# Appendix for "An updated Bioconductor workflow for correlation profiling subcellular proteomics"

Cambridge Centre for Proteomics, University of Cambridge

Charlotte Hutchings, Thomas Krueger, Oliver M. Crook, Laurent Gatto, Kathryn S. Lilley, Lisa M. Bre

## Contents



This Appendix accompanies the paper "An updated Bioconductor workflow for correlation profiling subcellular proteomics" by Hutchings et al., submitted to F1000Research in May 2025. Associated data can be found on Zenodo at doi:10.5281/zenodo.15100485 and also in the corresponding Github repository.

## Identification search with Proteome Discoverer

The raw mass spectrometry (MS) data generated for the use-case was processed using Proteome Discoverer version 3.1. Of note, the raw MS data was re-processed specifically for use in this workflow. As a result, the data generated here differs slightly from that presented by Christopher et al. (2025). Whilst much of the identification and quantification takes place out of sight of the user, Proteome Discoverer incorporates several user-defined search parameters which must be specified according to the sample preparation methods and MS instrumentation used. There is also the option to apply both basic and advanced data filtering parameters during the search. Users must be aware of these parameters as they will directly influence the data output and downstream processing.

Whilst an in-depth discussion of identification searches is outside of the scope of this workflow, a few key parameters are discussed to put the data into context. During sample preparation, the 8 TMT-labelled LOPIT-DC fractions from untreated and xray samples (16 fractions in total) were pooled and separated into 18 fractions in an offline pre-fractionation step using the Acquity UPLC system. Therefore, the three LOPIT-DC experiments generated 54 raw MS files in total. A single MS file (UPLC fraction) was removed from replicate 1 due to low quality, thus leaving 53 raw files to be processed. The 53 resulting raw files were uploaded to Proteome Discoverer 3.1 and processed using one processing workflow per TMTplex (3 in total) and a single multi-consensus workflow. Both the processing and consensus workflow templates are provided in the supplementary materials at Zenodo doi:10.5281/zenodo.15100485.

For the TMT workflow, SequestHT was selected as the search engine and trypsin specified as the enzyme used for proteolytic digestion. Since the digestion was carried out overnight with a 1:20 w/w ratio of

trypsin:protein, digestion was expected to be complete and a low threshold of 2 missed cleavages was allowed. For MS analysis, a Fourier Transform orbitrap with a resolving power of 120,000 m/z was used as the mass analyzer for precursor ion mass, and a linear ion trap was used to measure fragment ion mass. This information determined the thresholds for precursor and fragment mass tolerances, two key parameters for the identification search. The precursor mass tolerance determines which mass range of peptide sequences are considered for each observed spectrum, whilst the fragment mass tolerance specifies how similar the observed and theoretical peptide fragment spectra should be for a match. If these tolerances are too narrow then the correct peptide sequence may be omitted and true positives are lost. However, if thresholds are set too wide then incorrect peptide sequences are considered and false positives arise. Based on the instrumentation used in this experiment, standard mass tolerances of 10 ppm and 0.5 Da were allowed for precursors and fragments, respectively.

In addition to the parameters based on the experimental protocol, we also applied some basic non-specific filtering. We only retained high confidence PSMs from the identification search. Such filtering is necessary because only a fraction of the PSMs output by any given search engine will be genuine matches, or true discoveries, whilst the remainder are incorrect false discoveries. To deal with this problem, PSM confidence level (high, medium or low) is determined via the Proteome Discoverer Percolator node (Käll et al. 2007) which estimates each PSM's false discovery rate (FDR). The raw spectra are searched against the database of interest as well as a decoy database containing randomised peptide sequences, often generated by shuffling or reversing the original peptide sequences. False discovery rate is then defined as the proportion of total PSMs that are matched to the decoy database, and, therefore, are known false discoveries. This is done for all spectra and we considered a PSM to be of 'high confidence' if it had a false discovery rate <1 %, 'medium confidence' if <5 %, and 'low confidence' if the false discovery rate exceeded 5 %. Only PSMs annotated as high confidence were kept.

Whilst the basic filtering steps completed during this identification search could just have easily been carried out in R using the `SummarizedExperiment` and `QFeatures` infrastructure, applying them here saves time later on and reduces the burden of storing large data files. These steps are also relatively standard and non-specific so we do not need to assess the data prior to their implementation. However, Proteome Discoverer also provides the option to carry out more in-depth filtering through the use of parameters such as the SPS Mass Match %, co-isolation interference % and signal-to-noise thresholds. We advise against implementing such filtering at this stage since decisions regarding thresholds will likely be influenced by the quality of data output, as demonstrated in this workflow. Instead, thresholds for the three aforementioned parameters were set to 0 during the identification search.

## Alternative data import settings

How the data is converted into a `QFeatures` object will depend on several factors including, but not limited to, (i) the experimental design e.g. label-free versus multiplexed labelled (e.g. TMT) and, (ii) what third party software was used for the identification search and which mode/workflow was used in the search. For example, the Proteome Discoverer (PD) and MaxQuant software typically output wide table format, whilst the DIA-NN software outputs the data in long table format (see the subsequent DIA section).

As discussed in the main workflow, the use-case A549 dataset was comprised of three replicate experiments, each with two samples/conditions quantified within a TMTpro$^{TM}$ 16plex (eight TMT labels for the unstim sample and eight for xray). All three quantitative TMT datasets were run in a multi-consensus workflow in PD. This resulted in one .txt file with 16 quantitative channels (corresponding to the 16plex TMT channels in each experiment, here 8 per condition/sample), and a column called `File.ID` to identify which raw MS run and, therefore, which TMTplex/replicate the data was derived. This means that each quantitative column contained data for more than one sample and we chose to split the data by rows in order to get one dataset per sample.

Sometimes data will be structured such that each biochemical fraction from each sample/replicate has it's own unique quantitative column. This is often the case for LFQ datasets. Here, instead of splitting the data by rows to get individual samples we need to split by columns. As an example we download a protein

level correlation profiling data that was produced from Schessner et al. (2023). This data was produced using the Dynamic Organellar Maps (DOMs) method. LFQ with DDA was used and the resulting data was processed with MaxQuant. The data was downloaded directly from the "Supplementary Information" section in Supplementary Data 1. The data is provided as a Microsoft Excel Spreadsheet and contains LFQ protein correlation profiles from three replicate HeLa cell samples, denoted `Map1`, `Map2` and `Map3`.

We download the data into our working directory and then use the `readxl` package to read the data into R.

```
## Install package
install.packages("readxl")
library("readxl")
```

```
## Import multi-rep LFQ corrlation profile data (DOMs)
df <- read_excel("41467_2023_41000_MOESM4_ESM.xlsx", sheet = 1)
```

```
## New names:
## * `Protein IDs` -> `Protein IDs...1`
## * `Gene names` -> `Gene names...3`
## * `Protein IDs` -> `Protein IDs...35`
## * `Gene names` -> `Gene names...37`
```

```
## Verify
df %>%
  head()
```

```
## # A tibble: 6 x 42
##   `Protein IDs...1` Compartment `Gene names...3` Normalized Map profi~1 Map1_01K
##   <chr>             <chr>       <chr>            <lgl>                      <dbl>
## 1 Q92692            Plasma mem~ NECTIN2          NA                        0.108
## 2 Q969P0            Plasma mem~ IGSF8            NA                        0.104
## 3 P15151            Plasma mem~ PVR              NA                        0.0890
## 4 P15529            Plasma mem~ CD46             NA                        0.116
## 5 Q9ULF5            undefined   SLC39A10         NA                        0.108
## 6 Q13433            Plasma mem~ SLC39A6          NA                        0.116
## # i abbreviated name: 1: `Normalized Map profiles`
## # i 37 more variables: Map1_03K <dbl>, Map1_06K <dbl>, Map1_12K <dbl>,
## #   Map1_24K <dbl>, Map1_80K <dbl>, Map2_01K <dbl>, Map2_03K <dbl>,
## #   Map2_06K <dbl>, Map2_12K <dbl>, Map2_24K <dbl>, Map2_80K <dbl>,
## #   Map3_01K <dbl>, Map3_03K <dbl>, Map3_06K <dbl>, Map3_12K <dbl>,
## #   Map3_24K <dbl>, Map3_80K <dbl>, `SVM predictions` <lgl>,
## #   `Actin binding proteins` <dbl>, ER <dbl>, Endosome <dbl>, ...
```

