Urine Analysis Notes

Dr. Robert Buscaglia

August 08, 2023

This document will provide an overview of the work done with Urine Thermograms during the Spring 2023 semester conducted by MS Statistics student Bryan Sulzen. This PDF attempts to organize the many functions derived and resulting automated thermogram functionality. I have tried to include as much as possible, but there were many avenues attempted that are not included here. Much of this work was revisited to ensure consistency in the analysis and results, with the goal of propagating further research ideas that can incorporate some of these automated results and improve the analysis of urine thermograms.

Data Import and Notes

There were several data files provided. This included raw urine scans, but also buffers background samples and some saliva samples. We have focused primarily on the raw urine thermograms as presented in the file 'Urine Samples thermograms raw 4.23.2023 corrected.xlsx'. There are certainly uses for buffer backgrounds, as well as some paired plasma data, but this has not been evaluated to date.

The raw urine thermogram file contains 1572 columns with 5396 rows each. The large number of rows is due to each thermogram, in its raw form, having a temperature and heat capacity column due to differences in temperature at signal collection. However, there is also significant differences in sample scan temperatures, especially after samples with ID 'T267a' and later. These thermograms have approximately one-fifth the temperature points as the earlier scans (scan rate change??).

```r
Urine_Raw_Corrected <- readxl::read_excel(
  path = 'data_raw/Urine samples thermograms raw 4.23.2023 corrected.xlsx',
  sheet='Data')
str(Urine_Raw_Corrected, list.len=4)
```

```
## tibble [5,396 x 1,572] (S3: tbl_df/tbl/data.frame)
##  $ T1a  : num [1:5396] 20 20 20 20 20 ...
##  $ 1a   : num [1:5396] -1.17 -2.68 -3.46 -3.38 -3.12 ...
##  $ T1b  : num [1:5396] 20 20 20 20 20 ...
##  $ 1b   : num [1:5396] 0.1495 -0.0603 1.4881 2.466 1.9123 ...
##   [list output truncated]
```

There are also rows that are labeled for samples but are completely blank. This included the following samples (which did not have a reading in the first row):

```r
Check.na.cols <- Urine_Raw_Corrected %>% slice(1) %>% is.na() %>% which()
colnames(Urine_Raw_Corrected)[Check.na.cols]
```

```
##  [1] "T91b"  "91b"   "T174a" "174a"  "T174b" "174b"  "T179a" "179a"  "T321b"
## [10] "321b"
```

These columns were removed from the analysis before proceeding.

```r
### Removal of 10 empty columns
Urine_Raw_Corrected_Full <- Urine_Raw_Corrected %>% select(-all_of(Check.na.cols))
```

Next, data was formatted to a set ready for analysis, which included creating columns labels for 'Temperature', 'dCp', 'SampleID', 'SampleNumber', 'SampleIteration'. This set is then used as our working set for further analysis.

```r
#### Prepare rbind concatenation for long-form data handling
#### This will take some time to concatenate all 781 samples
Urine.Working <- NULL
Total.Samples <- ncol(Urine_Raw_Corrected_Full)/2
for(j in 1:Total.Samples)
{
  lwr <- 2*j - 1
  upr <- 2*j
  if(j %% 20 == 0) cat(j, ' of ', Total.Samples, 'completed. \n')
  temp.col <- Urine_Raw_Corrected_Full %>% select(lwr:upr)
  temp.col <- temp.col %>% mutate(SampleID = colnames(temp.col)[2])
  colnames(temp.col)[1:2] <- c('Temperature', 'dCp')
  Urine.Working <- Urine.Working %>% rbind(temp.col)
}
### Create variables for patient identification and tracking
### Remove all NA rows (non-aligned temperatures)
Urine.Working <- Urine.Working %>%
  filter(!is.na(Temperature)) %>%
  mutate(SampleNumber = factor(str_extract(SampleID, '\\d+')),
         SampleIteration = factor(str_extract(SampleID, '[a-f]'))) %>%
  mutate(SampleID = factor(SampleID)) %>%
  relocate(SampleID)
```

```r
### Save image for easy data reloading // clear all other elements
rm(list = setdiff(ls(), c('Urine.Working')))
# save.image('data/Urine_Working.RData') ### This should only be run once
```

At this time we have our working set that can be loaded (quickly) using

```r
load('data/Urine_Working.RData')
```

Graphical Overview of Samples

At this time it is best to get a few of what we are looking for. All data is on different scales, has different temperature ranges, and can have several iterations per sample number. It is likely best to save these as PDFs with all 781 samples on different pages. This is generated below (does not need to be run unless you want to create the PDF yourself).

```
### Store all sampleID for future use
All_Urine_ID <- Urine.Working %>% pull(SampleID) %>% unique() %>% as.vector()
```

```
### Create a PDF of all RAW thermograms (no changes)
pdf(file = 'Urine_All_Raw.pdf')
{
  for(j in All_Urine_ID)
  {
    working.sample <- Urine.Working %>% filter(SampleID == j)
    graph.out <- ggplot(working.sample, aes(x = Temperature, y = dCp)) + geom_line() +
      labs(title = paste0('Sample ', j))
    print(graph.out)
  }
}
dev.off()
```

It is evident from the visualization of the raw thermograms that temperature truncation can be done. This is implemented to reduce temperature range for all samples to 25 - 95 C.

```
Urine.Working.Final <- Urine.Working %>% filter(between(Temperature, 25, 95))
```

We can visualize the changes made by truncation. Please be aware of the size of the PDFs being created if running these again.

```
### Create a PDF of all RAW thermograms (Truncated Temperature)
pdf(file = 'Urine_All_Raw_TempTrunc.pdf')
{
  for(j in All_Urine_ID)
  {
    working.sample <- Urine.Working.Final %>% filter(SampleID == j)
    graph.out <- ggplot(working.sample, aes(x = Temperature, y = dCp)) + geom_line() +
      labs(title = paste0('Sample ', j))
    print(graph.out)
  }
}
dev.off()
```

Assessing Roughness and Signal Detection

A significant amount of time was dedicated to evaluating signal roughness and determining presence of signal. We may discuss new options moving forward, but there were many steps made here originally that may be of interest. Methods to evaluate roughness included

- Evaluation of changes in monotonicity (i.e. how many ups/down in a row?)
- Autocorrelation of raw signal
- Evaluation of signal as white-noise

Any of these elements could be used for a supervised analysis. However, the determination of a white-noise signal can also be treated as a direct statistical test. As all of the signals have some type of 'drift' or linear slope to the data, white-noise assessment after taking first differences should indicate the presence of signal/noise with statistical testing. These methods are discussed below:

Monotonic Evaluation

We wrote functions to evaluate how often curves change direction. Samples with large signal and defined peaks should show regions of significant monotone change. No-signal or heave noise will have rapid changes. The problem with this idea was that many of the curves have long monotonic decreases at higher temperature, which confounds with large monotonic changes where peaks are present.

We prepared a set of first difference curves to conduct this work.

```r
Urine.Differences <- NULL
### for each sample, take the first difference (left/lower truncated)
for(j in All_Urine_ID)
{
  working.sample <- Urine.Working.Final %>% filter(SampleID == j) %>%
    select(Temperature, dCp)
  working.diff <- data.frame(Temperature = working.sample$Temperature[-1],
                             Diff = diff(working.sample$dCp), SampleID = j)
  ### Store each first-difference curve (temp and delta-dCp)
  Urine.Differences <- Urine.Differences %>% rbind(working.diff)
}
```

Remove all objects no-longer in use:

```r
### Save image for easy data reloading // clear all other elements
rm(list = setdiff(ls(), c('Urine.Working.Final', 'Urine.Differences', 'All_Urine_ID')))
# save.image('data/Urine_Working.RData') ### This should only be run once
```

We can now quickly load all key information for evaluating roughness.

```r
load('data/Urine_Working.RData')
```

We can also visualize all first difference curves. Notice that clear distinct signal will show a first difference curve with distinct peaking, where as thermograms with no clear structure produce noise. We can exploit this later for a statistical test.

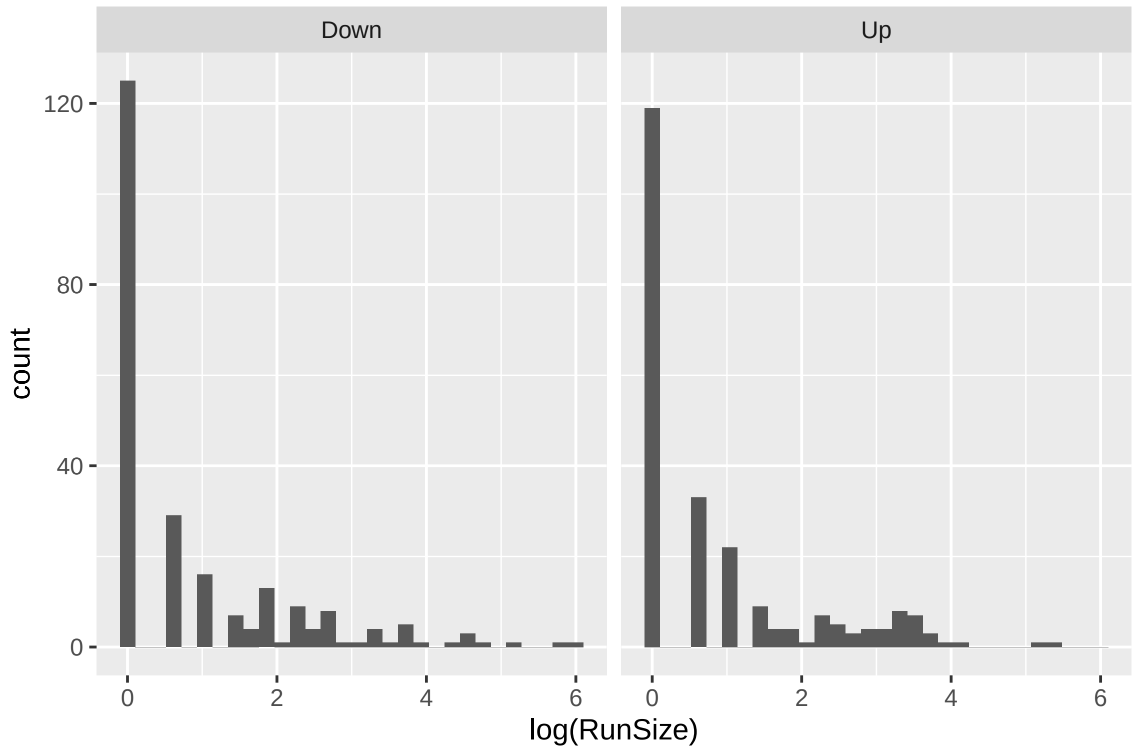
```r
### Create a PDF of all First Difference thermograms
pdf(file = 'Urine_First_Differences.pdf')
{
  for(j in All_Urine_ID)
  {
    working.sample <- Urine.Differences %>% filter(SampleID == j)
    graph.out <- ggplot(working.sample, aes(x = Temperature, y = Diff)) + geom_line() +
      labs(title = paste0('Sample ', j))
    print(graph.out)
  }
}
dev.off()
```

