Today I would like to demonstrate how to get automatic error propagation for your data analysis in R using the `errors` package by Iñaki Ucar et al. [1]
I would also like to demonstrate how to combine that with dimensional analysis (with automatic unit conversion) using the `units` package by Pebesma et al. [2]
This is usually known as *quantity calculus*.

To make it easy for you, dear reader, to follow along, we will take it from the start and assume all you have is a Windows 10 computer.
If you work on MacOS[1] or Linux, you can skip right ahead to "Working with measurement errors" (but you may have to adapt some of the steps, as they were written with Windows 10 in mind).

Most of the fundamental tools used for reproducible scientific work, such as R, knitr, git, Python, and not to be forgotten, the whole chain of document generation tools such as LaTeX, Markdown, and pandoc, "just work" on any Linux distribution, but can be a bit of a hassle on Windows.

It is not hard to see why that is. All these tools are available for free and with libre software licences. Windows is neither free nor libre, and has a long history of working against this academic model of software sharing.
These days, Microsoft, Google and other tech behemoths embrace open source software, simply because it allows them to monetise work without paying for it. Although, to be fair, they are also contributing back (to varying degrees) to this free/libre pool of software.
For an academic, open source tooling offers many tangible and intangible benefits, prime among them by making the entire work reproducible, and effectively future-proof.

So, let's get to it.
We will make use of the fact that Microsoft has added something called the Windows subsystem for Linux (WSL) on Windows 10, which allows you to run a (nearly feature-complete) Linux shell.
In fact, recently Microsoft released WSL version 2 (WSL2), which attempts to bring the Linux shell natively to the Windows 10 desktop.

In order to work with R, the next sections will show you how to make use of WSL2 to install Ubuntu 20.04, and then we will walk through installing R and RStudio Server using WSL2 Ubuntu. You will then be able to run R (in RStudio) from a browser window on your Windows desktop. Pretty cool, right?

With that in place, we will explore how to use R and the suite of `r-quantities` packages to seamlessly manipulate vectors that have associated measurement uncertainties *and* units through the steps of your data analysis, by way of an example using a subset of data from a real experiment in the Edvinsson lab.

Install WSL2 on Windows 10

In the near future, installing Ubuntu on Windows will be as simple as `wsl --install` (it's in the development version of Windows 10), but for now, we have to follow the steps below to achieve the same.

Open Powershell as administrator, then enable WSL1:

```
dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart
```

To enable WSL2, we must first enable the Windows virtual machine (VM) feature:

```
dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart
```

Now you have to restart Windows.

Next, download the WSL2 Linux kernel update package for x64 (or ARM64, depending on your machine), and install it.

Then open Powershell as administrator again, and set WSL2 as default for all future VMs:

```
wsl --set-default-version 2
```

Now it is time to install our Linux distribution of choice (i.e., Ubuntu) from the Microsoft Store. Open the Microsoft Store, find Ubuntu, and install it.
(And also, by the way, the Microsoft Store is awful. Such nasty underhanded nudging by Microsoft to create a wholly unnecessary “account”).

Anyway, you just installed Ubuntu on your Windows machine, congratulations!
Now you get to create your Linux username and password: simply start the app `Ubuntu` from the Start menu.

Parenthesis: if your Windows 10 is itself a virtual machine

This is unlikely to apply to 99% of my readers, so just skip ahead to the next section.

Who does this section apply to? Well, if you, like me, run Windows 10 as a VM on top of your regular OS, you will need to make sure that a) your host machine (layer 0) supports nested virtualisation, and b) that your hypervisor has nested virtualisation enabled.

Details on how for the KVM hypervisor on a Linux host in footnote.[2]

Parenthesis: managing and rebooting your WSL2 virtual machines

Closing the Ubuntu shell in Windows does not restart the Ubuntu VM, and sometimes it can be useful to effect a reboot.
So how do we do that?

It’s not obvious,
but the `wsl` executable has two useful attributes,

```
wsl --list --verbose
wsl --terminate Ubuntu-20.04
```

that allows you to list/view all installed VMs (running or not), and then you simply `--terminate` to shut down,
and `wsl -d Ubuntu-20.04` to start again. Starting the VM can also be achieved from the app link in the Start menu,
assuming the VM has one.