The data is read into R as a `tibble` by default. At this stage of the data import it does not matter is the data is a `tibble` or traditional `data.frame`. The `readQFeatures` function accepts both formats. As we do in the main paper we identify the columns that contain the quantitation data by using the `grep` command. In the data the word "Map" identifies the columns of the quantitation data. This data is provided at the protein level.

```
## Use grep to store indices of columns containing "Map" i.e., quant columns
quantID <- grep("^Map", colnames(df))
```

As expected, there are 18 quantitative columns representing three replicates of a correlation profiling experiment with six biochemical fractions. Now that we have identified the columns that contain the quantitation

data we can use the `readQFeatures` function to convert the `tibble` to a `QFeatures` object. At first we do this using a single-set import, meaning that all three replicates will be stored in a single experimental set in our newly generated `QFeatures` object.

```
## Single-set import
qf <- readQFeatures(df, quantCols = quantID, name = "proteins")
```

```
## Checking arguments.

## Loading data as a 'SummarizedExperiment' object.

## Formatting sample annotations (colData).

## Formatting data as a 'QFeatures' object.
```

```
# Verify
qf
```

```
## An instance of class QFeatures containing 1 set(s):
##  [1] proteins: SummarizedExperiment with 7443 rows and 18 columns
```

We can quickly see we have imported the data successfully. If we wanted to analyse and process the data as we have done in the main manuscript on the sample level we can create individual experimental sets for each replicate. Following the section in the main workflow "Subsetting by condition" we subset the data in a similar fashion but identifying which quantitation columns correspond to each sample/replicate.

```
## Identify which columns contain the quant data for replicates 1, 2 and 3
idx1 <- grep("Map1", colnames(qf[[1]]))
idx2 <- grep("Map2", colnames(qf[[1]]))
idx3 <- grep("Map3", colnames(qf[[1]]))
```

Now we create a `list` of `SummarizedExperiments` and add them back to the `QFeatures` object using the `addAssay` function.

```
## Create list of subset SummarizedExperiments
reps <- list(
  "map1" = qf[[1]][, idx1],
  "map2" = qf[[1]][, idx2],
  "map3" = qf[[1]][, idx3]
)

## Add back to the QFeatures object
qf <- addAssay(qf, reps)

## Verify
qf
```

```
## An instance of class QFeatures containing 4 set(s):
##  [1] proteins: SummarizedExperiment with 7443 rows and 18 columns
##  [2] map1: SummarizedExperiment with 7443 rows and 6 columns
##  [3] map2: SummarizedExperiment with 7443 rows and 6 columns
##  [4] map3: SummarizedExperiment with 7443 rows and 6 columns
```

## Using this workflow with MaxQuant data

This workflow was written using proteomics data processed using the Proteome Discoverer software. Nevertheless, the workflow and principles discussed are also applicable to the output of any similar proteomics raw data processing software, including MaxQuant. Below we outline the differences to be aware of when following this workflow using MaxQuant output text files. The code as written will require some minor modifications to work properly with MaxQuant formatted data.

1. The rough equivalent of the PSMs.txt file output by Proteome Discoverer is the evidence.txt file output by MaxQuant.
2. Decoy PSMs (known false discoveries which are used to calculate false discovery rate) are automatically filtered out by Proteome Discoverer, but this is not the case with MaxQuant. Hence when working with MaxQuant outputs it is important to filter out rows with '+' in the `Reverse` column.
3. Equivalent column names and the type of data contained are described here. Ellipses are put where there no equivalent column exists.

In PD the file `PSMs.txt` is equivalent to the MaxQuant `evidence.txt` file. Equivalent column names are shown in the table below:

| Proteome Discoverer | MaxQuant |
|---|---|
| `Abundance` (float) | `Reporter.intensity.corrected` (integer) |
| `Sequence` (string) | `Sequence` (string) |
| `Master.Protein.Accessions` (string) | `Leading.proteins` (string) |
| `Master.Protein.Descriptions` (string) | ... |
| `Contaminants` (string, True or False) | `Potential.contaminant` (string, + or blank) |
| ... | `Reverse` (string, + or blank) |
| `Rank` (integer) | ... |
| `Search.Engine.Rank` (integer) | ... |
| `PSM.Ambiguity` (string) | ... |
| `Number.of.Protein.Groups` (integer) | ... (You might calculate this by counting the number of ; in the Leading.proteins column and adding 1) |
| `Average.Reporter.SN` (float) | ... (You might calculate the average reporter ion intensity and threshold based on that instead) |
| `Isolation.Interference.in.Percent` (float) | `PIF` (float, to get the data in exactly the same format you have to calculate (1 - PIF)/100) |
| `SPS.Mass.Matches.in.Percent` (integer) | ... |

In PD the `Proteins.txt` file is equivalent to MaxQuant `proteinGroups.txt` file and equivalent columns are:

| Proteome Discoverer | MaxQuant |
|---|---|
| `Accession` (string) | `Majority.protein.IDs` (string) |
| `Protein.FDR.Confidence.Combined` (string; High, Medium, or Low) | `Q.value` (float, a Proteome Discoverer protein FDR of 'High' is equivalent to a Q.value $< 0.01$) |

## Using this workflow with DIA-NN data

As outlined in the workflow, MS-based protein correlation profiling data can be generated using a number of methodologies. These methods can differ with respect to (1) biochemical fractionation, (2) peptide

quantitation (label-based or label-free), and (3) mass spectrometry acquisition method (data-dependent acquisition, DDA, or data-independent acquisition, DIA). The general workflow presented is applicable to all subcellular proteomics experiments utilising protein correlation profiling data. However, since the use-case data was a TMT-labelled DDA experiment, some of the code would need to be adapted to be suitable for alternative datasets.

Here, we discuss how to alter the presented workflow for a DIA protein correlation profiling experiment using data processed via the open-source DIA-NN software (Demichev et al. 2019). We would like to note that DIA software is currently undergoing rapid development with regular version updates. Unfortunately, this means that the output file structures are also changing and this workflow will need to be altered accordingly. Further, the analysis of DIA data in general is still an active discussion point within the proteomics community and we envision that additional quality control parameters will be introduced in the future. Users should be aware of this and adapt this DIA workflow accordingly.

## Use-case: DIA-LOP

Using the same differential centrifugation gradient applied in LOPIT-DC (Geladaki et al. 2019), a DIA protein correlation profiling dataset was generated to map the subcellular proteome of U-2 OS cells (McCaskie 2025). Here, 10 biochemical fractions were analysed as independent label-free samples using a DIA method. For each sample, a total of 750 ng of peptide was analysed using using a timsTOF HT mass spectrometer (Bruker Daltonics, Bremen, Germany) coupled with a nanoElute 2 UHPLC system (Bruker Daltonics). The raw MS data is available online through ProteomeXchange via the PRIDE repository under the identifier PXD063082.

Raw MS data was subsequently searched against a Swiss-Prot *Homo sapiens* database (no isoforms, downloaded 20/01/2024) and the Protein Contaminant Libraries for DIA and DDA Proteomics (Frankenfield et al. 2022) using DIA-NN (v1.8.2 beta 27) (Demichev et al. 2019). A maximum of two missed cleavages was allowed, with peptide length restricted to 7–30 amino acids and match between runs enabled to reduce missing identifications. The output file generated by this identification search is provided at Zenodo doi:10.5281/zenodo.15100485.

## Part 1: Data processing and quality control within the `QFeatures` infrastructure

The only part of the workflow which needs to be adapted is 'Part 1: Data processing and quality control within the `QFeatures` infrastructure', that is the part that takes processed MS data and converts it into high quality protein correlation profiles. Given that each step is extensively described in the main workflow, we do not repeat these points here or provide detailed information about each function. Instead, we draw attention to places where the workflow may need to be edited to suit DIA data. Once protein profiles have been generated, Part 2 and Part 3 of the workflow remain the same.

