This post is Part I of a dive into the contents of the Apple Health Export.  
It will work through the mechanics of moving data from the Apple Health app  
out of your iPhone and into R where you can analyze it. It also will describe in detail the problem of  
adjusting the time stamps for daylight savings time and travel across time zones.  
Unfortunately the topic of time zones and the Apple Health Export is a complicated subject.

**First, Export the Data from the Health App**

You can export all of the data you are able to view via the Health app.  
Open the Health app on your iPhone.  
To export, go to your personal settings by  
clicking on your icon near the upper right corner of the Browse screen.  
(See the the first screenshot below.)  
Click on the icon and you will see some of your personal settings.  
You will need to scroll to the bottom of this page, where you will  
see a clickable line “Export All Health Data”, as shown in the second screenshot below.

Tap upper right icon:

Scroll to very bottom:

Once you click OK to go ahead with the export, it may take a significant amount of time.  
On my iPhone 8 it takes more than five minutes. Once it is complete, you’ll get a  
dialog that asks where to send the exported data. I use AirDrop to send it to the  
Mac where I am running RStudio. It ends up in the Downloads folder on that Mac.  
If you need to move the data to a Windows computer, you may need to send it  
via email or Dropbox.  
The exported file is named export.zip. If you double-click on that file it  
will expand into a folder called apple\_health\_export. The uncompressed file is huge  
in comparison with the size of the zip file. In my case, export.zip is about  
79 megabytes which becomes an apple\_health\_export folder that is 2.45 gigabytes!  
In my R code, I uncompress the file into my Downloads folder, which is excluded  
from my Time Machine backups.

**R Code to Expand the Export File and Import It As XML Data**

The R code below shows how to decompress export.zip and follow some  
basic steps to import it into R. I’m following in the footsteps of  
several people who have published code to accomplish these steps.

The R code uncompresses the zip file and replaces the apple\_health\_export folder.  
(Because of the size of this folder, I try to avoid having multiple copies and also  
avoid having it saved to my disk backup.)  
The big file inside that folder is export.xml. Following the examples cited above, I  
use the XML package to covert the major elements of the XML file into  
tidy data frames.

if (file\_exists("~/Downloads/export.zip")) {

rc <- unzip("~/Downloads/export.zip", exdir = "~/Downloads", overwrite = TRUE)

if (length(rc) != 0) {

# once unzipped, delete export.zip. Otherwise, the next time Air Drop sends export.zip

# to your mac it will be renamed as export2.zip and you may accidentally process

# an out-of-date set of data.

# takes a bit more than 20 seconds on my iMac

health\_xml <- xmlParse("~/Downloads/apple\_health\_export/export.xml")

# takes about 70 seconds on my iMac

health\_df <- XML:::xmlAttrsToDataFrame(health\_xml["//Record"], stringsAsFactors = FALSE) %>%

as\_tibble() %>% mutate(value = as.numeric(value))

activity\_df <- XML:::xmlAttrsToDataFrame(health\_xml["//ActivitySummary"], stringsAsFactors = FALSE) %>%

as\_tibble()

workout\_df <- XML:::xmlAttrsToDataFrame(health\_xml["//Workout"], stringsAsFactors = FALSE) %>%

as\_tibble

clinical\_df <- XML:::xmlAttrsToDataFrame(health\_xml["//ClinicalRecord"]) %>%

as\_tibble()

save(health\_xml, health\_df, activity\_df, workout\_df, clinical\_df,

file = paste0(path\_saved\_export, "exported\_dataframes.RData"))

if (file.exists("~/Downloads/export.zip")) file.remove("~/Downloads/export.zip")

}

}

I won’t go into the details of the XML structure of the health export.  
For most purposes, the Record, ActivitySummary, Workout, and Clinical data types  
will provide all that you are looking for. My expanded Apple Health Export  
folder also includes workout GPX files, electrocardiograms, and clinical  
records imported from my health system’s medical records system.