Install R and RStudio server on WSL2 Ubuntu

With a complete Ubuntu shell in place, we can now go about our business and install R.
In your Ubuntu shell, add the public key of the CRAN repository, then add the Ubuntu 20.04 CRAN repo of R version 4.x to your package manager's list of sources:

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys E298A3A825C0D65DFD57CBB651716619E084DAB9
echo "deb https://cran.uni-muenster.de/bin/linux/ubuntu focal-cran40/" | sudo tee /etc/apt/sources.list.d/r-project.list
```

Note that you can change the domain to any one of the CRAN mirrors if you want.
At this point, remember to tell `apt` about the new source by:

```
sudo apt update
```

Install R and its dependencies using the `aptitude` package manager
(we will also install `libudunits2-dev`, which is a C library and utility for handling physical units that the R package `units` depends on).

```
sudo apt install r-base r-base-core r-recommended r-base-dev gdebi-core build-essential libcurl4-gnutls-dev libxml2-dev libssl-dev libudunits2-dev
```

This installs about 400 MB of software in a jiffy. That's R!

Now download the latest RStudio Server package file for Ubuntu 18+ and install it:

```
wget https://rstudio.org/download/latest/stable/server/bionic/rstudio-server-latest-amd64.deb
sudo gdebi rstudio-server-latest-amd64.deb
```

Great! Now all you got to do is launch your RStudio Server:

```
sudo rstudio-server start
```

With that, you can open your browser (Chromium/Chrome recommended) at http://localhost:8787
and enjoy your very own instance of RStudio Server.
(You sign in with the same username and password as in your Ubuntu Linux VM).

And with that, all the preparatory steps are done.[3]
We can now start working in our data analysis environment in R.

Working with quantities and measurement errors

The `quantities` package (which ties together the capabilities of
the `units` and `errors` packages) has an excellent vignette on quantity calculus in R.

The `errors` package is available on CRAN, and its git repo is published on Github.
The package is also presented by its authors in an R Journal article from 2018, and there is also an informative blog post by the main author for the very first release of the package.

The `units` package handles physical quantities
(values with associated units) in R,
and builds on UNIDATA's `udunits-2` C library of
over 3000 supported units, making it a very robust implementation while avoiding the sin
of re-inventing the wheel. It does lead to a dependency on the system-level package `udunits2`

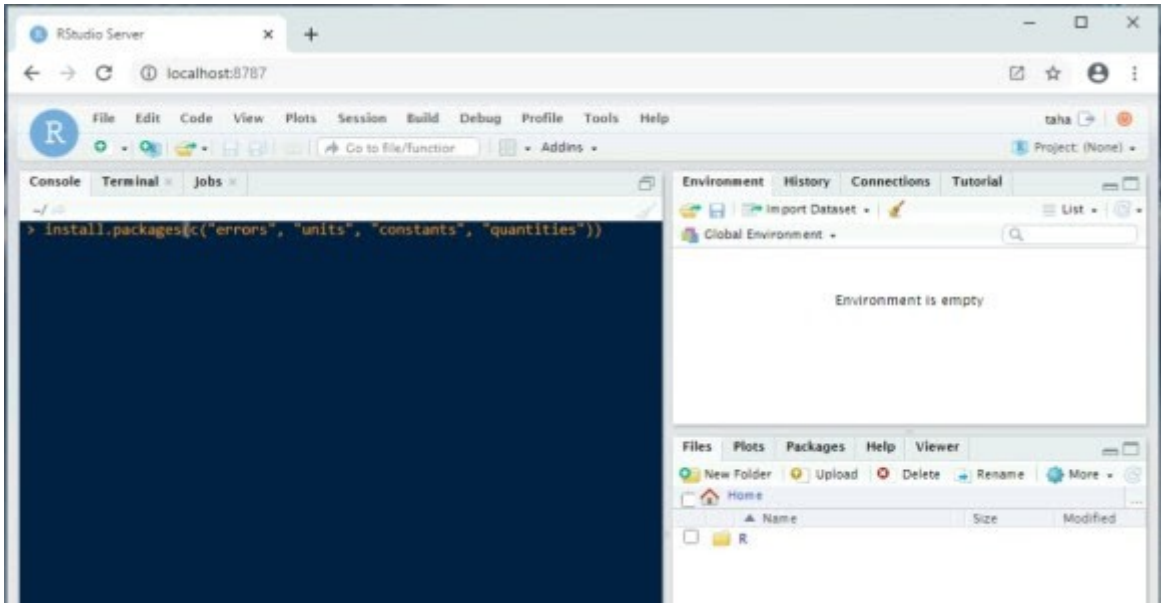
however, making it very hard to install and work with `units` on Windows.

I think that these packages have finally brought R up to par with Python with regards to quantity calculus, which is a crucial aspect for any scientist or engineer handling physical measurement values.
Although it should be noted that the `uncertainties` package in Python is more mature (v1.0 was released in 2010 whereas the `errors` package in R still has not achieved that milestone), the `errors` package is very usable, indeed I used it for practically all the analysis steps of the latest paper I co-authored (I even managed to cite the package). [3]

Let's take the `errors` and `units` packages out for a spin!
Since this is a clean R installation, first we will need to install these packages and any other packages we would like to have available.

Install necessary R packages

Let's install all the `r-quantities` packages, and a subset of the tidyverse packages (feel free to install any other packages you may like).