### Importing the data into R

The data we need to get started is stored in the main report `.tsv` output file from DIA-NN. In the code chunk below we use the `read_tsv` function to import the data as a `data.frame`.

```
## Tell R the file location
f <- "diann_report.tsv"

## Import into a dataframe
df <- read_tsv(f)
```

```
## Rows: 880781 Columns: 58
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: "\t"
## chr (13): File.Name, Run, Protein.Group, Protein.Ids, Protein.Names, Genes, ...
## dbl (44): PG.Quantity, PG.Normalised, PG.MaxLFQ, Genes.Quantity, Genes.Norma...
## lgl  (1): Translated.Quality
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## Check the column names of the file
df %>%
  names()
```

```
##  [1] "File.Name"                 "Run"
##  [3] "Protein.Group"             "Protein.Ids"
##  [5] "Protein.Names"             "Genes"
##  [7] "PG.Quantity"               "PG.Normalised"
##  [9] "PG.MaxLFQ"                 "Genes.Quantity"
## [11] "Genes.Normalised"          "Genes.MaxLFQ"
## [13] "Genes.MaxLFQ.Unique"       "Modified.Sequence"
## [15] "Stripped.Sequence"         "Precursor.Id"
## [17] "Precursor.Charge"          "Q.Value"
## [19] "PEP"                       "Global.Q.Value"
## [21] "Protein.Q.Value"           "PG.Q.Value"
## [23] "Global.PG.Q.Value"         "GG.Q.Value"
## [25] "Translated.Q.Value"        "Proteotypic"
## [27] "Precursor.Quantity"        "Precursor.Normalised"
## [29] "Precursor.Translated"      "Translated.Quality"
## [31] "Ms1.Translated"            "Quantity.Quality"
## [33] "RT"                        "RT.Start"
## [35] "RT.Stop"                   "iRT"
## [37] "Predicted.RT"              "Predicted.iRT"
## [39] "First.Protein.Description" "Lib.Q.Value"
## [41] "Lib.PG.Q.Value"            "Ms1.Profile.Corr"
## [43] "Ms1.Area"                  "Evidence"
## [45] "Spectrum.Similarity"       "Averagine"
## [47] "Mass.Evidence"             "CScore"
## [49] "Decoy.Evidence"            "Decoy.CScore"
## [51] "Fragment.Quant.Raw"        "Fragment.Quant.Corrected"
## [53] "Fragment.Correlations"     "MS2.Scan"
## [55] "IM"                        "iIM"
## [57] "Predicted.IM"              "Predicted.iIM"
```

We can see many columns, each containing information about a given precursor peptide in a given sample. These include the precursor peptide sequence, the protein(s) and protein group (PG) to which it has been mapped, and various quality control parameters. For a full explanation of the DIA-NN output we direct users to the DIA-NN GitHub page.

Information about the precursor Id and its peptide sequence can be found in the `Precursor.Id`, `Stripped.Sequence` and `Modified.Sequence` columns, as shown below.

```
df %>%
  dplyr::select(Precursor.Id, Stripped.Sequence, Modified.Sequence)
```

```
## # A tibble: 880,781 x 3
##    Precursor.Id Stripped.Sequence Modified.Sequence
##    <chr>        <chr>             <chr>
##  1 AAAAAAALQAK1 AAAAAAALQAK       AAAAAAALQAK
##  2 AAAAAAALQAK1 AAAAAAALQAK       AAAAAAALQAK
##  3 AAAAAAALQAK1 AAAAAAALQAK       AAAAAAALQAK
##  4 AAAAAAALQAK1 AAAAAAALQAK       AAAAAAALQAK
##  5 AAAAAAALQAK1 AAAAAAALQAK       AAAAAAALQAK
##  6 AAAAAAALQAK1 AAAAAAALQAK       AAAAAAALQAK
##  7 AAAAAAALQAK1 AAAAAAALQAK       AAAAAAALQAK
##  8 AAAAAAALQAK1 AAAAAAALQAK       AAAAAAALQAK
##  9 AAAAAAALQAK1 AAAAAAALQAK       AAAAAAALQAK
## 10 AAAAAAALQAK2 AAAAAAALQAK       AAAAAAALQAK
## # i 880,771 more rows
```

We can see that we have the same `Precursor.Id` across multiple rows. Here, each row contains information about the precursor peptide in a different sample (biochemical fraction). We can also see that we have two `Precursor.Id` entries corresponding to the same peptide sequence, one succeeded by the number 1 and one by the number 2. These numbers indicate the charge state of the peptide, which can also be found in the `Precursor.Charge` column. Hence, data was collected for this peptide sequence under different charge states.

Information about which MS sample (here, biochemical fraction) each precursor was derived from can be found in the `Run` column.

```
## Check annotation of each sample in Run column
df %>%
  dplyr::select(Run) %>%
  unique()
```

```
## # A tibble: 10 x 1
##    Run
##    <chr>
##  1 F9_DIA
##  2 F8_DIA
##  3 F7_DIA
##  4 F6_DIA
##  5 F5_DIA
##  6 F4_DIA
##  7 F3_DIA
##  8 F2_DIA
##  9 F1_DIA
## 10 F10_DIA
```

We can see that we have 10 unique run names, each corresponding to one of our 10 biochemical fractions.

The main difference between this DIA data and the DDA data presented in the main workflow is that the DIA data is output in a long format. This means that instead of having one row per precursor peptide and multiple quantitative columns representing each biochemical fraction (referred to as a 'wide' format), each precursor peptide has multiple rows representing each biochemical fraction it was identified in and one quantititive column. In fact, we have multiple qunatitative columns (`PG.Quantity`, `PG.Normalised`, `PG.MaxLFQ`, `Genes.Quantity`, `Genes.Normalised`, `Genes.MaxLFQ`, `Genes.MaxLFQ.Unique`, `Precursor.Quantity`, `Precursor.Normalised`, `Precursor.Translated`, `Fragment.Quant.Raw`, `Fragment.Quant.Corrected`), but these represent different types of quantitation rather than quantitation across different samples, as discussed below.

The current long data format is not compatible with the `QFeatures` infrastructure used in the main workflow. In order to use `QFeatures` we will need to convert our long data into a standard wide data format. However, when we do this we will lose any column that contains information about a precursor peptide in a given run, including sample-specific quality control information. Therefore, we will first demonstrate how to complete quality control and cleaning of the data in it's long format within a `data.frame` before proceeding to convert the data to a wide format and import into `QFeatures` for compatibility with the rest of the main workflow.

### Create a copy of the raw data

Before we do any data cleaning, we want to create another copy of the data so that we don't overwrite our raw data. We call the new object `df_filtered` as this is the object we will remove data from below.

```
## Create a copy of the raw data
df_filtered <- df
```

### Quality control and data cleaning

As discussed in the main workflow, the quality control parameters available to filter on will depend upon the exact experimental design and third party software used for database searching. As mentioned above, we expect additional DIA quality control parameters to be introduced in the near future as this field continues to move forwards.

Here, we remove:

1. Precursor peptides corresponding to contaminant proteins
2. Precursor peptides which lack a master protein accession
3. Precursor peptides which are not unique to a single protein within a single protein group

**Removal of contaminant peptides** As in the database search described above for the DDA use-case data, the DIA-NN search of this DIA data included a database containing common proteomics contaminants. Specifically, the Protein Contaminant Libraries for DIA and DDA Proteomics Frankenfield et al. (2022). Here, we import this `.fasta` file using the `fasta.index` function from the `Biostrings` package (Pages et al. 2022), extract the protein accessions of contaminants, and then manually search for these accessions in the `Protein.Group` column of our DIA-NN output using a modified version of a function presented in Hutchings et al. (2023).

```
## Load Hao group .fasta file used in search
cont_fasta <- "220813_universal_protein_contaminants_Haogroup_modified.fasta"
conts <- Biostrings::fasta.index(cont_fasta, seqtype = "AA")

## Extract only the protein accessions (not Cont_ at the start)
cont_acc <- regexpr("(?<=\\_).*?(?=\\|)", conts$desc, perl = TRUE) %>%
  regmatches(conts$desc, .)

## Define function to find contaminants
find_cont <- function(object, cont_acc) {
  cont_indices <- c()
  for (i in 1:length(cont_acc)) {
    cont_protein <- cont_acc[i]
    cont_present <- grep(cont_protein, object$Protein.Group)
    output <- c(cont_present)
    cont_indices <- append(cont_indices, output)
```

```
  }
  cont_indices_all <- cont_indices
}

## Store row indices of entries matched to a contaminant-containing protein group
conts <- find_cont(df_filtered, cont_acc)
```

We remove any rows (precursor peptides) with a contaminant protein in the `Protein.Groups` column.

```
## If we find contaminants, remove these rows from the data
if (length(conts) > 0)
  df_filtered <- df_filtered[-conts, ]

## Check dimensions
dim(df_filtered)
```

```
## [1] 873631     58
```

Previously we had 880781 precursor entries in the data and after removal of contaminant proteins we are left with 873631 precursor rows.

**Removal of peptides with no protein assignment** Since the analysis will be carried out at the protein-level, we will need to aggregate precursor peptides into their corresponding proteins. Precursor entries which lack protein assignment cannot be used and must be removed.