I have a bit over two years of Apple Watch data in my iPhone.  
After a full career working  
as a data analyst, this is the largest number of data points  
I have ever dealt with. Extracting the “Record” data from export.xml produces  
more than four million rows and takes about 70 seconds on my 2019 iMac.

**The Types of Data in the Export**

The counts by “type” describe the breadth and quantity of data are shown in Table [1](https://johngoldin.com/post/2020-02-15-apple-health-export1/#tab:freq-type).

| Table 1: Frequency by Type and Data Source | | | | | |
| --- | --- | --- | --- | --- | --- |
| **type** | **Watch** | **Phone** | **Lose It!** | **Other** | **Total** |
| HKQuantityTypeIdentifierActiveEnergyBurned | 1,459,787 | 0 | 0 | 1 | 1,459,788 |
| HKQuantityTypeIdentifierBasalEnergyBurned | 911,246 | 0 | 0 | 0 | 911,246 |
| HKQuantityTypeIdentifierDistanceWalkingRunning | 664,925 | 140,647 | 0 | 1 | 805,573 |
| HKQuantityTypeIdentifierHeartRate | 655,002 | 0 | 0 | 1,684 | 656,686 |
| HKQuantityTypeIdentifierStepCount | 55,350 | 136,637 | 0 | 1 | 191,988 |
| HKQuantityTypeIdentifierAppleExerciseTime | 53,944 | 337 | 0 | 0 | 54,281 |
| HKQuantityTypeIdentifierFlightsClimbed | 13,358 | 11,988 | 0 | 1 | 25,347 |
| HKCategoryTypeIdentifierAppleStandHour | 20,276 | 7 | 0 | 0 | 20,283 |
| HKQuantityTypeIdentifierAppleStandTime | 9,657 | 0 | 0 | 0 | 9,657 |
| HKQuantityTypeIdentifierEnvironmentalAudioExposure | 6,586 | 0 | 0 | 0 | 6,586 |
| HKQuantityTypeIdentifierDietaryFatTotal | 0 | 0 | 5,628 | 0 | 5,628 |
| HKQuantityTypeIdentifierDietaryEnergyConsumed | 0 | 0 | 5,064 | 0 | 5,064 |
| HKQuantityTypeIdentifierDietaryFatSaturated | 0 | 0 | 5,040 | 0 | 5,040 |
| HKQuantityTypeIdentifierDietaryProtein | 0 | 0 | 4,736 | 0 | 4,736 |
| HKQuantityTypeIdentifierDietarySodium | 0 | 0 | 4,699 | 0 | 4,699 |
| HKQuantityTypeIdentifierDietaryFiber | 0 | 0 | 4,695 | 0 | 4,695 |
| HKQuantityTypeIdentifierDietarySugar | 0 | 0 | 4,603 | 1 | 4,604 |
| HKQuantityTypeIdentifierHeartRateVariabilitySDNN | 4,573 | 0 | 0 | 0 | 4,573 |
| HKQuantityTypeIdentifierDietaryCholesterol | 0 | 0 | 4,447 | 0 | 4,447 |
| HKCategoryTypeIdentifierSleepAnalysis | 0 | 0 | 0 | 3,377 | 3,377 |
| HKQuantityTypeIdentifierBloodPressureDiastolic | 0 | 0 | 0 | 2,428 | 2,428 |
| HKQuantityTypeIdentifierBloodPressureSystolic | 0 | 0 | 0 | 2,428 | 2,428 |
| HKQuantityTypeIdentifierDistanceCycling | 1,415 | 0 | 0 | 0 | 1,415 |
| HKQuantityTypeIdentifierRestingHeartRate | 861 | 0 | 0 | 0 | 861 |
| HKQuantityTypeIdentifierWalkingHeartRateAverage | 781 | 0 | 0 | 0 | 781 |
| HKQuantityTypeIdentifierBodyMass | 0 | 1 | 263 | 1 | 265 |
| HKQuantityTypeIdentifierVO2Max | 187 | 0 | 0 | 0 | 187 |
| HKCategoryTypeIdentifierMindfulSession | 76 | 0 | 0 | 0 | 76 |
| HKQuantityTypeIdentifierHeadphoneAudioExposure | 0 | 17 | 0 | 0 | 17 |
| HKQuantityTypeIdentifierHeight | 0 | 1 | 0 | 1 | 2 |
| HKQuantityTypeIdentifierDietaryCaffeine | 0 | 0 | 0 | 1 | 1 |
| HKQuantityTypeIdentifierDietaryCarbohydrates | 0 | 0 | 0 | 1 | 1 |
| HKQuantityTypeIdentifierNumberOfTimesFallen | 1 | 0 | 0 | 0 | 1 |
| Total | 3,858,025 | 289,635 | 39,175 | 9,926 | 4,196,761 |