```
install.packages(c("errors", "units", "constants", "quantities"))
install.packages(c("dplyr", "magrittr", "ggplot2", "ggforce", "tibble"))
install.packages("remotes")
remotes::install_github("solarchemist/R-common")
remotes::install_github("solarchemist/oceanoptics")
```

(This operation may take a minute).

Now let's load all the packages we will need and get to work on some real-life data.

```
library(errors)
library(constants)
library(units)
library(quantities)
library(common)
library(oceanoptics)
library(dplyr)
library(magrittr)
```

```
library(ggplot2)
library(ggforce)
```

Back to work

For the sake of keeping this post short, I will spare you the legwork of reading the individual spectra files into a dataframe. I have done that and saved to resultant dataframe to an rda-file which has been uploaded to a publicly accessible URL.
Let's import the data to a variable, and have a quick look at the dataframe:

```
# read the MB abs spectra from the public URL
df.MB <-
   common::LoadRData2Variable(url="https://public.solarchemist.se/data/uvvis-abs-MB-conc-series.rda")
df.MB %>% glimpse()
## Rows: 28,672
## Columns: 13
## $ sampleid            "H02AB", "H02AB", "H02AB", "H02AB", "H02AB", "H02AB", …
## $ range               "001", "001", "001", "001", "001", "001", "001", "001"…
## $ wavelength          188.76, 189.23, 189.70, 190.17, 190.64, 191.12, 191.59…
## $ intensity           0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000…
## $ IntegrationTime     "1000", "1000", "1000", "1000", "1000", "1000", "1000"…
## $ n_Averaged          "10", "10", "10", "10", "10", "10", "10", "10", "10", …
## $ Boxcar              "0", "0", "0", "0", "0", "0", "0", "0", "0", "0", "0",…
## $ CorrElectricDark    "No", "No", "No", "No", "No", "No", "No", "No", "No", …
## $ StrobeLampEnabled   "No", "No", "No", "No", "No", "No", "No", "No", "No", …
## $ CorrDetectorNonLin  "No", "No", "No", "No", "No", "No", "No", "No", "No", …
## $ CorrStrayLight      "No", "No", "No", "No", "No", "No", "No", "No", "No", …
## $ n_Pixels            "2048", "2048", "2048", "2048", "2048", "2048", "2048"…
## $ conc                50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50…
```

As you can see, it is raw data from the spectrometer, and it comes with neither measurement errors nor units.
(Unfortunately, it is still all-to-common for lab instruments to export data without such attributes).

It is perfectly acceptable to add *estimated* standard errors to this raw data
(we will add units at the same time using the `set_quantities()` function of the `quantities` package).

```
# specify unit and error for the measured concentration values
```

```
df.MB$conc <-
   set_quantities(
      df.MB$conc * 1e-6,
      "mol/L",
      # set relative error 2% (estimate)
      df.MB$conc * 1e-6 * 0.02)
# specify unit and error for wavelength values to instrumental resolution
df.MB$wavelength <-
   set_quantities(df.MB$wavelength, "nm", 1.5)
# specify error for absorbance values to 1% rel error (estimate)
df.MB$intensity <-
   # absorbance is unitless, or more precisely, has a unit of "1"
   set_quantities(df.MB$intensity, "1", df.MB$intensity * 0.01)
# add a column describing the solvent (H2O or H2O+EtOH)
df.MB$solvent <- "H2O"
df.MB$solvent[grep("+EtOH$", df.MB$sampleid)] <- "H2O+EtOH"
```

For plotting, we will use the package `ggplot2`, which is like a cousin to the `dplyr` data manipulation package. Both come with a sort of “coding grammar”, which is very comfortable to use once you get used to it.
In my opinion it is much easier to express chains of manipulations using this grammar (you can find plenty of examples in the code below).

We should note that ggplotting two columns that both are `units` objects **will fail** with `Error in Ops.units(): both operands of the expression should be "units" objects`, due to the `units` package (rightfully) determining that they don’t have the same units (or units that can be converted into each other).
The (very elegant) solution to this finicky behaviour is provided by the `ggforce` package.
All we have to do is make sure we load `library(ggforce)`, and then we can get back to using ggplot2 as usual (mostly as usual, you will notice that we now use a `scale_*_unit()` function instead of the usual `scale_*_continuous()`).