Let's see if we have any such precursors in the dataset.

```
## Check for missing values in Protein.Ids and Protein.Group
df_filtered %>%
  pull(Protein.Ids) %>%
  anyNA()
```

```
## [1] FALSE
```

```
df_filtered %>%
  pull(Protein.Group) %>%
  anyNA()
```

```
## [1] FALSE
```

This data does not contain any precursor peptides without a protein and protein group assignment, so we don't have to remove any here.

**Removal of shared peptides** As well as removing precursors without a protein assignment, we also wish to remove precursor peptides which are assigned to multiple proteins. Although most precursor peptides have a single accession in the `Protein.Ids` column, there are some precursors for which this column contains multiple accessions separated by a `;`. Since the output does not contain a column to specifically tell us how many proteins a peptide is mapped to, we take advantage of the `;` and remove all rows (precursors) with a `;` in the `Protein.Ids` column.

```
## Remove precursors corresponding to multiple proteins
df_filtered <- df_filtered %>%
  filter(grepl(";", Protein.Ids) == FALSE)

## Check dimensions
dim(df_filtered)
```

```
## [1] 833370     58
```

**Controlling false discovery rate** The final step in data cleaning is to account for the false discovery rate (FDR). The output from DIA-NN contains information about q-values at different data levels. As discussed in the main workflow, FDR should be controlled at the highest data level to minimise the impact of FDR inflation during data aggregation. Here, the highest data level corresponds to protein groups. Therefore, we set a 1% FDR on `Lib.PG.Q.Value` and `PG.Q.Value`, as well as the global experiment-wide `Q.Value`.

```
## Manual FDR filtering - 0.01% threshold
df_filtered <- df_filtered %>%
  filter(Lib.PG.Q.Value < 0.01 & PG.Q.Value < 0.01 & Q.Value < 0.01)

## Check dimensions
dim(df_filtered)
```

```
## [1] 826051     58
```

### Convert long data format into wide data

Now that we have done all of the quality control filtering which requires run- or sample-specific information, we can re-organise our data into a wide format compatible with `QFeatures`. Unfortunately, this currently requires us to do some data wrangling. As DIA data becomes more widely used, we expect new functions to be created to make this process easier and more streamlined. Currently this process requires us to:

1. Save mappings between precursor peptides and protein-level information
2. Create a quantitation matrix in which each row is a precursor peptide and each column is a sample/fraction
3. Join the peptide and protein information with the wide quantitative data

**Create mappings between precursor peptide and allocated proteins** The first thing to do is create a mapping between precursor peptides and their assigned protein. We select the columns that we require for the rest of the workflow: `Precursor.Id`, `Stripped.Sequence`, `Protein.Ids`, `Protein.Group`, `Protein.Names` and `First.Protein.Descriptions`. We remove any rows representing duplicated `Precursor.Ids` as we only need to store each mapping once.

```
## Extract key information about each precursor peptide
prot_mappings <- df_filtered %>%
  dplyr::select(Precursor.Id, Stripped.Sequence, Protein.Ids,
         Protein.Group, Protein.Names, First.Protein.Description) %>%
  filter(duplicated(Precursor.Id) == FALSE)

## Check
prot_mappings %>%
  head()
```

```
## # A tibble: 6 x 6
##   Precursor.Id         Stripped.Sequence Protein.Ids Protein.Group Protein.Names
##   <chr>                <chr>             <chr>       <chr>         <chr>
## 1 AAAAAAALQAK1         AAAAAAALQAK       P36578      P36578        RL4_HUMAN
## 2 AAAAAAALQAK2         AAAAAAALQAK       P36578      P36578        RL4_HUMAN
## 3 AAAAAATAPPSPGPAQPGP~ AAAAAATAPPSPGPAQ~ Q6SPF0      Q6SPF0        SAMD1_HUMAN
## 4 AAAAAATVLLR2         AAAAAATVLLR       O60779      O60779        S19A2_HUMAN
## 5 AAAAADLANR2          AAAAADLANR        O76031      O76031        CLPX_HUMAN
## 6 AAAAALSGSPPQTEKPTHY~ AAAAALSGSPPQTEKP~ Q9H9P5      Q9H9P5        UNKL_HUMAN
## # i 1 more variable: First.Protein.Description <chr>
```

**Create a wide quantitative matrix** Next, we create a new column called `Fraction` to indicate which biochemical fraction each precursor was derived from. This can be extracted from the file names in the `Run` column. For users with multiple replicates and/or conditions, this column can be altered to contain the fraction and experiment such that each column has a unique column name and can be easily subset. We then select the `Precursor.Id`, `Precursor.Quantity` and newly generated `Fraction` columns before pivoting wide using the `pivot_wide` function. We pass the `names_from = Fraction` and `values_from = Precursor.Quantity` arguments to create one column per unique value in our `Fraction` column and fill this column with the `Precursor.Quantity` per `Precursor.Id`.

**Why use `Precursor.Quantity`?** As outlined above, the DIA-NN output contains a number of quantitative columns. Since our analysis started from the precursor level, we will use precursor-level quantitation and aggregate upwards to protein-level. Whilst DIA-NN provides the `Precursor.Normalised` column containing quantitative data pre-normalised using the MaxLFQ algorithm, this normalisation step makes the assumption that the majority of the proteome does not change between any two samples (Cox et al. 2014). Since biochemical fractions derived from a protein correlation profiling experiment are expected to be relatively different from each other in terms of their composition, our data does not meet the assumptions required. Hence, we use the `Precursor.Quantity` column which contains pre-normalised quantitation data.

```
quant_wide <- df_filtered %>%
  mutate(Fraction = str_extract(Run, "[^_]+")) %>%
  dplyr::select(Precursor.Id, Precursor.Quantity, Fraction) %>%
  pivot_wider(names_from = Fraction, values_from = Precursor.Quantity)

## Check
quant_wide %>%
  head()
```

```
## # A tibble: 6 x 11
##   Precursor.Id          F9      F8      F7     F6      F5      F4      F3     F2
##   <chr>              <dbl>   <dbl>   <dbl>  <dbl>   <dbl>   <dbl>   <dbl>  <dbl>
## 1 AAAAAAALQAK1       6345.  11555.   3698.  1597.    817.   2456.   1391. 2.13e3
## 2 AAAAAAALQAK2     124514  513744  285341  33384. 155713  549252  704940  5.45e5
## 3 AAAAAATAPPSPGPA~  10601.   6086.   4729.  5701.   2180.   4110.   3626. 3.73e3
## 4 AAAAAATVLLR2         NA      NA      NA     NA   13307.   6521.   5844. 3.18e3
## 5 AAAAADLANR2          NA      NA      NA     NA      NA      NA      NA  9.69e4
## 6 AAAAALSGSPPQTEK~   5243.   8027.   4182.  6154.   4494.   3896.   2945. 2.05e3
## # i 2 more variables: F1 <dbl>, F10 <dbl>
```

Currently, the quantitative columns are not in the order of our biochemical fractionation gradient. To make downstream visualisation easier, we here re-order the quantitative columns. This can be done as shown below.

```
## Re-order quantitative columns
col_order <- c(paste0("F", 1:10))
quant_wide <- quant_wide[, c("Precursor.Id", col_order)]

## Verify
quant_wide %>%
  names()
```

```
##  [1] "Precursor.Id" "F1"           "F2"           "F3"           "F4"
##  [6] "F5"           "F6"           "F7"           "F8"           "F9"
## [11] "F10"
```

**Join wide precursor quantitative matrix with protein mappings** Finally, we join the quantitative matrix with the information we stored about each precursor in our `prot_mappings` object.

```
df_filtered_wide <- left_join(prot_mappings, quant_wide, by = "Precursor.Id")

## Check
df_filtered_wide %>%
  names()
```

```
##  [1] "Precursor.Id"              "Stripped.Sequence"
##  [3] "Protein.Ids"               "Protein.Group"
##  [5] "Protein.Names"             "First.Protein.Description"
##  [7] "F1"                        "F2"
##  [9] "F3"                        "F4"
## [11] "F5"                        "F6"
## [13] "F7"                        "F8"
## [15] "F9"                        "F10"
```

Now we have a single `data.frame` object which is similar to the one we imported in step 1 of the main workflow. We have one row per precursor and one quantitative column per biochemical fraction.