We then looked into functions that could evaluate how often curves turn as a measure of signal vs. noise. This was an excellent thread of work where we spent significant time before realize the large monotonic tails were causing some confounding issues. For clarity, in case this avenue is of interest, we produced a function that could count how many monotonic changes occurred in a row (runs of ups, runs of downs, with a run stopping if the sign changes).

```
### Function that determines if the signal moves up or down
### Requires being provided first difference curve (i.e. the deltas)
run.indicators <- function(x) return(ifelse(x <= 0, -1, 1))

### Function that counts runs of ups/downs
### Relies on the above function to count how many consecutive ups/downs
updown.count <- function(x){
  require(tidyverse)
  j = 1
  n.its <- length(x)
  df.out <- data.frame()
  count <- 1
  while(j <= n.its){
    if(x[j] == x[j+1] & j != n.its)
    {
      j <- j+1
      count <- count+1
    } else {
      out <- c(count, run.indicators(x[j]))
      df.out <- df.out %>% rbind(out)
      j <- j+1
      count <- 1
    }
  }
  colnames(df.out) <- c('RunSize', 'Up/Down')
  df.out$`Up/Down` <- ifelse(df.out$`Up/Down` < 0, 'Down', 'Up')
  return(df.out)
}
```

We can then investigate signals for roughness by counting runs, here is an example.

```
working.difference <- Urine.Differences %>% filter(SampleID == '1a')
working.indicators <- run.indicators(working.difference$Diff)
working.roughness <- updown.count(working.indicators)
ggplot(working.roughness, aes(x = log(RunSize))) + geom_histogram() + facet_grid(.~`Up/Down`)
```



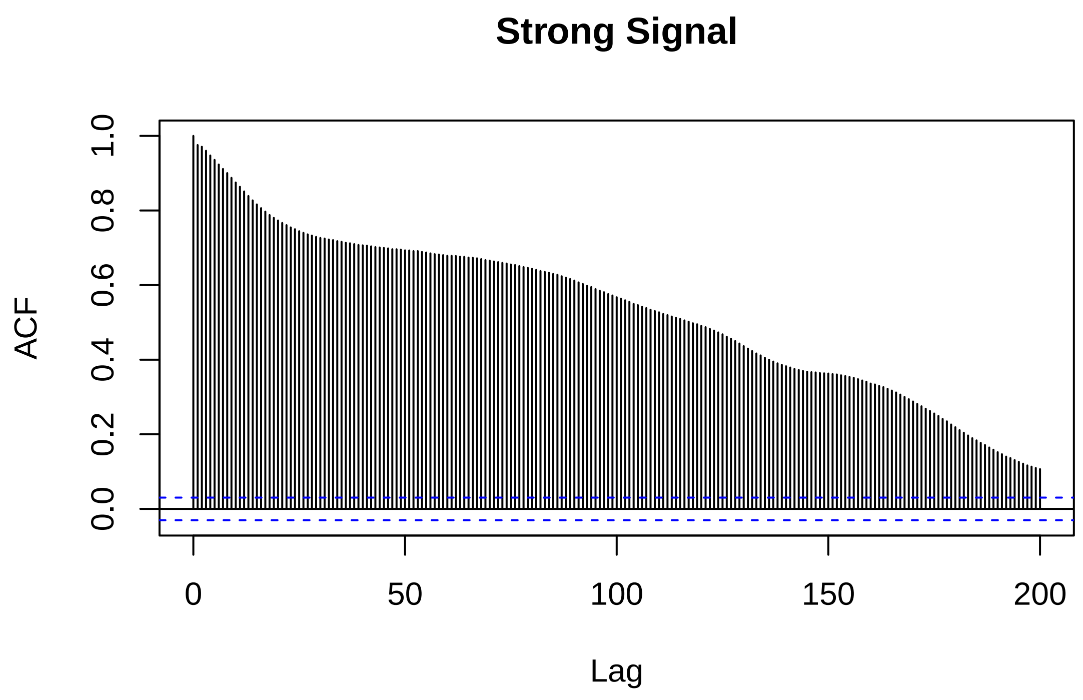
The problem as we continued down this path was the inability to clearly distinguish the roughness due to several different features of thermogram signatures. Specifically, we thought the presence of very long up-runs would indicate a peak is present, followed by a long down-run. However, this is confounded if there is monotonicity in the tails. We can re-approach this later if of interest.

Autocorrelation

Our next approach was to evaluate autocorrelation of first differences. As nearly all thermograms have the presence of a linear slope, first differences will eliminate this trend. We then ask if there is signal present in the first difference or not.

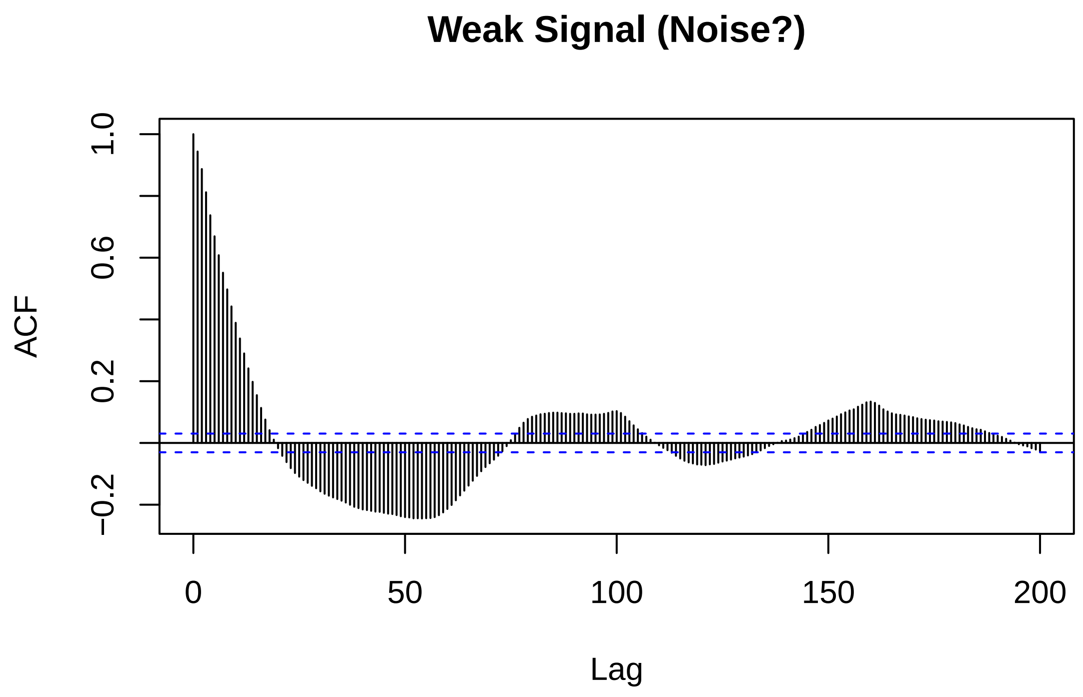
Here is what a strong signal gives:

```
working.difference <- Urine.Differences %>% filter(SampleID == '1a')
acf(working.difference$Diff, lag.max = 200, main = 'Strong Signal')
```



Here is the result from a noise signal.

```r
working.difference <- Urine.Differences %>% filter(SampleID == '2b')
acf(working.difference$Diff, lag.max = 200, main = 'Weak Signal (Noise?)')
```



NOTE: Supervised Learning using Trees

Based on the autocorrelation, we did develop a tree-based estimation procedure. This was not as effective based on our initial evaluations, but could be reconsidered so adding this note here. The autocorrelation (and partial ACF) can be calculated and used as input to a supervised learning method. This method relies on the initial guesses for strong predictions. We only attempted simple trees, but more advanced ML techniques (Random Forests?) could be considered if this is of interest.

Estimated Signal/Noise based on Stationary First-Difference

We can try to determine if there is signal present by treating the data as a time series. Here, we are asking, is there non-stationary components in the first difference signal? If yes, this may indicate signal as there would be something like 'peaks' retained in the first difference curve. However, if by taking the first difference we obtain a stationary signal, then the presence of any peaks is missing; thus, this may indicate noise. This uses the forecast package to make detection of the difference automatic.

```
ACF.detect <- NULL
### for each sample, determine if the first-difference is stationary
for(j in 1:length(All_Urine_ID))
{
  if(j %% 20 == 0) cat(j, '\n')
  working.difference <- Urine.Differences %>% filter(SampleID == All_Urine_ID[j])
  ### auto.arima fits a time-series model, we are interested in the Delta
  ### Delta = 0 implies stationary, Delta > 0 implies non-stationary
  out <- auto.arima(working.difference$Diff)$model$Delta
  detection <- ifelse(length(out)==0, 'No Signal', 'Signal')
  bind.temp <- tibble(SampleID = All_Urine_ID[j], Detection = detection)
  ACF.detect <- ACF.detect %>% rbind(bind.temp)
}
# rm(list = setdiff(ls(), 'ACF.detect'))
# save.image('data/ACF_Detect.RData')
```

This provides a first estimate if the signal contains strong peaks resulting in first differences having non-stationary components. We can then filter any sample and request if this method determined if signal or noise was present. Below I present the first 10 sample estimates, as well as ask for a particular samples estimate.

```
load('data/ACF_Detect.RData')
head(ACF.detect, n = 10)
```

```
## # A tibble: 10 x 2
##    SampleID Detection
##    <chr>    <chr>
##  1 1a       Signal
##  2 1b       Signal
##  3 2a       Signal
##  4 2b       No Signal
##  5 3a       No Signal
##  6 3b       No Signal
##  7 4a       Signal
##  8 4b       No Signal
##  9 5a       Signal
## 10 5b       Signal
```

```
ACF.detect %>% filter(SampleID == '1a')
```

```
## # A tibble: 1 x 2
##   SampleID Detection
##   <chr>    <chr>
## 1 1a       Signal
```

Present a confusion matrix based on guesses of signal/noise put together by the team. Here, 0 == no signal, 1 == mixed, 2 == signal.

```
estimated.signals <- readxl::read_excel('BlindDetection_FIXED.xlsx')
estimated.signals <- estimated.signals %>% select(Patient, Level)
joined.estimates <- full_join(ACF.detect, estimated.signals, by = c('SampleID' = 'Patient'))
table(joined.estimates$Detection, joined.estimates$Level)
```

```
##
##               0   1   2
##   No Signal  97 187  25
##   Signal     36 188 248
```

As can be seen, the method does have decent sensitivity to the noise and signal cases. We obtain 97/133 (73%) properly estimated noise samples based on the test cases, and 248/273 (91%) of the signal cases. The mixed samples here are roughly 50/50, agreeing with the 'by-hand' analysis that these samples are difficult to tease apart.