```
ggplot(
   df.MB %>%
      filter(wavelength %>% as.numeric() >= 250) %>%
      filter(wavelength %>% as.numeric() <= 850)) +
   facet_wrap(~solvent) +
   geom_errorbar(
      colour = "red",
      aes(x = wavelength,
          ymin = errors_min(intensity),
          ymax = errors_max(intensity))) +
   geom_line(aes(x = wavelength, y = intensity, group = sampleid)) +
   scale_x_unit(
      breaks = seq(300, 800, 100),
      expand = expansion(0.03, 0)) +
   scale_y_unit(expand = expansion(0.03, 0)) +
   labs(x = "Wavelength", y = "Abs") +
   theme(legend.position = "none")
```

Recorded UV-Vis spectra of MB in water and MB in water with 10 vol% EtOH (with the estimated measurement error of the absorbances shown as vertical error bars in red).

Figure 1: Recorded UV-Vis spectra of MB in water and MB in water with 10 vol% EtOH (with the estimated measurement error of the absorbances shown as vertical error bars in red).

To check whether a unit already exists in the `udunits2` library, I like to use the `udunits2::ud.is.parseable()` function.
If you know of a better way, please let me know!

Now, as you have surely already noticed, this dataset is simply a series of UV-Vis absorbance spectra at varying concentrations
(of methylene blue in water and water with some ethanol, in this case).

Here's how the data looks at this point (after we set errors and/or units for some columns):

```
df.MB %>% glimpse()
## Rows: 28,672
## Columns: 14
## $ sampleid            "H02AB", "H02AB", "H02AB", "H02AB", "H02AB",
"H02AB", …
## $ range               "001", "001", "001", "001", "001", "001", "001",
"001"…
## $ wavelength         (err) /nm 189(2) /nm, 189(2) /nm, 190(2) /nm, 190(2)
/nm, 19…
## $ intensity          (err) /1 0(0) /1, 0(0) /1, 0(0) /1, 0(0) /1, 0(0) /1,
0(0) /…
## $ IntegrationTime     "1000", "1000", "1000", "1000", "1000", "1000",
"1000"…
## $ n_Averaged          "10", "10", "10", "10", "10", "10", "10", "10",
"10", …
## $ Boxcar              "0", "0", "0", "0", "0", "0", "0", "0", "0", "0",
"0",…
## $ CorrElectricDark    "No", "No", "No", "No", "No", "No", "No", "No",
"No", …
## $ StrobeLampEnabled   "No", "No", "No", "No", "No", "No", "No", "No",
"No", …
## $ CorrDetectorNonLin  "No", "No", "No", "No", "No", "No", "No", "No",
"No", …
## $ CorrStrayLight      "No", "No", "No", "No", "No", "No", "No", "No",
"No", …
## $ n_Pixels            "2048", "2048", "2048", "2048", "2048", "2048",
"2048"…
## $ conc               (err) /mol*L^-1 5.0(1)e-5 /mol*L^-1, 5.0(1)e-5
/mol*L^-1, 5.…
## $ solvent             "H2O", "H2O", "H2O", "H2O", "H2O", "H2O", "H2O",
"H2O"…
```

Most columns are of type "character", but note how `$wavelength` (and two other columns) say something else in that field. We can take a closer look at the structure of the dataframe using the `str()` function:

```
df.MB %>% str()
```

```
## 'data.frame':    28672 obs. of  14 variables:
##  $ sampleid          : chr  "H02AB" "H02AB" "H02AB" "H02AB" ...
##  $ range             : chr  "001" "001" "001" "001" ...
##  $ wavelength        : Units+Errors: /nm num  189(2) 189(2) 190(2) 190(2)
191(2) ...  $ intensity         : Units+Errors: /1 num  0(0) 0(0) 0(0) 0(0)
0(0) 0(0) 0(0) 0(0) 0(0) 0(0) ...  $ IntegrationTime   : chr  "1000" "1000"
"1000" "1000" ...
##  $ n_Averaged        : chr  "10" "10" "10" "10" ...
##  $ Boxcar            : chr  "0" "0" "0" "0" ...
##  $ CorrElectricDark  : chr  "No" "No" "No" "No" ...
##  $ StrobeLampEnabled : chr  "No" "No" "No" "No" ...
##  $ CorrDetectorNonLin: chr  "No" "No" "No" "No" ...
##  $ CorrStrayLight    : chr  "No" "No" "No" "No" ...
##  $ n_Pixels          : chr  "2048" "2048" "2048" "2048" ...
##  $ conc              : Units+Errors: /mol*L^-1 num  5.0(1)e-5 5.0(1)e-5
5.0(1)e-5 5.0(1)e-5 5.0(1)e-5 5.0(1)e-5 5.0(1)e-5 5.0(1)e-5 5.0(1)e-5
5.0(1)e-5 ...  $ solvent           : chr  "H2O" "H2O" "H2O" "H2O" ...
```

Having thus set suitable estimates (or perhaps even properly calculated) errors for the measured quantities *wavelength*, *absorbance*, and *concentration*, we will use this data to calculate the spectral absorption coefficient of MB in the two solvents we used.

As an introductory exercise, let's define the wavelength of the strongest absorption band along with its unit and a synthetic standard error value:

```
wl.max.abs <- set_quantities(665, "nm", 5)
```

We can change the notation of the error from `parenthesis` to `plus-minus` when printing a quantity:

```
print(wl.max.abs, notation="plus-minus", digits=1)
## 665 ± 5 /nm
```

> By default, `errors` uses *parenthesis* notation (which is more compact) and a single significant digit for errors.

We can also print more digits for the error part by setting `digits`.
Both of these settings can also be set globally (for our entire R session) by using the `options()` function, like this:

```
options(errors.notation="plus-minus", errors.digits=2)
```

Before we get into calculating the spectral absorption coefficients, recall that the Beer-Lambert law

$$\begin{equation}
A = \epsilon lc \;\Longleftrightarrow\; \epsilon = \frac{A}{lc}
\end{equation}$$

includes the optical path length of the cuvette, $l=\SI{1}{\cm}$ in our case.
Let's set that as a variable so we can use it when calculating ϵ:

```
# I don't know the tolerances of the cuvette dimensions, (quite the
oversight, I know)
# but since these cuvettes are mass-produced I am assuming they are pretty
tight and
```

```
# so estimating the error at 1/10 mm
optical.length <- set_quantities(1, "cm", 0.01)
```

With a series of UV-Vis absorbance spectra at varying concentrations, we can of course plot the linear relationship between \(A \propto c\) at a particular wavelength (such as the main absorbance peak), but
for the sake of this exercise we will take the calculation one step further and calculate \(\epsilon(\lambda, A, c)\), i.e., the absorption coefficient for all wavelengths.
It is not pretty code, but it gets the job done. Here goes:

```
# loop over each spectrum by solvent
solvents <- df.MB %>% pull(solvent) %>% unique()
# beware, unique() silently drops all quantity attributes
wl.steps <- df.MB %>% filter(sampleid == unique(df.MB$sampleid)[1]) %>%
pull(wavelength)
# hold the results of the loop in an empty dataframe, pre-made in the right
dimensions
# complication! quantities cols must use the same (or convertable) units from
the start
abs.coeff <-
   data.frame(
      wavelength = rep(wl.steps, length(solvents)),
      solvent = "",
      # slope of linear fit
      k = rep(set_quantities(1, "L/mol", 1), length(solvents) *
length(wl.steps)),
      # intercept of linear fit
      m = rep(set_quantities(1, "1", 1), length(solvents) *
length(wl.steps)),
      # calculated abs coeff
      coeff = rep(set_quantities(1, "L*cm^-1*mol^-1", 1), length(solvents) *
length(wl.steps)),
      rsq = NA, # R-squared of linear fit
      adj.rsq = NA) # adj R-squared of linear fit
i <- 0
# Fair warning: this loop is horribly slow, and may take several minutes to
complete!
for (s in 1:length(solvents)) {
   message("Solvent: ", solvents[s])
   # varying conc of a particular solvent
   for (w in 1:length(wl.steps)) {
      # to keep track of the current row in abs.coeff, we will use a counter
i
      i <- i + 1
      message("[", i, "] ", solvents[s], " :: ", wl.steps[w], " nm")
      # temporary df which we will use to calculate linear fits (abs.coeff)
for each wl.step
      intensity.by.wl <-
         df.MB %>%
         filter(solvent == unique(df.MB$solvent)[s]) %>%
         # even though we drop the errors when filtering, it does not affect
the returned rows
```