### Importing the data into a `QFeatures` object

Now that we have a suitable data format, we import the data into a `QFeatures` object, as outlined in the main workflow. We pass our `df_filtered_wide` object and the indices of our quantitative columns.

```
## Create QFeatures object
qf <- readQFeatures(assayData = df_filtered_wide,
                    quantCols = 7:16,
                    name = "precursors")

## Verify
qf
```

We can see that we have a `QFeatures` object with a single experimental set which we named "precursors" to reflect the stage of data analysis. The data contains 128713 rows (precursors) and 10 columns (biochemical fractions).

Users may now join the main workflow at the "Management of missing data" section. We advise users to read the main workflow for all details on the `QFeatures` infrastructure and functions. For completeness, we demonstrate the remainder of Part I of the workflow below.

### Management of missing data

Proteomics datasets contain missing values for a number of reasons, and we refer users to the main workflow for discussion on this. Importantly, it is necessary to explore the frequency and distribution of missing values in any dataset before deciding how best to deal with these.

First, we check how missing values are denoted in the DIA-NN output. We check for the presence of zero and NA values.

```
## Check data for zero values
qf[["precursors"]] %>%
  assay() %>%
  longFormat() %>%
  filter(value == 0) %>%
  nrow()
```

```
## [1] 0
```

```
## Check for NA values
qf[["precursors"]] %>%
  assay() %>%
  longFormat() %>%
  filter(is.na(value) == TRUE) %>%
  nrow()
```

```
## [1] 461079
```

We can see that the data does not contain any zero values but does have a lot of NAs. Since our missing data is denoted as NA values, we can continue to use the `nNA` function from the `QFeatures` infrastructure.

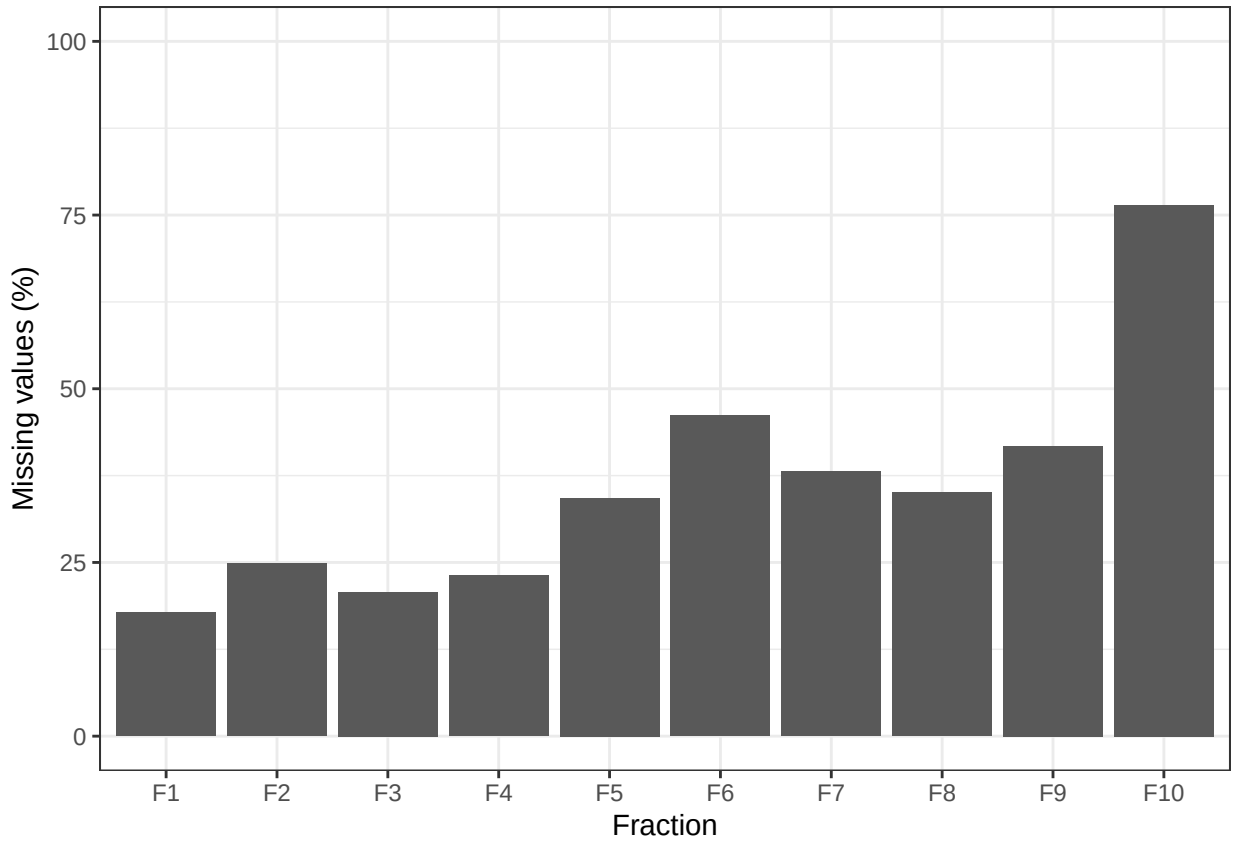
```
## Check frequency and distribution of MVs
nNA(qf, i = "precursors")
```

```
## $nNA
## DataFrame with 1 row and 3 columns
##         assay       nNA       pNA
##   <character> <integer> <numeric>
## 1  precursors    461079  0.358223
##
## $nNArows
## DataFrame with 128713 rows and 4 columns
##              assay        name       nNA       pNA
##        <character> <character> <integer> <numeric>
## 1       precursors           1         1       0.1
## 2       precursors           2         0       0.0
## 3       precursors           3         1       0.1
## 4       precursors           4         5       0.5
## 5       precursors           5         8       0.8
## ...            ...         ...       ...       ...
## 128709  precursors      128709         5       0.5
## 128710  precursors      128710         4       0.4
## 128711  precursors      128711         6       0.6
## 128712  precursors      128712         3       0.3
```

```
## 128713  precursors      128713         1       0.1
##
## $nNAcols
## DataFrame with 10 rows and 4 columns
##          assay        name       nNA       pNA
##    <character> <character> <integer> <numeric>
## 1   precursors          F1     22903  0.177939
## 2   precursors          F2     31927  0.248048
## 3   precursors          F3     26677  0.207260
## 4   precursors          F4     29778  0.231352
## 5   precursors          F5     44090  0.342545
## 6   precursors          F6     59411  0.461577
## 7   precursors          F7     49085  0.381352
## 8   precursors          F8     45111  0.350477
## 9   precursors          F9     53721  0.417370
## 10  precursors         F10     98376  0.764305
```

We see that we have 461079 missing values, which represent 0.36 of the entire dataset. This is a much higher proportion of missing data than seen for the DDA TMT use-case presented in the main workflow. Let's plot the percentage of missing values per quantitative sample (i.e., per biochemical fraction).

```
## Plot the data per biochemical fraction
nNA(qf[["precursors"]])$nNAcols %>%
  as_tibble() %>%
  mutate(name = factor(name, levels = c(paste0("F", 1:10)))) %>%
  ggplot(aes(x = name, y = (pNA * 100))) +
  geom_col() +
  ylim(c(0, 100)) +
  labs(x = "Fraction", y = "Missing values (%)") +
  theme_bw()
```




We do still see the same general trend observed for the DDA TMT LOPIT-DC data with missing values showing an increase in frequency towards the end of the biochemical gradient. However, the overall proportion of missing data is much higher. As discussed in the main workflow and our previous sister workflow (Hutchings et al. 2023), it is standard to set a threshold on the number of missing values per precursor across the samples, here biochemical fractions.

```
## Missing values per precursor
nNA(qf, i = "precursors")$nNArows$nNA %>% table()
```

```
## .
##     0     1     2     3     4     5     6     7     8     9
## 18195 23838 14975 11765 10305 12395 12813 10769  6382  7276
```

If we applied our standard LOPIT-DC TMT workflow, we would remove any features (here precursors) with more than 2 MVs across the LOPIT gradient. Doing this here would remove a huge proportion of the dataset and is not reasonable.

```
## Number proteins currently
qf[["precursors"]] %>%
  rowData() %>%
  as_tibble() %>%
  pull(Protein.Group) %>%
  unique() %>%
  length()
```

```
## [1] 9021
```

```
## Number of proteins after removing precursors with > 2 MVs
qf[["precursors"]] %>%
  filterNA(pNA = 2/10) %>%
  rowData() %>%
  as_tibble() %>%
  pull(Protein.Group) %>%
  unique() %>%
  length()
```

```
## [1] 6247
```

This is one example of how different methods can result in data structures which require tailored analysis pipelines. In correlation profiling experiments we expect that subcellular compartments and their constituent proteins should be separated across biochemical fractions. In some cases, as is often observed for the mitochondrion, organelles may only be found in a few of the biochemical fractions. Consequently, we would expect the proteins in these compartments to have biologically meaningful missing values in the remaining fractions. Fortunately, when using a DDA TMT approach, as presented in the main workflow, TMT co-isolation interference results in very low abundance values even where there should be a missing value. This means that after row sum normalisation we end up with complete protein profiles without the need to impute much of the data. In the case of DIA correlation profiling, however, missing values present as truly missing. This means that we need to reduce the missing value threshold at which we remove precursors and carefully implement an imputation method.