Signal Detection Conclusions

Based on this work, no single method seemed highly effective (>90% accuracy) in detecting signal vs. noise based on the estimated results collected across three separate individuals. There is certainly good considerations above. By looking into the 'roughness' of a curve, we can extract elements of the analysis that determine when a signal has significant noise, but due to long runs of ups/down outside of the region of interest, this method failed to clearly distinguish signal from noise. A probability estimate could be further developed based on a binomial distribution of counting ups/downs, but this path was not investigated further once it was noticed that either signal or noise can have long monotonic runs.

The treatment of the thermograms as time-series seemed most effective. By evaluating if the first difference curve is stationary, we obtained significant ability to determine the well established signal samples (91% properly predicted). This method is automatic, but does nothing to help distinguish the intermediate cases, and only properly estimated 73% of the noise cases.

Other considerations can continue to be made, but we were encouraged to focus on baseline approximations at this time in our work.

Automated Baseline Predictions

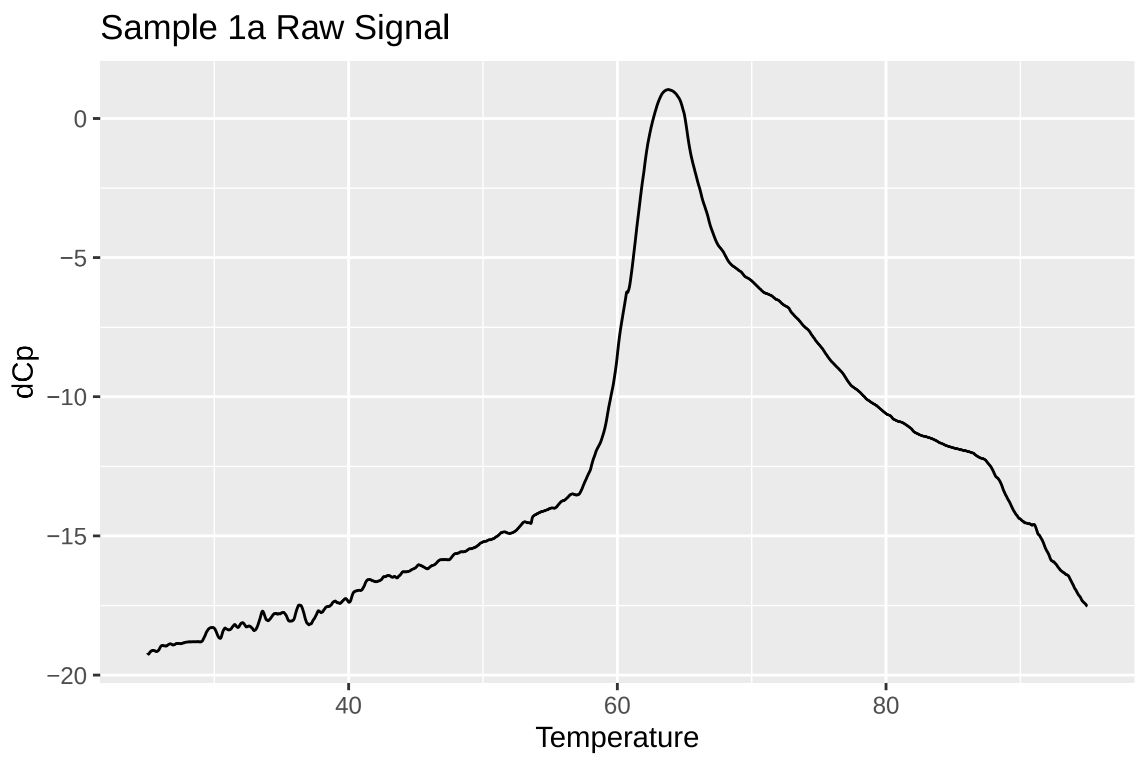
As this area of work seemed the most promising, I will attempt to provide rigorous breakdown to how we developed what we have coined the 'Moving-Window Variance Baseline Estimator'. At the end of this section, you will be provided a function that can take in a raw thermogram reading, and return an interpolated baseline-subtracted thermogram ready for further analysis.

Baseline Splines

The first element we considered was placing splines on the outer regions where signal is not present. As is well known, there are many methods for placing a baseline on a thermogram. However, most of these cases have to do with how the two baseline regions are connected, with the need for the outer regions to be zero well established. By using splines in this region, we may be adding more 'points' to out baseline fit, but we achieve the desired result of zero-ing out the non-signal regions. The thermogram will then likely be truncated, so although the by-hand methods used previously may provide slightly different zeroed regions, ours is effective and automated. The difficult is choosing the temperature cutoffs (discussed below).

Lets start by showing how a spline can be used to zero these regions. First, we need to select a sample to work with, lets choose the reliable `1a` sample with strong signal.

```r
working.sample <- Urine.Working.Final %>% filter(SampleID == '1a') %>%
  select(Temperature, dCp)
ggplot(working.sample, aes(x = Temperature, y = dCp)) + geom_line() +
  labs(title = 'Sample 1a Raw Signal')
```



We then see the immediate need to choose a lower and upper cutoff point for the baseline. For this example, it looks like ~54-58 C would be optimal for the lower temperature cutoff. The upper temperature cutoff is a bit more difficult to see, even by careful eye-ball detection, but lets choose 86C for now.

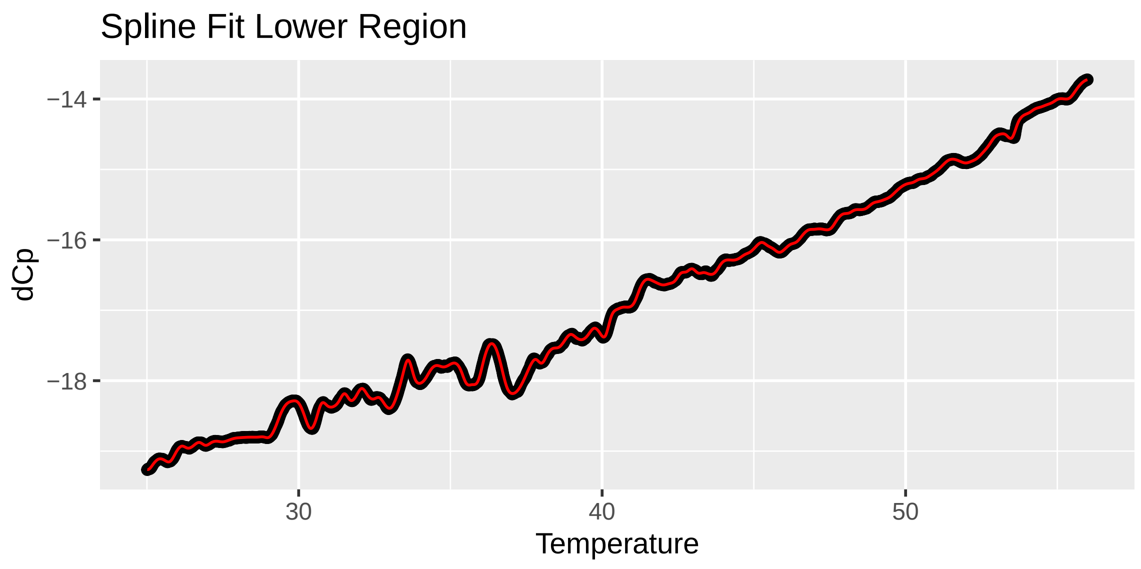
```r
### Developing Splines Example
lower <- 56 # All points lower than this temperature considered baseline
upper <- 86 # All points higher than this temperature considered baseline

### Extract the baseline regions
work.lower <- working.sample %>% filter(Temperature < lower)
work.upper <- working.sample %>% filter(Temperature > upper)
```

We will discuss how to automate the choice of upper/lower below, but we can certainly produce a function so that this value can be entered by a technician, the rest then automated. We then fit validated splines to these regions, which we can then use to 'blank' or zero-out these areas.

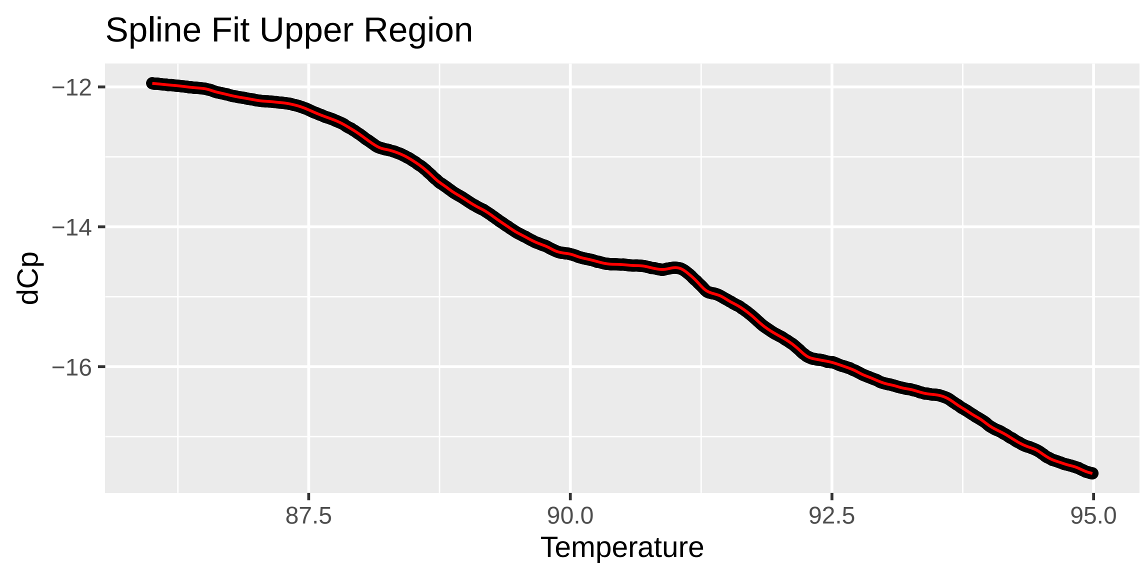
```r
### Splines for lower/upper regions
spline.lower <- smooth.spline(work.lower$Temperature, work.lower$dCp, cv = TRUE)
spline.upper <- smooth.spline(work.upper$Temperature, work.upper$dCp, cv = TRUE)
### Store data for graphing
spline.lower.fit <- data.frame(Temperature = work.lower$Temperature, fit = spline.lower$y)
spline.upper.fit <- data.frame(Temperature = work.upper$Temperature, fit = spline.upper$y)
```

Lets view what this does, first we show the raw chosen region with the spline overlaid.

