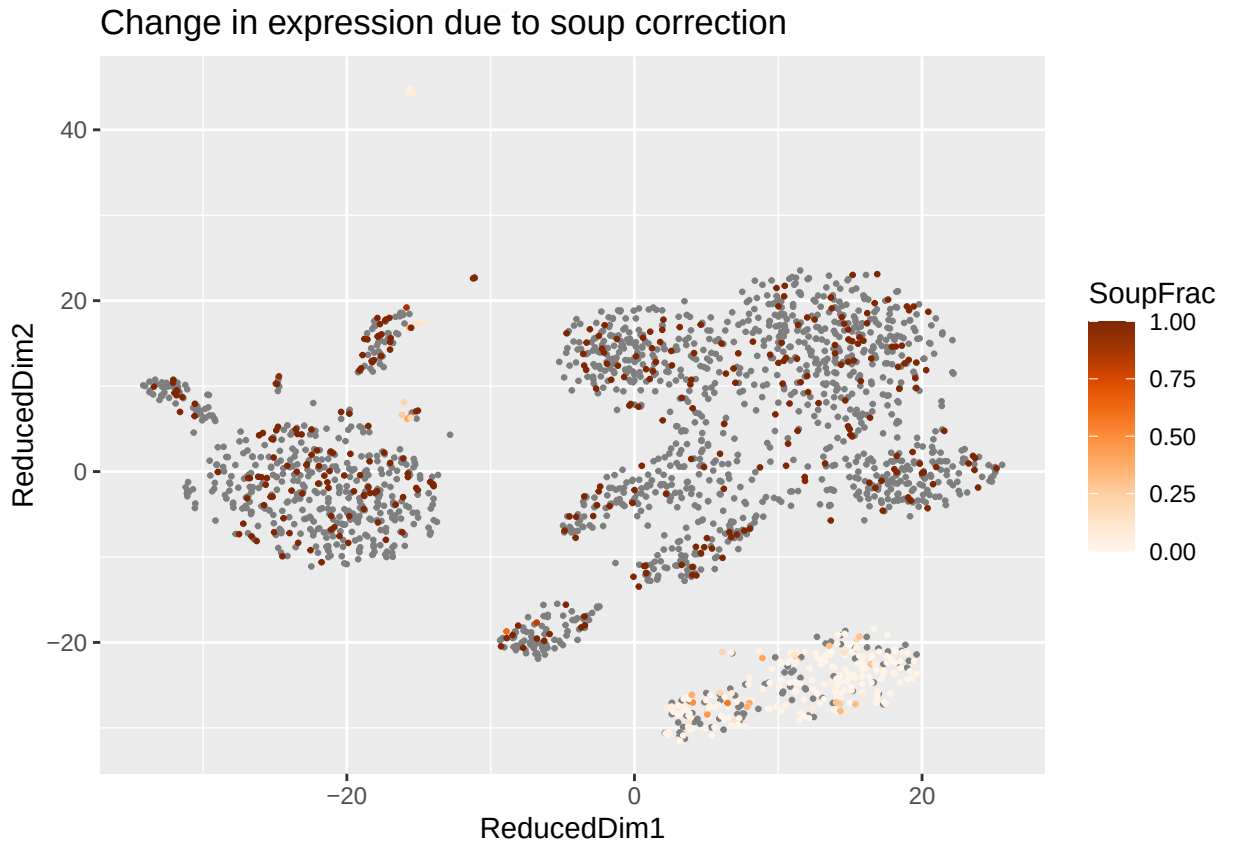
```
##     MT-ATP6      MT-CO3      MT-ND3     MT-ND4L      MT-ND4      MT-ND5      MT-ND6
##           1           1           1           1           1           1           1
##      MT-CYB BX004987.4 AC011043.1 AL592183.1 AC007325.4 AC004556.1
##           1           1           1           1           1           1
```

we find genes associated with metabolism and translation. This is often the case as mitochondrial genes are over represented in the background compared to cells, presumably as a result of the soup being generated from distressed cells.

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 the fraction of expression in each cell that has been deemed to be soup and removed.

```
plotChangeMap(sc, out, "IGKC")
```



which shows us that the expression has been heavily decreased in the areas where it was very surprising to observe it before.

The interpretation of which cells are expressing which genes can change quite dramatically when we correct for soup contamination. You should explore this yourself by plotting a variety of different genes and seeing which change and which do not. For example, you could look at,

```
plotChangeMap(sc, out, "LYZ")
plotChangeMap(sc, out, "CD74")
plotChangeMap(sc, out, "IL32")
plotChangeMap(sc, out, "TRAC")
```

```
plotChangeMap(sc, out, "S100A9")
plotChangeMap(sc, out, "NKG7")
plotChangeMap(sc, out, "GNLY")
plotChangeMap(sc, out, "CD4")
plotChangeMap(sc, out, "CD8A")
```

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.

The corrected matrix can then be used for any downstream analysis in place of the uncorrected raw matrix. If you are using 10X data and would like to save these final counts out in the same format, you can use the DropletUtils `write10xCounts` function like this,

```
DropletUtils:::write10xCounts("./strainedCounts", out)
```

Loading into Seurat

For loading into Seurat or other R packages, there is no need to save the output of SoupX to disk. For a single channel, you can simply run the standard Seurat construction step on the output of `adjustCounts`. That is,

```
library(Seurat)
srat = CreateSeuratObject(out)
```

If you have multiple channels you want to process in SoupX then load into Seurat, you need to create a combined matrix with cells from all channels. Assuming `scs` is a list named by experiment (e.g., `scs = list(Experiment1 = scExp1,Experiment2 = scExp2)`), this can be done by running something like:

```
library(Seurat)
srat = list()
for (nom in names(scs)) {
    # Clean channel named 'nom'
    tmp = adjustCounts(scs[[nom]])
    # Add experiment name to cell barcodes to make them unique
    colnames(tmp) = paste0(nom, "_", colnames(tmp))
    # Store the result
    srat[[nom]] = tmp
}
# Combine all count matricies into one matrix
srat = do.call(cbind, srat)
srat = CreateSeuratObject(srat)
```