The imputation of proteomics data is a very complex topic and there is no one-size-fits-all solution. However, it is important to consider which imputation method is appropriate as this will have a profound effect on the resolution of any downstream subcellular maps. We ultimately leave this decision to the user and would advise trialing several approaches. For completeness, we here demonstrate how to first implement an initial filter on the number of missing values per precursor by removing those with >=80% MVs before applying a mixed imputation approach on the remaining missing values.

**Removing precursors with >= 80% missing values** In order to prevent the loss of meaningful biological data, we here impose a very lenient threshold and only remove precursors with 8 or more missing values across the 10 biochemical fractions.

```
## Filter to allow a maximum of 7/10 MVs
qf <- filterNA(qf,
               i = "precursors",
               pNA = 7/10)

## Check we have a maximum of 7 MVs per precursor
nNA(qf[["precursors"]])$nNArows$nNA %>%
  max()
```

```
## [1] 7
```

We can see that all precursors now have a maximum of 7 missing values across the DC gradient.

**Mixed imputation of remaining missing values** We next show how to implement a mixed imputation approach using the `impute` function within `QFeatures`. Mixed imputation can be useful when dealing with a mixture of missing at random (MAR) and missing not at random (MNAR) values. Determining which missing values in the dataset belong to each of these categories is extremely challenging. As an example, it seems

reasonable to assume that proteins which have tight/narrow abundance distributions across our biochemical gradient would have a larger number of missing values across each precursor as they are genuinly missing from the other fractions, as is the case for mitochondrial proteins in this dataset. These are biologically relevant missing not at random (MNAR) values. By contrast MAR values arising for technical reasons should be evenly distributed across our dataset and appear at a low frequency for any given precursor. Therefore, we make a distinction that precursors with <= 2 missing values are MAR, whilst those with >= 2 are MNAR. We note that this strategy is used for demonstration purposes only and users should test different approaches on their own dataset.

We first create a vector with one `TRUE` or `FALSE` value per row (precursor). This will be used to indicate whether missing values within that row should be treated as MAR or MNAR (where MAR = `TRUE` and MNAR = `FALSE`).

```
## Create logical vector indicating whether precursor is MAR (2 or less MVs)
no_mvs <- rowSums(is.na(assay(qf[["precursors"]])))
mar_mvs <- ifelse(no_mvs <= 2, TRUE, FALSE)
```

We now use the `impute` function, specify `method = "mixed"` and pass the vector we have created to the `randna` argument. Finally, we specify the methods we wish to use for each category of missingness. Here, we use `mnar = "min"` and `mar = "knn"`.

```
## Carry out mixed imputation
qf <- QFeatures::impute(qf,
                        i = "precursors",
                        name = "precursors_imputed",
                        method = "mixed",
                        randna = mar_mvs,
                        mnar = "min",
                        mar = "knn")
```

Let's double check that we don't have any missing values remaining.

```
## Verify
nNA(qf[["precursors_imputed"]])$nNA
```

```
## DataFrame with 1 row and 2 columns
##         nNA       pNA
##   <integer> <numeric>
## 1         0         0
```

### Normalisation

To generate protein correlation profiles from our raw abundance data we need to apply row sum normalisation. This step will scale all values in each row (precursor) to between 0 and 1, such that the row sum is 1. Essentially, all values become a proportion of the total abundance of that precursor found in each fraction of the gradient.

In the main workflow the normalisation step was completed prior to k-NN imputation as we have previously found k-NN to work better once all data has been scaled into the same space. However, given the high proportion of missing values in DIA-LOP data, row sum normalisation prior to imputation would result in as few as three values contributing to the row sum (which will total 1). Thus, after imputing the data we would end up with profiles with row sums much greater than 1. Therefore, we here imputed and now normalise. We pass our `precursors_imputed` assay to the `normalize` function and specify `method = "sum"`.

```
## Apply normalisation to imputed precursors
qf <- normalize(qf,
                i = "precursors_imputed",
                name = "precursors_norm",
                method = "sum")

## verify
qf
```

```
## An instance of class QFeatures containing 3 set(s):
##  [1] precursors: SummarizedExperiment with 115055 rows and 10 columns
##  [2] precursors_imputed: SummarizedExperiment with 115055 rows and 10 columns
##  [3] precursors_norm: SummarizedExperiment with 115055 rows and 10 columns
```

Let's take a look at what this normalisation has done to the data.

```
## Before nornmalisation
qf[["precursors_imputed"]] %>%
  assay() %>%
  head()
```

```
##          F1        F2        F3        F4         F5         F6          F7
## 1   1067.07   2125.12   1391.09   2456.14    817.044  1597.0900   3698.2100
## 2 324031.00 544858.00 704940.00 549252.00 155713.000 33383.9000 285341.0000
## 3  22181.80   3727.15   3626.16   4110.18   2180.090  5701.2400   4729.1900
## 4   4823.14   3182.10   5844.16   6521.19  13307.400    42.0019     42.0019
## 6   2607.08   2052.06   2945.09   3896.11   4494.120  6154.1700   4182.1200
## 7   7105.28   1418.06   2448.10   2124.09   2952.110  3470.1400   5653.2200
##            F8          F9         F10
## 1  11554.6000   6345.3900   2025.0650
## 2 513744.0000 124514.0000 202029.0000
## 3   6086.2500  10601.4000  51827.0378
## 4     42.0019     42.0019     42.0019
## 6   8027.2400   5243.1400   2658.3148
## 7  11110.4000   9608.3700   3325.0750
```

```
## After normalisation
qf[["precursors_norm"]] %>%
  assay() %>%
  head()
```

```
##           F1         F2         F3         F4         F5          F6
## 1 0.03226036 0.06424802 0.04205634 0.07425563 0.02470141 0.048284268
## 2 0.09425518 0.15849004 0.20505521 0.15976818 0.04529430 0.009710816
## 3 0.19327092 0.03247481 0.03159488 0.03581216 0.01899521 0.049675135
## 4 0.14232590 0.09390050 0.17245515 0.19243361 0.39268768 0.001239433
## 6 0.06169224 0.04855861 0.06969069 0.09219501 0.10634593 0.145628274
## 7 0.14437270 0.02881366 0.04974312 0.04315954 0.05998414 0.070510026
##            F7          F8          F9         F10
## 1 0.111806701 0.349326215 0.191837976 0.061223088
## 2 0.083000905 0.149439502 0.036219031 0.058766843
## 3 0.041205624 0.053029743 0.092370428 0.451571082
```

```
## 4 0.001239433 0.001239433 0.001239433 0.001239433
## 6 0.098962966 0.189951384 0.124070253 0.062904630
## 7 0.114868187 0.225753022 0.195233166 0.067562440
```

The quantitative data has been transformed into correlation profiles, each with a sum of 1. This scales all data into the same space so that we can use the shape rather than the intensity of profiles for downstream machine learning.

#### Aggregation

Finally, we aggregate our precursor-level profiles to protein correlation profiles. To do so, we make use of the `aggregateFeatures` function and group all precursors with the same value in the `Protein.Group` column. We take the median of their normalised quantitation values as the protein-level value, as specified by `fun = matrixStats::colMedians`.

```
## Aggregate from precursor to protein
qf <- aggregateFeatures(qf,
                        i = "precursors_norm",
                        fun = matrixStats::colMedians,
                        name = "prots",
                        fcol = "Protein.Group")

## Verify
qf
```

```
## An instance of class QFeatures containing 4 set(s):
##  [1] precursors: SummarizedExperiment with 115055 rows and 10 columns
##  [2] precursors_imputed: SummarizedExperiment with 115055 rows and 10 columns
##  [3] precursors_norm: SummarizedExperiment with 115055 rows and 10 columns
##  [4] prots: SummarizedExperiment with 8668 rows and 10 columns
```

## Using this workflow to generate peptide-level subcellular maps

As discussed at the start of Part 2, peptide-level data can also be generated using the `QFeatures` infrastructure and this data can be used to create a peptide-level subcellular maps. To do so, the peptide-level data can be extracted into an `MSnSet` and `pRoloc` functions used in the same way as presented in the main workflow for the protein-level `MSnSet`. Here, we briefly demonstrate how this could be done.