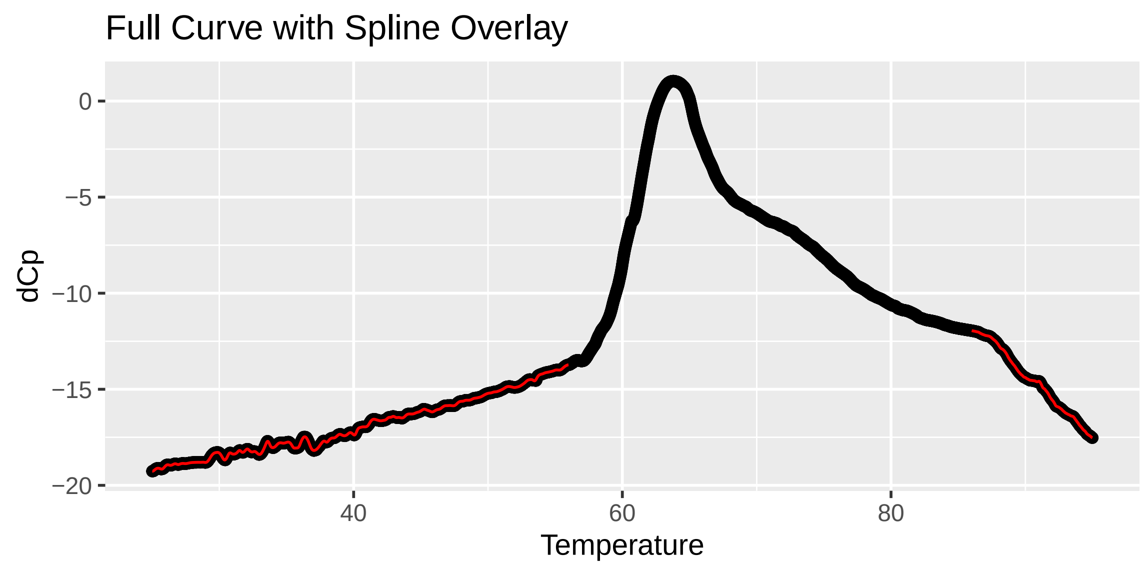
```r
### View of spline fit on lower region
work.lower %>% ggplot(aes(x = Temperature, y = dCp)) + geom_point() +
  geom_line(data = spline.lower.fit, aes(x = Temperature, y = fit), color = 'red') +
  labs(title = 'Spline Fit Lower Region')
```



```r
### View of spline fit on upper region
work.upper %>% ggplot(aes(x = Temperature, y = dCp)) + geom_point() +
  geom_line(data = spline.upper.fit, aes(x = Temperature, y = fit), color = 'red') +
  labs(title = 'Spline Fit Upper Region')
```



```r
### View of spline fit piecewise
working.sample %>% ggplot(aes(x = Temperature, y = dCp)) + geom_point() +
  geom_line(data = spline.lower.fit, aes(x = Temperature, y = fit), color = 'red') +
  geom_line(data = spline.upper.fit, aes(x = Temperature, y = fit), color = 'red') +
  labs(title = 'Full Curve with Spline Overlay')
```

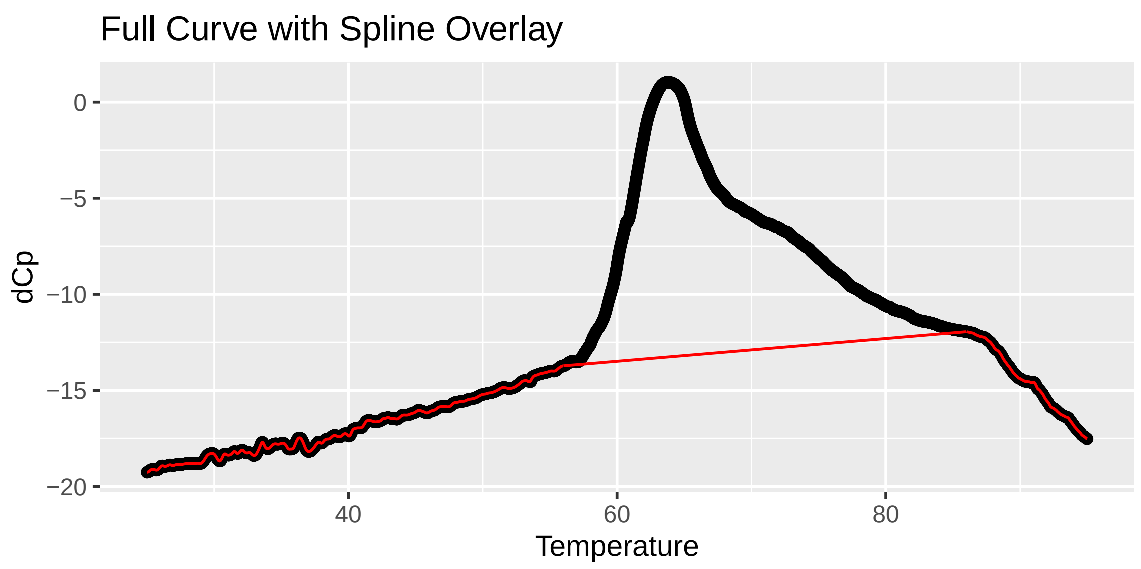


The choice of how to connect these two splines is where there is significant discussion in the DSC world. The Garbett lab uses a simple linear connection, so we found no reason to change this (although we did play with some more 'flexible' connections, linear is the most practical). To connect the lines, we do something silly, but it works! We take the last point of the lower region, the first point of the upper region, and connect them with a simple line (fast, efficient, nothing tricky here - only relies on the spline).

```r
### store middle (signal) region
work.mid <- working.sample %>% filter(between(Temperature, lower, upper))
### find endpoints of splines
spline.connect.points <- rbind(
  spline.lower.fit %>% filter(Temperature == max(Temperature)),
  spline.upper.fit %>% filter(Temperature == min(Temperature)))
### connect endpoints and store
spline.connect.lm <- lm(fit ~ Temperature, data = spline.connect.points)
spline.connect.fit <- data.frame(
  Temperature = work.mid$Temperature,
  fit = predict(spline.connect.lm, data.frame(Temperature = work.mid$Temperature)))
```

Here is the baseline prediction as done by hand, using the automation of spline fitting. However, as you can see here, I did choose the upper and lower, which is functionality we could put into a executable function. Maybe we should discuss details of how to make this intractable for ease of use?

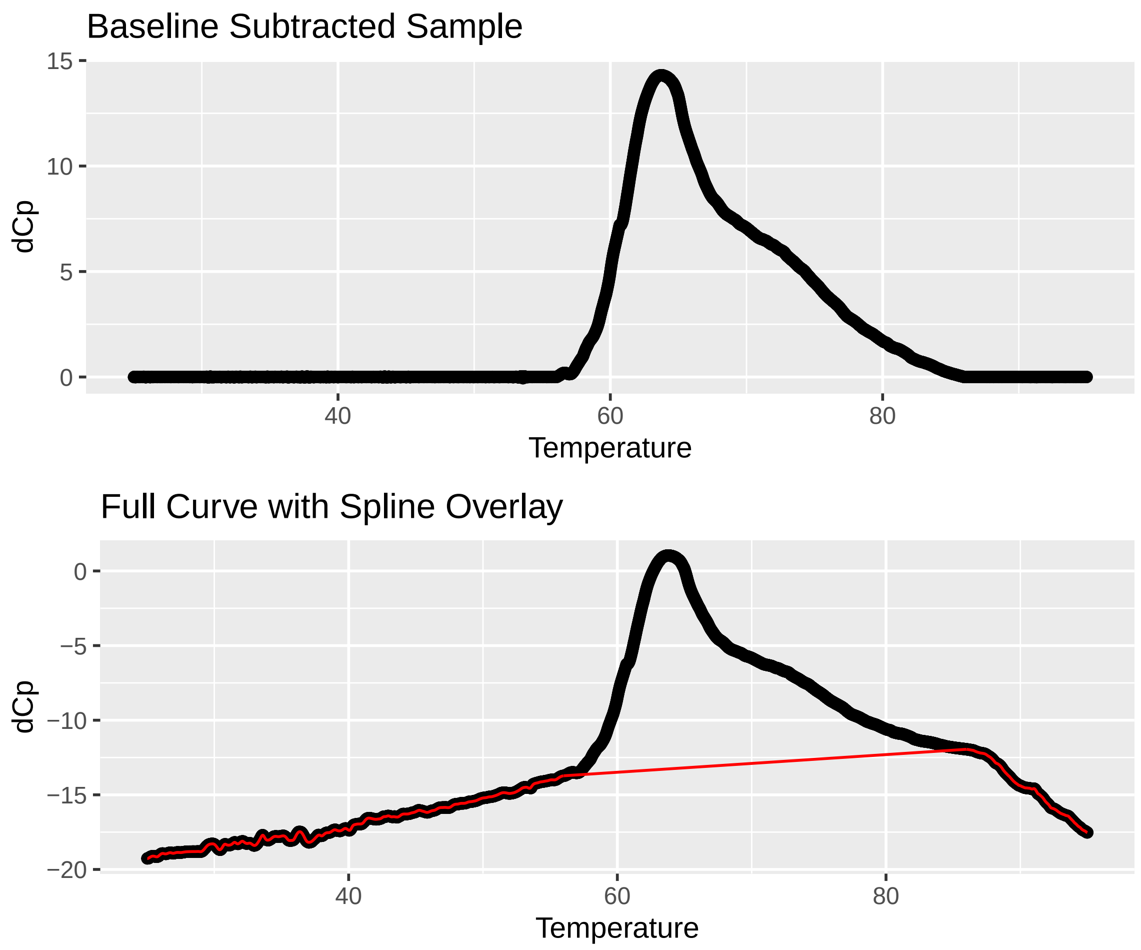
```
### View of spline fit on lower region
g.spline <- working.sample %>% ggplot(aes(x = Temperature, y = dCp)) + geom_point() +
  geom_line(data = spline.lower.fit, aes(x = Temperature, y = fit), color = 'red') +
  geom_line(data = spline.upper.fit, aes(x = Temperature, y = fit), color = 'red') +
  geom_line(data = spline.connect.fit, aes(x = Temperature, y = fit), color = 'red') +
  labs(title = 'Full Curve with Spline Overlay')
g.spline
```