```
         filter(as.numeric(wavelength) == unique(as.numeric(df.MB$
wavelength))[w]) %>%
         select(sampleid, wavelength, intensity, conc, solvent)
      # now we can calculate abs coeff for this solvent at the current
wavelength
      this.fit <-
         # now this is the really tricky one
         # https://github.com/r-quantities/quantities/issues/10
         qlm(data = intensity.by.wl, formula = intensity ~ conc)
       # save R-squared and adjusted R-squared
      abs.coeff$rsq[i] <- summary(this.fit)$r.squared
      abs.coeff$adj.rsq[i] <- summary(this.fit)$adj.r.squared
      # save coefficients of linear fit
      abs.coeff$m[i] <- coef(this.fit)[[1]]
      abs.coeff$k[i] <- coef(this.fit)[[2]]
      abs.coeff$coeff[i] <- coef(this.fit)[[2]] / optical.length
      # solvent
      abs.coeff$solvent[i] <- unique(intensity.by.wl$solvent)
   }
}
save(
   abs.coeff,
   file = "/media/bay/taha/sites/hugo/solarchemist/static/assets/data/uvvis-
abscoeff-MB-conc-series.rda")
# read the calculated MB abs coeff from the public URL
abs.coeff <-
   LoadRData2Variable(url="https://public.solarchemist.se/data/uvvis-abscoeff-MB-conc-series.
rda")
```

Well, that was quite a workout![4] But now that we have saved the calculated absorption coefficients to an rda-file on disk, we won't have to repeat that calculation (for a while at least).
But the proof is in eating the pudding, so let's plot the absorption coefficients we got.

```
ggplot(
   abs.coeff %>%
      filter(wavelength %>% as.numeric() >= 250) %>%
      filter(wavelength %>% as.numeric() <= 850)) +
   geom_errorbar(
      colour = "red",
      aes(x = wavelength,
          ymin = errors_min(coeff),
          ymax = errors_max(coeff))) +
   geom_line(aes(x = wavelength, y = coeff, group = solvent)) +
   scale_x_unit(
      breaks = seq(300, 800, 100),
      expand = expansion(0.03, 0)) +
   scale_y_unit(expand = expansion(0.03, 0)) +
   labs(x = "Wavelength", y = "Abs. coeff.") +
   theme(legend.position = "none")
```

Calculated spectral absorption coefficient for MB in water and MB in water with 10 vol% EtOH. With the

estimated standard error due to propagated measurement errors of the underlying quantities (red errorbars).

Figure 2: Calculated spectral absorption coefficient for MB in water and MB in water with 10 vol% EtOH. With the estimated standard error due to propagated measurement errors of the underlying quantities (red errorbars).

That looks good. Note how the vertical errorbars of the two curves (different solvents) overlap completely at certain spectral regions, notably at the main absorbance peak.

And not only did our entire analysis automatically propagate the measurement errors, we also managed to do the same for the units, doing automatic dimensional analysis at the same time (note how the units in the plot axes labels is automatically derived from the data).

That's quantity analysis, in R, with real-life data!
I hope you learned something new, and that this will inspire you to use quantity analysis for your future work.

But what about Python?

I feel it would be bad form to end this post without even mentioning Python's support for error propagation and dimensional analysis, which far predates R's.

Python has a whole array of packages to choose from for working with physical units, but as far as I can tell none of them build on the UNIDATA database.

However, the `pint` package appears to be the most popular by far (with 1200 stars on Github, it's actually an order of magnitude more popular than all the related R packages), and it integrates with both NumPy and the `uncertainties` package and Pandas.

Support for errors and error propagation has been around since 2010 courtesy of the `uncertainties` package by Eric O. Lebigot.

I am not aware of any package for integrating between the `uncertainties` and `pint` packages, but perhaps that is not necessary in Python. I'm not sure on that point.
is a little lacking.

In any case, getting up and running with Python is fairly easy.
In fact, Python 3.x comes pre-installed on the Ubuntu shell we just configured.
You should be aware, that you could just *write and execute* your Python code directly from RStudio Server (or R in general) using the `reticulate` package for R.

But if you prefer working with Python in its own IDE, you can actually install Jupyter Notebook and work from a browser, just like for RStudio Server. Just follow these steps:

In the WSL Ubuntu shell, run the following commands to install Jupyter Notebook:

```
sudo apt install python3-pip python3-dev
sudo -H pip3
sudo -H pip3 install --upgrade pip
sudo -H pip3 install jupyter
```

then run `jupyter notebook` and copy-paste the provided URL into your browser.

> Note: for serious work, you should look into installing JupyterLab and/or JupyterHub instead of Jupyter Notebook.

This template for a student lab report
I co-authored way back when uses both `uncertainties` and `numpy` for its calculations, and may be of interest.

Wrapper function redefining `lm()` to support units/errors

```
# wrap lm() to drop quantities and get around error when calling
summary(lm())
# https://www.r-spatial.org/r/2018/08/31/quantities-final.html
qlm <- function(formula, data, ...) {
   # get units info, then drop quantities
   row <- data[1, ]
   for (var in colnames(data)) if (inherits(data[[var]], "quantities")) {
      data[[var]] <- drop_quantities(data[[var]])
   }

   # fit linear model and add units info for later use
   fit <- lm(formula, data, ...)
   fit$units <- lapply(eval(attr(fit$terms, "variables"), row), units)
   class(fit) <- c("qlm", class(fit))
   fit
}
# get quantities attributes back for the coef method
# (which is the only class we care about in this demonstration)
coef.qlm <- function(object, ...) {
  # compute coefficients' units
  coef.units <- lapply(object$units, as_units)
  for (i in seq_len(length(coef.units) - 1) + 1)
    coef.units[[i]] <- coef.units[[1]] / coef.units[[i]]
  coef.units <- lapply(coef.units, units)

  # use units above and vcov diagonal to set quantities
  coef <- mapply(set_quantities, NextMethod(), coef.units,
                 sqrt(diag(vcov(object))), mode="symbolic", SIMPLIFY=FALSE)

  # use the rest of the vcov to set correlations
  p <- combn(names(coef), 2)
  for (i in seq_len(ncol(p)))
    covar(coef[[p[1, i]]], coef[[p[2, i]]]) <- vcov(object)[p[1, i], p[2, i]]

  coef
}
```

sessionInfo()

```
sessionInfo()
## R version 3.6.2 (2019-12-12)
```

```
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 18.04.5 LTS
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.7.1
## LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.7.1
##
## locale:
##  [1] LC_CTYPE=en_GB.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_GB.UTF-8        LC_COLLATE=en_GB.UTF-8
##  [5] LC_MONETARY=en_GB.UTF-8    LC_MESSAGES=en_GB.UTF-8
##  [7] LC_PAPER=en_GB.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_GB.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] ggforce_0.3.2          ggplot2_3.3.2          magrittr_1.5
##  [4] dplyr_1.0.2            oceanoptics_0.0.0.9004 common_0.0.2
##  [7] quantities_0.1.5       units_0.6-7            constants_0.0.2
## [10] errors_0.3.4           knitr_1.29
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_1.0.4.6     highr_0.8        compiler_3.6.2   pillar_1.4.6
##  [5] tools_3.6.2      digest_0.6.27    evaluate_0.14    lifecycle_0.2.0
##  [9] tibble_3.0.4     gtable_0.3.0     pkgconfig_2.0.3  rlang_0.4.8
## [13] cli_2.1.0        yaml_2.2.1       blogdown_0.20    xfun_0.15
## [17] withr_2.3.0      stringr_1.4.0    generics_0.1.0   vctrs_0.3.4
## [21] grid_3.6.2       tidyselect_1.1.0 glue_1.4.2       R6_2.5.0
## [25] fansi_0.4.1      rmarkdown_2.3    bookdown_0.20    polyclip_1.10-0
## [29] farver_2.0.3     purrr_0.3.4      tweenr_1.0.1     scales_1.1.1
## [33] htmltools_0.5.0  ellipsis_0.3.1   MASS_7.3-51.6    assertthat_0.2.1
## [37] colorspace_1.4-1 labeling_0.4.2   utf8_1.1.4       stringi_1.4.6
## [41] munsell_0.5.0    crayon_1.3.4
```