### Import QFeatures object from main workflow

First we will load the `QFeatures` object generated by Part 1 of the main workflow.

```
## Load QFeatures object from the main workflow
load("qf.rda")

## Check current experimental sets
experiments(qf)
```

```
## ExperimentList class object of length 32:
##  [1] psms_raw_rep1: SummarizedExperiment with 115302 rows and 16 columns
```

```
##   [2] psms_raw_rep2: SummarizedExperiment with 135169 rows and 16 columns
##   [3] psms_raw_rep3: SummarizedExperiment with 120343 rows and 16 columns
##   [4] psms_filtered_rep1: SummarizedExperiment with 79117 rows and 16 columns
##   [5] psms_filtered_rep2: SummarizedExperiment with 90226 rows and 16 columns
##   [6] psms_filtered_rep3: SummarizedExperiment with 80446 rows and 16 columns
##   [7] psms_rep1_unstim: SummarizedExperiment with 78864 rows and 8 columns
##   [8] psms_rep2_unstim: SummarizedExperiment with 90078 rows and 8 columns
##   [9] psms_rep3_unstim: SummarizedExperiment with 80302 rows and 8 columns
##   [10] psms_rep1_xray: SummarizedExperiment with 78794 rows and 8 columns
##   [11] psms_rep2_xray: SummarizedExperiment with 89910 rows and 8 columns
##   [12] psms_rep3_xray: SummarizedExperiment with 80099 rows and 8 columns
##   [13] psms_rep1_unstim_norm: SummarizedExperiment with 78864 rows and 8 columns
##   [14] psms_rep2_unstim_norm: SummarizedExperiment with 90078 rows and 8 columns
##   [15] psms_rep3_unstim_norm: SummarizedExperiment with 80302 rows and 8 columns
##   [16] psms_rep1_xray_norm: SummarizedExperiment with 78794 rows and 8 columns
##   [17] psms_rep2_xray_norm: SummarizedExperiment with 89910 rows and 8 columns
##   [18] psms_rep3_xray_norm: SummarizedExperiment with 80099 rows and 8 columns
##   [19] psms_rep1_unstim_imputed: SummarizedExperiment with 78864 rows and 8 columns
##   [20] psms_rep2_unstim_imputed: SummarizedExperiment with 90078 rows and 8 columns
##   [21] psms_rep3_unstim_imputed: SummarizedExperiment with 80302 rows and 8 columns
##   [22] psms_rep1_xray_imputed: SummarizedExperiment with 78794 rows and 8 columns
##   [23] psms_rep2_xray_imputed: SummarizedExperiment with 89910 rows and 8 columns
##   [24] psms_rep3_xray_imputed: SummarizedExperiment with 80099 rows and 8 columns
##   [25] prots_rep1_unstim: SummarizedExperiment with 6446 rows and 8 columns
##   [26] prots_rep2_unstim: SummarizedExperiment with 6689 rows and 8 columns
##   [27] prots_rep3_unstim: SummarizedExperiment with 6375 rows and 8 columns
##   [28] prots_rep1_xray: SummarizedExperiment with 6446 rows and 8 columns
##   [29] prots_rep2_xray: SummarizedExperiment with 6689 rows and 8 columns
##   [30] prots_rep3_xray: SummarizedExperiment with 6374 rows and 8 columns
##   [31] prots_unstim: SummarizedExperiment with 5701 rows and 24 columns
##   [32] prots_xray: SummarizedExperiment with 5700 rows and 24 columns
```

We see that our current `QFeatures` object, `qf`, contains PSM- and protein-level data. This is because the main workflow used the `aggregateFeatures` function to aggregate directly from the PSM to protein level. Hence, we first need to generate peptide-level data.

## Aggregation to peptide level

As demonstrated in Hutchings et al. (2023), it is also possible to aggregate from PSM to peptide level. As in the main workflow, we extract the indices of our imputed PSM datasets and define the names of our new aggregated assays. We then use the `aggregateFeatures` function in the same way as before but pass the `"Sequence"` column to the `fcol` argument instead of `"Master.Protein.Accessions"`. This means that we will aggregate all PSMs with the same peptide sequence. Of note, PD contains both a `"Sequence"` and `"Annotated.Sequence"` column, the latter containing information about peptide modifications. By aggregating using the `"Sequence"` column, we consider only stripped peptide sequences and do not account for differential modifications. Users who wish to map modified and unmodified peptides separately should change the code to aggregate by the `"Annotated.Sequence"` column.

```
## Extract indices of imputed normalised psm assays we wish to aggregate
ind <- grep("imputed", names(qf))

## Define original names vector
names <- c("psms_rep1_unstim", "psms_rep2_unstim", "psms_rep3_unstim",
```

```
          "psms_rep1_xray", "psms_rep2_xray", "psms_rep3_xray")

## Define the names of new peptide-level assays
assay_names <- gsub("psms", "peps", names)

## Aggregate from psm to peptide
for (z in seq_along(ind)) {
  qf <- QFeatures::aggregateFeatures(qf,
                                     i = ind[z],
                                     fun = matrixStats::colMedians,
                                     name = assay_names[z],
                                     fcol = "Sequence")
}


## Verify
qf
```

```
## An instance of class QFeatures containing 38 set(s):
##  [1] psms_raw_rep1: SummarizedExperiment with 115302 rows and 16 columns
##  [2] psms_raw_rep2: SummarizedExperiment with 135169 rows and 16 columns
##  [3] psms_raw_rep3: SummarizedExperiment with 120343 rows and 16 columns
##  ...
##  [36] peps_rep1_xray: SummarizedExperiment with 52608 rows and 8 columns
##  [37] peps_rep2_xray: SummarizedExperiment with 54961 rows and 8 columns
##  [38] peps_rep3_xray: SummarizedExperiment with 50379 rows and 8 columns
```

We can see that we have now generated six new experimental sets with 50,000-55,000 peptides per sample.

### Concatenating peptide datasets for machine learning

As in the main workflow, we can concatenate replicate peptide-level datasets using the `joinAssays` function.

```
## Combine replicates - now each with unique column names
qf <- joinAssays(x = qf,
                 i = c("peps_rep1_unstim",
                       "peps_rep2_unstim",
                       "peps_rep3_unstim"),
                 name = "peps_unstim")

qf <- joinAssays(x = qf,
                 i = c("peps_rep1_xray",
                       "peps_rep2_xray",
                       "peps_rep3_xray"),
                 name = "peps_xray")

## Keep only peptides found across all three replicates
qf <- filterNA(qf, i = "peps_unstim")
qf <- filterNA(qf, i = "peps_xray")

## Verify
qf
```

```
## An instance of class QFeatures containing 40 set(s):
##  [1] psms_raw_rep1: SummarizedExperiment with 115302 rows and 16 columns
##  [2] psms_raw_rep2: SummarizedExperiment with 135169 rows and 16 columns
##  [3] psms_raw_rep3: SummarizedExperiment with 120343 rows and 16 columns
##  ...
##  [38] peps_rep3_xray: SummarizedExperiment with 50379 rows and 8 columns
##  [39] peps_unstim: SummarizedExperiment with 31169 rows and 24 columns
##  [40] peps_xray: SummarizedExperiment with 31114 rows and 24 columns
```

We now have two peptide-level concatenated datasets, one per condition. The dataset `peps_unstim` has 24 quantitation channels and contains 31169 peptides common across the 3 replicates. The dataset `peps_xray` has 31114 peptides across the 3 replicates for the 12hr-XRAY stimulated dataset.