We can do quick baseline subtraction at this point, producing a final thermogram result. Below we subtract the baseline from the working sample and produce a finalized thermogram.

```
### store baseline as one unit
working.baseline.final <- rbind(spline.lower.fit, spline.connect.fit, spline.upper.fit)
### join for tidyverse simplification
baseline.join <- full_join(working.sample, working.baseline.final, by = 'Temperature')
### final sample!
final.sample <- baseline.join %>% mutate(final.dcp = dCp - fit) %>%
  select(Temperature, final.dcp) %>% rename(dCp = final.dcp)
```

```
### View of final sample above baseline procedure
g.final <- final.sample %>% ggplot(aes(x = Temperature, y = dCp)) + geom_point() +
  labs(title = 'Baseline Subtracted Sample')
cowplot::plot_grid(g.final, g.spline, nrow=2)
```



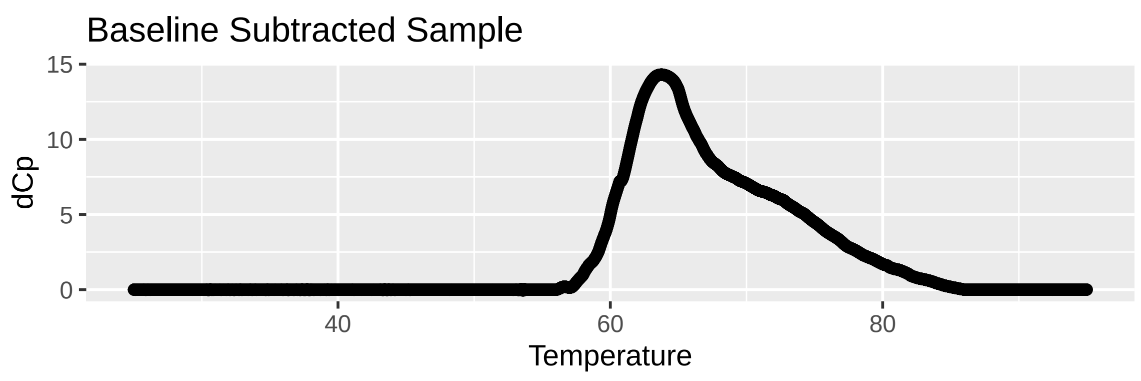
Function for 'by-hand' baseline subtraction

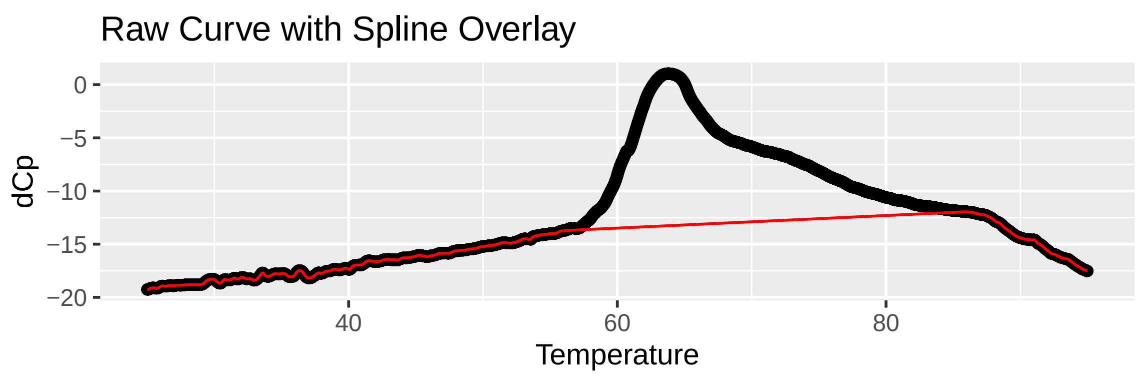
Lets take this work here, which allows for baseline subtraction using a technician's eye, and wrap it together for easy use. The function will take in a data.frame that contains two columns (Temperature and dCp). We will also allow for by-hand choice of a lower and upper cutoff (lwr.temp, upr.temp). I'll produce a function that will allow for output of the final thermogram, with functionality to 'view' the result while working. This can be turned off/on if desired. Here we go!

```r
### baseline.subtraction.byhand
###
### @use
### Function for by-hand baseline subtraction
### This can be used to introduce some automation to the temperature choices
### but does provide a function in R for doing baseline subtraction!
###
### @requires
### dplyr
### cowplot
###
### @inputs
### x = data.frame that contains two columns: Temperature and dCp
### lwr.temp : lower cutoff temperature
### upr.temp : upper cutoff temperature
### plot.on : outputs a graphic of final sample + baseline procedure when TRUE
###
### @function
baseline.subtraction.byhand <- function(x, lwr.temp, upr.temp, plot.on = TRUE)
{
  ### check-conditions of boundaires - this effects automation
  if(lwr.temp < min(x$Temperature)+1) lwr.temp = lwr.temp + 1
  if(upr.temp > max(x$Temperature)-1) upr.temp = upr.temp - 1
  ### Extract the baseline regions
  work.lower <- x %>% filter(Temperature < lwr.temp)
  work.upper <- x %>% filter(Temperature > upr.temp)
  ### Splines for lower/upper regions
  spline.lower <- smooth.spline(work.lower$Temperature, work.lower$dCp, cv = TRUE)
  spline.upper <- smooth.spline(work.upper$Temperature, work.upper$dCp, cv = TRUE)
  ### Store data for graphing
  spline.lower.fit <- data.frame(Temperature = work.lower$Temperature, fit = spline.lower$y)
  spline.upper.fit <- data.frame(Temperature = work.upper$Temperature, fit = spline.upper$y)

  ### store middle (signal) region
  work.mid <- x %>% filter(between(Temperature, lwr.temp, upr.temp))
  ### find endpoints of splines
  spline.connect.points <- rbind(
    spline.lower.fit %>% filter(Temperature == max(Temperature)),
    spline.upper.fit %>% filter(Temperature == min(Temperature)))
  ### connect endpoints and store
  spline.connect.lm <- lm(fit ~ Temperature, data = spline.connect.points)
  spline.connect.fit <- data.frame(
    Temperature = work.mid$Temperature,
    fit = predict(spline.connect.lm, data.frame(Temperature = work.mid$Temperature)))

  ### store baseline as one unit
  working.baseline.final <- rbind(spline.lower.fit, spline.connect.fit, spline.upper.fit)
  ### join for tidyverse simplification
  baseline.join <- full_join(x, working.baseline.final, by = 'Temperature')
  ### final sample!
  baseline.sample <- baseline.join %>% mutate(final.dcp = dCp - fit) %>%
    select(Temperature, final.dcp) %>% rename(dCp = final.dcp)

  if(plot.on)
  {
    ### graph of raw with spline
    g.spline <- working.sample %>% ggplot(aes(x = Temperature, y = dCp)) + geom_point() +
      geom_line(data = spline.lower.fit, aes(x = Temperature, y = fit), color = 'red') +
      geom_line(data = spline.upper.fit, aes(x = Temperature, y = fit), color = 'red') +
      geom_line(data = spline.connect.fit, aes(x = Temperature, y = fit), color = 'red') +
      labs(title = 'Raw Curve with Spline Overlay')
    ### final baseline subtracted sample
    g.final <- baseline.sample %>% ggplot(aes(x = Temperature, y = dCp)) + geom_point() +
      labs(title = 'Baseline Subtracted Sample')
    ### overlaid output
    print(cowplot::plot_grid(g.final, g.spline, nrow=2))
  }

  return(baseline.sample)
}
```

We now have a function for doing baseline subtraction in R!

```
### select a sample
working.sample <- Urine.Working.Final %>% filter(SampleID == '1a') %>%
  select(Temperature, dCp)
### baseline subtraction with by-hand upr/lwr selection
baseline.output <- baseline.subtraction.byhand(
  x = working.sample,
  lwr.temp = 56,
  upr.temp = 86)
```





The output is an easy to use tibble on the original temperature grid.

```
### output of function is a flexible tibble for further use.
str(baseline.output)
```

```
## tibble [4,195 x 2] (S3: tbl_df/tbl/data.frame)
##  $ Temperature: num [1:4195] 25 25 25 25.1 25.1 ...
##  $ dCp        : num [1:4195] -0.00634 0.00484 0.0084 0.00736 0.00135 ...
```

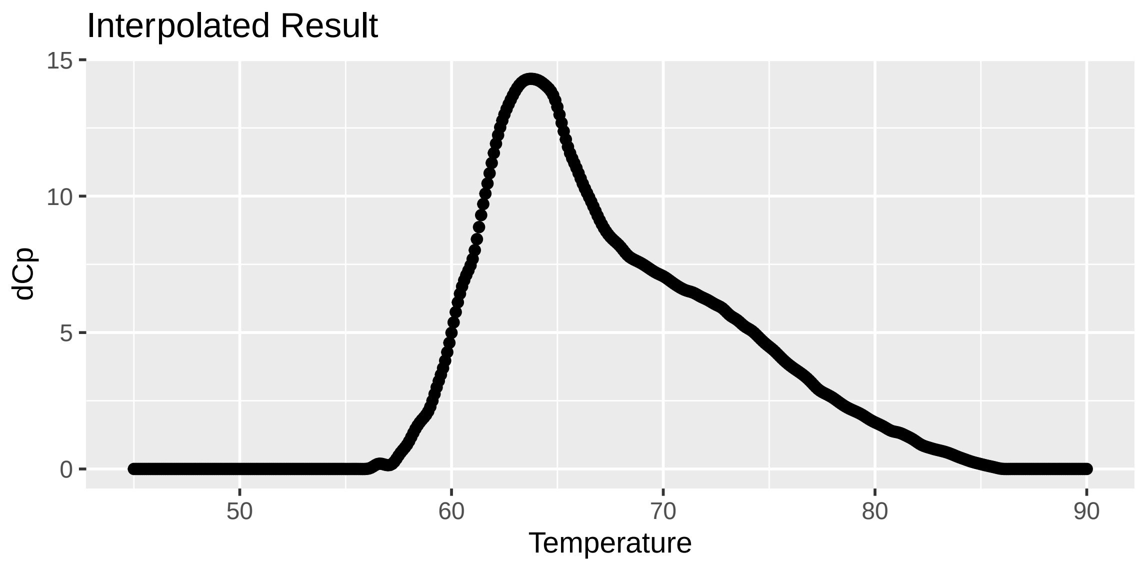
Interpolation

Now that we have a function that will do the baseline procedure, lets wrap-up the interpolation to a standard grid. Provided here is a function that will take as input the baseline subtracted sample from above plot the desired output temperature grid, and produce the final product that might be used in future analysis (dCp at say 45 to 90 by 0.1).

```
### final.sample.interpolate
###
### @use
### Takes a baseline subtracted sampled and produces an interpolated result
### on a chosen temperature grid.
###
### @requires
### dplyr
### cowplot
###
### @inputs
### x = baseline subtracted data.frame (use baseline function above!)
### grid.temp : grid of desired temperatures
### plot.on : outputs a graphic of interpolated sample?
###
### @function
final.sample.interpolate <- function(x, grid.temp, plot.on = TRUE)
{
  spline.fit <- smooth.spline(x$Temperature, x$dCp, cv = TRUE)
  interpolated.sample.pred <- predict(spline.fit, grid.temp)
  interpolated.sample <- data.frame(Temperature = grid.temp,
                                    dCp = interpolated.sample.pred$y)

  if(plot.on)
  {
    g.out <- ggplot(interpolated.sample, aes(x = Temperature, y = dCp)) + geom_point() +
      labs(title = 'Interpolated Result')
    print(g.out)
  }

  return(interpolated.sample)
}
```

Here is its use given the `baseline.output` result from the previous function.

```
final.sample <- final.sample.interpolate(
  x = baseline.output,
  grid.temp = seq(45, 90, 0.1))
```



The returned object is a flexible tibble on the desired grid.

```
### ready to use data
str(final.sample)
```

```
## 'data.frame':    451 obs. of  2 variables:
##  $ Temperature: num  45 45.1 45.2 45.3 45.4 45.5 45.6 45.7 45.8 45.9 ...
##  $ dCp        : num  3.91e-05 3.67e-05 2.82e-05 1.53e-05 -9.40e-07 ...
```

Moving-Window Variance for Upper and Lower Temperature Selection

How to automate the selection of the upper and lower temperatures? This is where we left off, with a few ideas I will implement here. The basic idea is to set a moving window size that might relate to something such as the scan rate (although some samples had different sizes, see the section Data Import and Notes). How to choose the window is still something to study more, but here is the full idea:

- Set a window size based on a number of temperature readings
  - we were using things like 60, 90, 120, which seemed to relate well to the size of the output temperatures - not sure what exact scan rate was?)
- Fit a CV spline to RAW curve
- Scan through upper/lower regions using window size for a variance estimate
  - we encourage the use of a 'region of known signal' - maybe 65 - 80? T
  - his eliminates a region of the curve from being scanned.
  - although when we allowed it to scan all the way up to half the temperatures on either side, things got interesting!
  - the estimates are created for all possible overlapping windows (I will try to show this below)
- temperature selected based on the region with MINIMUM variance
  - idea was that variance seems to minimize when a peak begins!

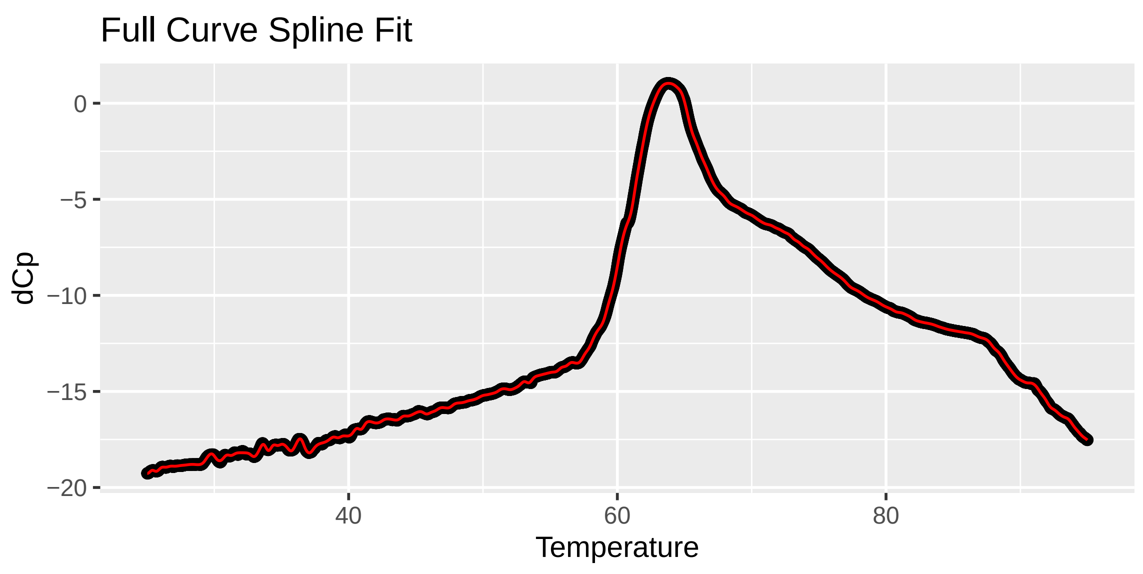
Moving-Window Variance Concept

Lets try to demonstrate this with our trusty sample `1a`. The idea is that when we fit a CV spline, we are returned an estimate of variance based on how well the spline fits the original data. Lets get started by selecting a sample and fitting spline to the entire curve.

```
### select sample
working.sample <- Urine.Working.Final %>% filter(SampleID == '1a') %>% select(Temperature, dCp
### fit a CV spline to entire raw curve
full.spline.fit <- smooth.spline(working.sample$Temperature, working.sample$dCp, cv=TRUE)
```

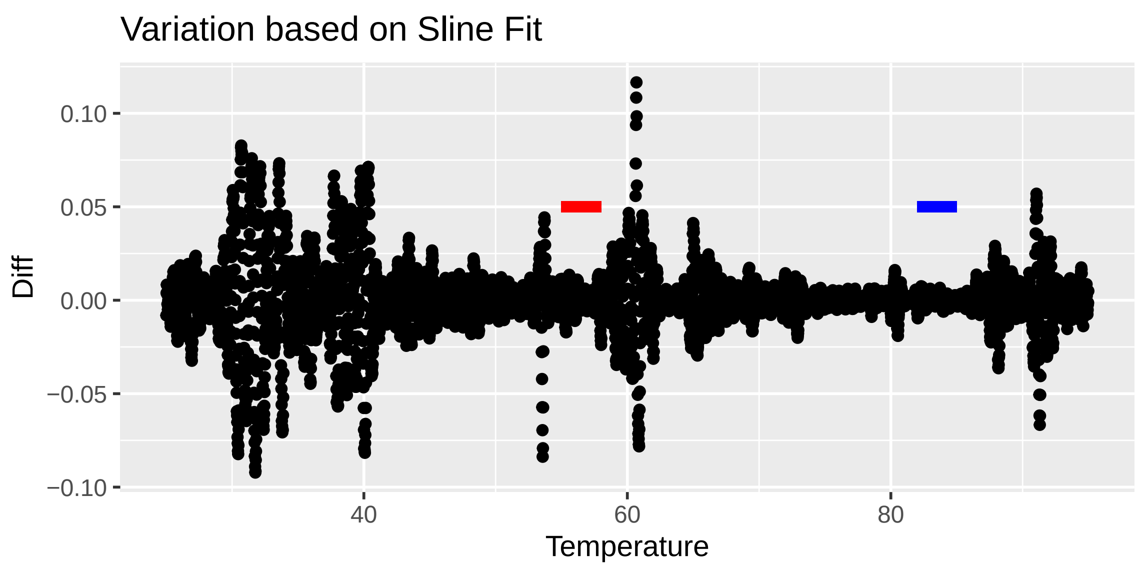
Graphic of the spline fit, which looks a lot like the above but to the whole curve.

```
spline.df <- data.frame(Temperature = working.sample$Temperature, fit = full.spline.fit$y)
ggplot(working.sample, aes(x = Temperature, y = dCp)) + geom_point() +
  geom_line(data = spline.df, aes(x = Temperature, y = fit), color = 'red') +
  labs(title = 'Full Curve Spline Fit')
```



This isn't really what we are interested in! We are interested in the errors of this fit instead.

```r
Fit.Variation <- data.frame(Temperature = working.sample$Temperature,
                            Diff = resid(full.spline.fit))
ggplot(Fit.Variation, aes(x = Temperature, y = Diff)) + geom_point() +
  geom_segment(x = 55, xend = 58, y = 0.05, yend = 0.05, color = 'red', linewidth=2) +
  geom_segment(x = 82, xend = 85, y = 0.05, yend = 0.05, color = 'blue', linewidth=2) +
  labs(title = 'Variation based on Sline Fit')
```



The concept here is that there seems to be large variation in the 'tails' with less variation near where peaks begin. There is clearly some significant variation within the peak area. However, when we looked at this across all the urine samples, this minimum variance seemed to line up well with peak departure. Here, the red line and blue line are to guide the eye to where the variance seems to minimize. Could we use this to chose a window?