### Extract peptide-level data into MSnSet infrastructure for use in pRoloc

Now that we have generated and stored the peptide-level data within our `QFeatures` object, we can extract the concatenated peptide-level `SummarizedExperiment` and convert this into an `MSnSet`. As in the main workflow, this is demonstrated only for the concatenated unstimulated dataset.

```
## Coerce experimental set of interest into an MSnSet
unstim_msn_pep <- as(object = qf[["peps_unstim"]], Class = "MSnSet")

## Verify
unstim_msn_pep
```

```
## MSnSet (storageMode: lockedEnvironment)
## assayData: 31169 features, 24 samples
##   element names: exprs
## protocolData: none
## phenoData
##   sampleNames: rep1_126 rep1_127N ... rep3_130N (24 total)
##   varLabels: runCol quantCols condition
##   varMetadata: labelDescription
## featureData
##   featureNames: AAAAAAALQAK AAAAAGGK ... YYYQGCASWK (31169 total)
##   fvarLabels: Checked Tags ... Number.of.Protein.Groups (24 total)
##   fvarMetadata: labelDescription
## experimentData: use 'experimentData(object)'
## Annotation:
## - - - Processing information - - -
##  MSnbase version: 2.34.0
```

This `MSnSet` can be used as an input to functions in the `pRoloc` package. However, users should be aware that the code will take longer to run with a peptide-level `MSnSet` as this contains many more features (rows) compared to the protein-level equivalent.

### Continue the rest of the workflow to generate peptide-level map

As mentioned above, the peptide-level `MSnSet` can be used as an input for `pRoloc` functions in the same was as a protein-level `MSnSet`. Therefore, users are referred back to the main workflow for all remaining steps. For demonstration purposes, however, we here show how to add markers to the peptide data and generate an initial subcellular map.

Markers can be added using the `addMarkers` function, as demonstrated in the main workflow. Currently, however, the peptide-level dataset feature names are peptide sequences.

```
## Check MSnSet feature names
featureNames(unstim_msn_pep) %>%
  head()
```

```
## [1] "AAAAAAALQAK"
## [2] "AAAAAGGK"
## [3] "AAAAATVVPPMVGGPPFVGPVGFGPGDR"
## [4] "AAAAAWEEPSSGNGTAR"
## [5] "AAAACLDK"
## [6] "AAAAPAATTATPPPEGAPPQPGVHNLPVPTLFGTVK"
```

Therefore, we include an additional `fcol` argument to indicate which column of the `MSnSet` `fData` contains the names which will match our marker protein list, here the `Master.Protein.Accessions` column. When this argument is excluded, the `addMarkers` function default is to look for matches between the marker names and `MSnSet` feature names, which would not work.

```
## Add markers
unstim_msn_pep <- addMarkers(object = unstim_msn_pep,
                             markers = "mrk.csv",
                             fcol = "Master.Protein.Accessions",
                             mcol = "markers_initial")
```

```
## Markers in data: 11593 out of 31169

## organelleMarkers
##          chromatin            cytosol                 ER              ERGIC
##                366               2759               2138                105
##                 GA          lysosome      mitochondrion            nucleus
##                 84                210               2454               1936
##         peroxisome                 PM         proteasome ribosome/complexes
##                134                477                377                553
##            unknown
##              19576
```

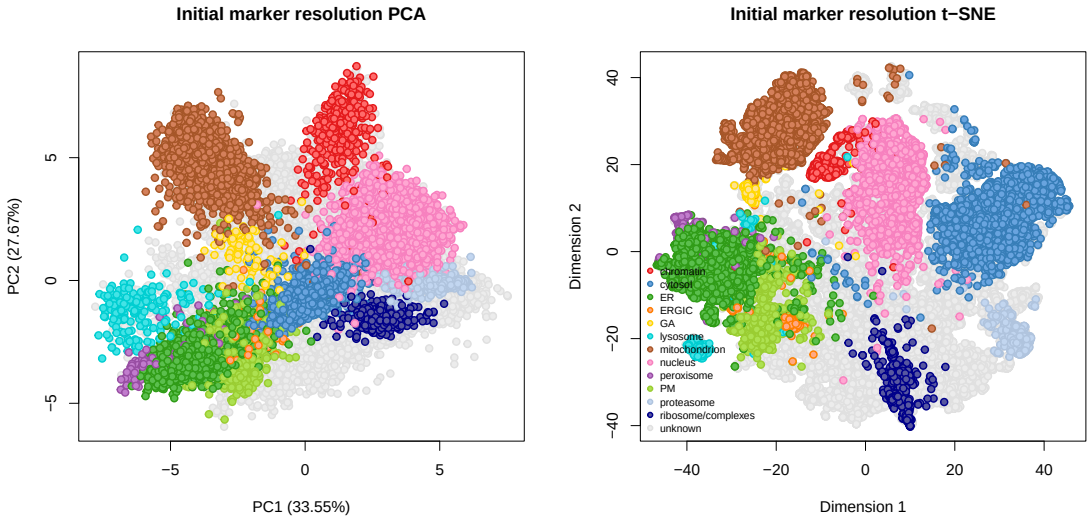
Now that we have added our markers, we can generate peptide-level maps using the `plot2D` function.

```
## Plot peptide-level maps
par(mfrow = c(1, 2))

unstim_msn_pep %>%
  plot2D(method = "PCA", fcol = "markers_initial",
         main = "Initial marker resolution PCA")

set.seed(399)
unstim_msn_pep %>%
  plot2D(method = "t-SNE", fcol = "markers_initial",
         main = "Initial marker resolution t-SNE")

unstim_msn_pep %>%
addLegend(where = "bottomleft", fcol = "markers_initial", cex = .7)
```




Users are referred back to the main workflow for how to complete marker curation and classification. The peptide-level `MSnSet` can be used in the same way as the protein-level use-case in the main workflow.

## References


Christopher, Josie A., Lisa M. Breckels, Oliver M. Crook, Mercedes Vazquez-Chantada, Derek Barratt, and Kathryn S. Lilley. 2025. "Global Proteomics Indicates Subcellular-Specific Anti-Ferroptotic Responses to Ionizing Radiation." *Molecular and Cellular Proteomics* 24 (1): 100888. https://doi.org/10.1016/j.mcpro.2024.100888.

Cox, Jürgen, Marco Y. Hein, Christian A. Luber, Igor Paron, Nagarjuna Nagaraj, and Matthias Mann. 2014. "Accurate Proteome-Wide Label-Free Quantification by Delayed Normalization and Maximal Peptide Ratio Extraction, Termed MaxLFQ." *Molecular &Amp; Cellular Proteomics* 13 (9): 2513–26. https://doi.org/10.1074/mcp.m113.031591.

Demichev, Vadim, Christoph B. Messner, Spyros I. Vernardis, Kathryn S. Lilley, and Markus Ralser. 2019. "DIA-NN: Neural Networks and Interference Correction Enable Deep Proteome Coverage in High Throughput." *Nature Methods* 17 (1): 41–44. https://doi.org/10.1038/s41592-019-0638-x.

Frankenfield, Ashley M., Jiawei Ni, Mustafa Ahmed, and Ling Hao. 2022. "Protein Contaminants Matter: Building Universal Protein Contaminant Libraries for DDA and DIA Proteomics." *Journal of Proteome Research* 21 (9): 2104–13. https://doi.org/10.1021/acs.jproteome.2c00145.

Geladaki, Aikaterini, Nina Kočevar Britovšek, Lisa M. Breckels, Tom S. Smith, Owen L. Vennard, Claire M. Mulvey, Oliver M. Crook, Laurent Gatto, and Kathryn S. Lilley. 2019. "Combining LOPIT with Differential Ultracentrifugation for High-Resolution Spatial Proteomics." *Nature Communications* 10 (1). https://doi.org/10.1038/s41467-018-08191-w.

Hutchings, Charlotte, Charlotte S. Dawson, Thomas Krueger, Kathryn S. Lilley, and Lisa M. Breckels. 2023. "A Bioconductor Workflow for Processing, Evaluating, and Interpreting Expression Proteomics Data." *F1000Research* 12 (October): 1402. https://doi.org/10.12688/f1000research.139116.1.

Käll, Lukas, Jesse D Canterbury, Jason Weston, William Stafford Noble, and Michael J MacCoss. 2007. "Semi-Supervised Learning for Peptide Identification from Shotgun Proteomics Datasets." *Nature Methods* 4 (11): 923–25. https://doi.org/10.1038/nmeth1113.

McCaskie, Kieran. 2025. "Localisation of Organelle Proteins Using Data-Independent Acquisition (DIA-LOP)." *Submitted 22.04.2025*.

Pages, H., P. Aboyoun, R. Gentleman, and S. DebRoy. 2022. *Biostrings: Efficient Manipulation of Biological Strings*.

Schessner, Julia P., Vincent Albrecht, Alexandra K. Davies, Pavel Sinitcyn, and Georg H. H. Borner. 2023.

"Deep and Fast Label-Free Dynamic Organellar Mapping." *Nature Communications* 14 (1). https://doi.org/10.1038/s41467-023-41000-7.