Lets make a few assumptions first, things we could change. We will let the window include 90 temperature points. This sample has a total of 4195 temperature readings for reference. We will also eliminate the region from 60 to 80 C, as this is likely to contain the signal.

```
w = 90
exclude.lwr = 60
exclude.upr = 80
```

The code below will get a little complicated with looping, but the fundamental idea is above. We'll look within a window of 90 temperature points, starting from the first temperature reading, and scan through calculating a variance at each step. Here it is implemented for the lower region, scanning from the far left up to the point of exclusion. The first window would end at the `w = 90`th point, thus the first output would be at w = 90, then w = 91, and so-on until we reach the point of exclude. We then ask where the minimum variance was?

```
### spline residuals into a data.frame with ids for tracking.
r <- resid(full.spline.fit)
r.df <- data.frame(Temperature = working.sample$Temperature,
                   r=r,
                   id = 1:length(working.sample$Temperature))
### scan through lower region calculating variance within a window size of w
i=0
df.var <- data.frame()
### how far do we need to scan?
points.in.lower <- nrow(working.sample %>% filter(Temperature < exclude.lwr))
### scan and calculate variance for each window
while ((w+i) < points.in.lower){
  rout <- r.df %>% slice((1+i):(w+i)) %>% summarise(temp.stop=(w+i),mean = mean(r), sd=sd(r))
  df.var <- rbind(df.var, rout)
  i=i+1
}
### where was the minimum variance?
low <- df.var %>% filter(sd == min(sd))
lower <- working.sample$Temperature[low$temp.stop-w]
lower
```

```
## [1] 56.24152
```

This method with `w = 90` suggest using a lower cutoff at 56.2C by selecting the beginning of the window. I had chosen 56C by hand. The `df.var` data.frame takes a long time to estimate as it scans over a pretty wide window. Its output is a long list of estimated parameters that we want to find the minimum variance from.

```
head(df.var)
```

```
##   temp.stop          mean         sd
## 1        90 -0.0009468023 0.01041417
## 2        91 -0.0008650582 0.01038632
## 3        92 -0.0008805542 0.01038023
## 4        93 -0.0008967016 0.01036689
## 5        94 -0.0008710856 0.01039247
## 6        95 -0.0007628797 0.01048487
```

We can do the same but by scanning from the very last point in the thermogram toward lower temperatures until we hit the exclusion window.

```
### total size of thermogram
l <- length(working.sample$Temperature)
j=0
df.var.upper <- data.frame()
### how far from upper endpoint do we need to scan?
smallest.point.in.upr <- nrow(working.sample) - nrow(working.sample %>% filter(Temperature > e
### scan from highest point to region of exclusion
while ((l-w-j) > smallest.point.in.upr) {
  rout <- r.df %>% slice((l-w-j):(l-j)) %>% summarise(i=(l-w-j),mean = mean(r), sd=sd(r))
  df.var.upper <- rbind(df.var.upper, rout)
  j=j+1
}
### where was the minimum variance?
up <- df.var.upper %>% filter(sd == min(sd))
upper <- working.sample$Temperature[up$i+w]
upper
```

```
## [1] 85.57179
```

By scanning inward from the end, this suggests a temperature cutoff of 85.6C for the upper endpoint, of which I had chosen 86 by hand. This all depends on the `w = 90` and region of exclusion, but notice the estimates are not far away from our by-hand subtraction! The drawback is it is a bit slow, especially for low `w`, and when having to do many samples this can add up. We are just about ready to automate the whole process though.

Lets write a function that will apply the moving-window approach to a provided sample.

```
### moving.window
###
### @use
### Takes a raw thermogram and tries to estimate where the upper and lower
### cutoffs for baseline should be.  Not an easy task! This function can
### be slow when scanning over large windows or using a small window size (w).
###
### @requires
### dplyr
###
### @inputs
### x : raw thermogram as data.frame with Temperature and dCp columns.
### w : window size (default 90, try others!)
### exclusion.lwr : lower temperature exclusion point
### exclusion.upr : upper temperature exclusion point
###
### @function
moving.window <- function(x, w = 90, exclusion.lwr = 60, exclusion.upr = 80)
{
  ### fit a CV spline
  full.spline.fit <- smooth.spline(x$Temperature, x$dCp, cv=TRUE)
  ### spline residuals into a data.frame with ids for tracking.
  r <- resid(full.spline.fit)
  r.df <- data.frame(Temperature = working.sample$Temperature,
                  r=r,
                  id = 1:length(working.sample$Temperature))
  ### scan through lower region calculating variance within a window size of w
  cat('Scanning Lower. \n')
  i=0
  df.var <- data.frame()
  ### how far do we need to scan?
  points.in.lower <- nrow(working.sample %>% filter(Temperature < exclude.lwr))
  ### scan and calculate variance for each window
  while ((w+i) < points.in.lower){
   rout <- r.df %>% slice((1+i):(w+i)) %>% summarise(temp.stop=(w+i),mean = mean(r), sd=sd(r))
   df.var <- rbind(df.var, rout)
   i=i+1
  }
  ### where was the minimum variance?
  low <- df.var %>% filter(sd == min(sd))
  lower <- working.sample$Temperature[low$temp.stop-w]

  ### total size of thermogram
  cat('Scanning Upper. \n')
  l <- length(working.sample$Temperature)
  j=0
  df.var.upper <- data.frame()
  ### how far from upper endpoint do we need to scan?
  smallest.point.in.upr <- nrow(working.sample) - nrow(working.sample %>% filter(Temperature >
  ### scan from highest point to region of exclusion
  while ((l-w-j) > smallest.point.in.upr) {
    rout <- r.df %>% slice((l-w-j):(l-j)) %>% summarise(i=(l-w-j),mean = mean(r), sd=sd(r))
    df.var.upper <- rbind(df.var.upper, rout)
    j=j+1
  }
  ### where was the minimum variance?
  up <- df.var.upper %>% filter(sd == min(sd))
  upper <- working.sample$Temperature[up$i+w]

  output <- data.frame(lower = lower, upper = upper)
  return(output)
}
```

We can now automate the selection of the endpoints based on the rules above. We could continue to implement more rules if desired.

```
### select a sample
working.sample <- Urine.Working.Final %>% filter(SampleID == '1a') %>%
  select(Temperature, dCp)
```

```r
### Automate the selection of the endpoints
moving.window(
  x = working.sample,
  w = 90,
  exclusion.lwr = 60,
  exclusion.upr = 80
)
```

```
## Scanning Lower.
## Scanning Upper.
```

```
##      lower    upper
## 1 56.24152 85.57179
```

Before trying to wrap this all together into a fully automated approach, here is what we get if we use different window sizes for sample `1a`, keeping the same exclusion window.

```r
w.sizes <- c(30, 60, 90, 120, 150, 180)
window.df <- data.frame()
for(j in 1:length(w.sizes))
{
  out <- moving.window(
    x = working.sample,
    w = w.sizes[j],
    exclusion.lwr = 60,
    exclusion.upr = 80)
  out <- out %>% mutate(w = w.sizes[j]) %>% relocate(w)
  window.df <- window.df %>% rbind(out)
}
```

```
## Scanning Lower.
## Scanning Upper.
## Scanning Lower.
## Scanning Upper.
## Scanning Lower.
## Scanning Upper.
## Scanning Lower.
## Scanning Upper.
## Scanning Lower.
## Scanning Upper.
## Scanning Lower.
## Scanning Upper.
```

```r
window.df
```

```
##     w    lower    upper
## 1  30 51.27490 81.78838
## 2  60 56.55815 81.85505
## 3  90 56.24152 85.57179
## 4 120 55.72482 86.07181
## 5 150 49.82490 85.57179
## 6 180 49.04158 86.08848
```

As can be seen, when using too large or too small of a window, certain regions may be overshot. In this example, `w` between 90 and 120 seems to return a consistent result, whereas outside this size, either the upper or lower is poorly estimated.

Let us try to put all this together into an automated result. There is more to do, but hopefully its clear how far Bryan and I pushed these ideas.

Automated Baseline Subtraction

The steps here are to 1) select a sample, 2) scan for baseline cutoffs, 3) estimates the baseline, 4) produce a final thermogram on interpolated grid for further analysis. We have functions to do each step - we will use the trusty sample `1a` for the verbose walkthrough, then apply to all samples in an automated fashion.

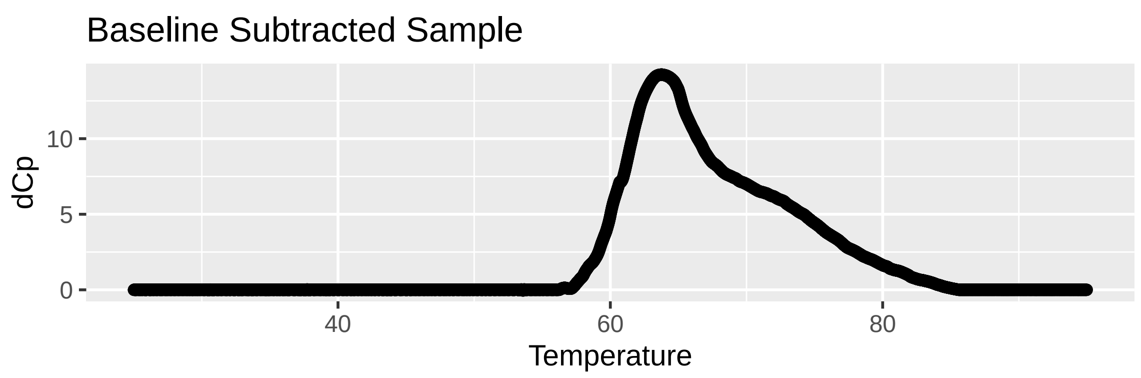
Sample 1a automated

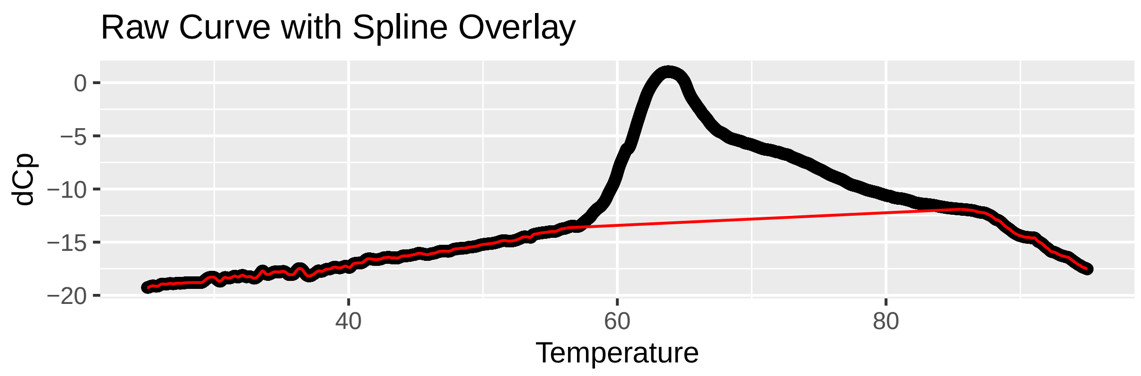
We know the answer, but here it is with the functions we have built above!

```
### select a sample
working.sample <- Urine.Working.Final %>% filter(SampleID == '1a') %>%
  select(Temperature, dCp)
### automate selection of endpoints
endpoints <- moving.window(
  x = working.sample,
  w = 90,
  exclusion.lwr = 60,
  exclusion.upr = 80
)
```

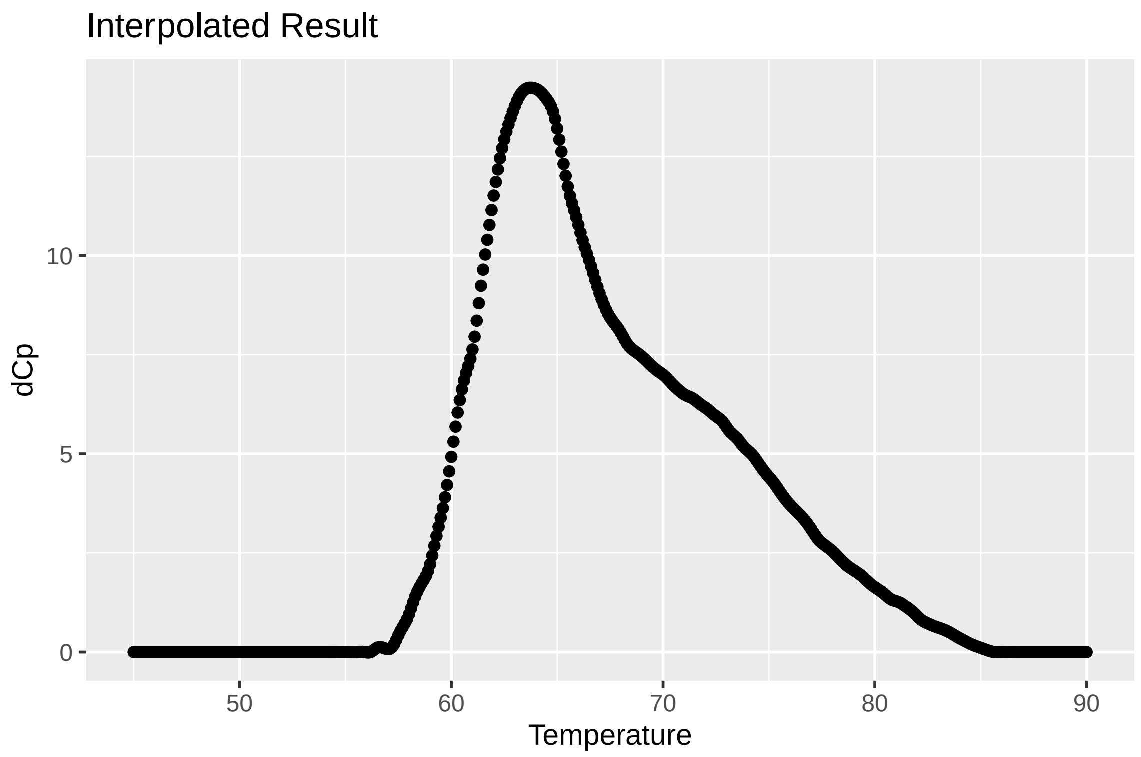
```
## Scanning Lower.
## Scanning Upper.
```

```
### baseline subtraction with auto-selected upr/lwr points
baseline.output <- baseline.subtraction.byhand(
  x = working.sample,
  lwr.temp = endpoints$lower,
  upr.temp = endpoints$upper)
```





```
### generate a final sample on chosen grid!
final.sample <- final.sample.interpolate(
  x = baseline.output,
  grid.temp = seq(45, 90, 0.1))
```



Not too shabby for our beloved Sample `1a`.

Automated Baseline Function!

We can now apply to this all samples in the urine data set. We will stick with `w = 90` and an exclusion window from `60` to `90`C. I will write one final function that can be used to automate a sample with one command rather than four.

```r
### auto.baseline
###
### @use
### Automatically determines a baseline for RAW thermograms and returns
### baseline subtracted set on a chosen temperature grid!
###
### @requires
### dplyr
###
### @inputs
### x : raw thermogram as data.frame with Temperature and dCp columns.
### w : window size (default 90, try others!)
### exclusion.lwr : lower temperature exclusion point
### exclusion.upr : upper temperature exclusion point
### grid.temp : chosen temperature grid for final data
###
### @function
auto.baseline <- function(x, w = 90, exclusion.lwr = 60, exclusion.upr = 80,
                          grid.temp = seq(45, 90, 0.1), plot.on = FALSE)
{
  ### automate selection of endpoints
  endpoints <- moving.window(
  x = working.sample,
  w = 90,
  exclusion.lwr = 60,
  exclusion.upr = 80)
### baseline subtraction with auto-selected upr/lwr points
  baseline.output <- baseline.subtraction.byhand(
  x = working.sample,
  lwr.temp = endpoints$lower,
  upr.temp = endpoints$upper,
  plot.on = plot.on)
### generate a final sample on chosen grid!
  final.sample <- final.sample.interpolate(
  x = baseline.output,
  grid.temp = seq(45, 90, 0.1),
  plot.on = plot.on)
### return the interpolated baseline-subtracted result
  return(final.sample)
}
```

Real quick ensure that the function does what it should for sample `1a`. Graphics are off by default but can be turned 'on' if desired.

```
### select a sample
working.sample <- Urine.Working.Final %>% filter(SampleID == '1a') %>%
  select(Temperature, dCp)
### get a baseline-subtracted and interpolated final result!
auto.output <- auto.baseline(x = working.sample)
```

```
## Scanning Lower.
## Scanning Upper.
```

```
head(auto.output)
```

```
##   Temperature           dCp
## 1        45.0 -3.008546e-07
## 2        45.1 -1.723452e-05
## 3        45.2 -3.556220e-06
## 4        45.3  2.343709e-05
## 5        45.4  3.900028e-05
## 6        45.5  2.159625e-05
```

Nice! Lets run it for every urine sample and store everything in a nice output file. If we wanted graphics, we might want to use the more piecemeal construction above to output the graphics to a PDF. Here I am producing a .csv that could be used for further analysis. This is likely slightly altered from what was sent earlier as I have improved how the endpoints are chosen from what Bryan originally tried.

Preparing CSV of Urine Samples

Ignoring for now the choice of signal vs. noise, a CSV file will be prepared for all automated baseline-subtracted urine samples. As they will all be on the sample grid, we need only store the dCp along with the sample names. This does a lot of work and if run will take significant time. The CSV file will be provided and can be used to prepare graphics after this loop is finished.

```
### Store all sample IDs and how many samples we need to analyze.
All.IDs <- Urine.Working.Final %>% pull(SampleID) %>% unique() %>% as.vector()
n.samples <- length(All.IDs)

### Final temperature grid
grid.temp <- seq(45, 90, 0.1)

### Setup frame to store all data generated
Urine.Final.Data <- data.frame(Temperature = grid.temp)

### loop over all samples and store the dCp with chosen temperature grid
for(j in 582:n.samples)
{
  cat('Working on Sample ', All.IDs[j], 'element ', j,' of', n.samples,' \n')
  ### select a sample
  working.sample <- Urine.Working.Final %>%
    filter(SampleID == All.IDs[j]) %>%
    select(Temperature, dCp)
  ### get a baseline-subtracted and interpolated final result!
  auto.output <- auto.baseline(x = working.sample, grid.temp = grid.temp)
  Urine.Final.Data <- Urine.Final.Data %>% cbind(out = auto.output$dCp)
  cat("\014")
}

colnames(Urine.Final.Data)[-1] <- All.IDs
### Only run the below once!!
# write.csv(x = Urine.Final.Data, file = 'generated_output/Final_Urine_Results.csv')
```

With everything finished, we can load the finalized urine baseline results and prepare a PDF with graphics for each scan. Lets see how we did (I expect there is still more to polish).

```
Urine_Auto_Final <- read.csv('generated_output/Final_Urine_Results.csv') %>% select(-X)
Sample_Names <- colnames(Urine_Auto_Final)[-1]

pdf('generated_output/Urine_Final_Auto_Graphics.pdf')
{
  for(j in 1:length(Sample_Names))
  {
    g1 <- Urine_Auto_Final %>% select(Temperature, Sample_Names[j]) %>%
      ggplot(aes(x = Temperature, y = .[,2])) +
      geom_point() +
      labs(title = paste0('Final Automated Sample for ', str_sub(Sample_Names[j], 2)))
    print(g1)
  }
}
dev.off()
```

The PDF will be provided through an alternative means since it is quite large. Many of the estimations are quite good, some are still kinda silly, and a few do miss the mark. We've certainly made progress with the automation, but there is likely more to try? Curious how the group will feel.

I have made the PDF available here: https://drive.google.com/file/d/1_ZK8qtqXrDWQSPnZK5xcJT00paFBmej1

Final Comments

The automation process is difficult but coming together. We have shown many ways to assess if a curve has signal vs noise, with no single method giving full agreement with the by-hand work that was done stating if a reading was signal or noise. The auto.arima technique gave very good sensitivity to detecting signal, less specificity for detecting noise.

The baseline technique has been coded in several different forms. There does now exist a function that could be used by a technician/laboratory member to work through the baseline fitting process in R. Additional steps were taken to automate the full procedure, but choice of endpoints for the baseline is still very tricky.

I would suggest the following work-flow.

- Run all samples through automated process
- Check resulting thermograms and mark those that need further follow-up
- Remove samples that are clearly noise from the set
- Use by-hand tools to determine the baseline for poorly represented curves

This would allow a significant chunk of baseline to be done quickly, while leaving the remainder to be done by a laboratory member more thoroughly.