The same data may be repeated from multiple sources  
so it is important to pay attention to sourceName.  
Note that step counts, flights climbed, and distance walking/running  
are recorded from both the Watch and the iPhone.  
On any particular day you probably want to include only one of the sources.  
Otherwise you risk double counting.  
Generally I focus on the Watch data, but I have almost three years of  
data from my iPhone before I started wearing the Apple Watch.

The largest quantity of data is collected via my Apple Watch rather than my iPhone.  
There are other sources as well.  
I have been using the *Lose It!* app on my iPhone for about six months to count calories,  
and that produces a noticeable amount of data.  
The free version of the app that I am using does not display much beyond  
basic calorie counts.  
It’s interesting to see that the more detailed nutrition breakdowns are passed  
into the Health app.

As far as the “Other” category, I’m using an Omron blood pressure cuff  
that can transfer readings to the Omron app on the iPhone via Bluetooth. Those  
readings are then updated in the Apple Health database.  
There are a few odds and ends contributed by apps on my iPhone  
such as AllTrails, Breathe, and AutoSleep. Note that heart rate data comes both from  
the watch and from the blood pressure data.

The other major XML categories are ActivitySummary, Workout, and ClinicalRecord.  
For ActivitySummary I have one row per  
day which basically summarizes some of the intra-day activity data.  
Workout has one row per workout. If I were  
still running I would focus much more on that data frame.  
For each workout it shows type, distance, duration,  
and energy burned Most of my workouts are outdoor walks.  
Quite often I forget to end the workout when I end the walk,  
which certainly reduces the usefulness of the data.  
But I would imagine that for a runner or a swimmer or a cyclist,  
the Workout information would be interesting and useful.

ClinicalRecord is a bit tricky.  
I have set things up so that my health organization shares my health records with the  
Apple Health app. The count by type is in Table.

| **type** | **n** |
| --- | --- |
| AllergyIntolerance | 1 |
| Condition | 12 |
| DiagnosticReport | 63 |
| Immunization | 15 |
| MedicationOrder | 7 |
| MedicationStatement | 13 |
| Observation | 694 |
| Patient | 1 |
| Procedure | 7 |

In the clinical data frame there is a column called resourceFilePath that contains  
the path to a json dataset in the Apple Health Export/clinical records folder. Presumably  
this would allow you to retrieve items such as individual lab test results.  
I haven’t attempted to get into this data. I only know what’s available  
here because I can view it via the Apple Health app.

**The Problem of Time Zones and of Daylight Savings**

When I first looked at day by day data for resting heart rate I bumped into  
problems caused by the issue of time zones. I have about 800 rows of data of type  
HKQuantityTypeIdentifierRestingHeartRate which should be one per day.  
I quickly discovered I had several days where there were two values in a single day,  
and these were related to  
occasions when I traveled by air to a different time zone.

This leads to a long digression on the subject of computer time stamps and time zones.

**Attention Conservation Notice:** If you don’t care about time of day and can tolerate  
a few things like resting heart rate being a day off, then you can ignore the  
issues with the time stamps. Your life will be a lot simpler. You can basically skip  
the rest of this post and look forward to Part II.

Now we shall dive deep into the weeds.  
Each item in the records of the health dataset has a creation, start, and end time stamp.  
In the export dataset  
they appear as a character string that looks like this: “2019-04-10 08:10:34 -0500”.  
There is “-0500” on the end  
because as I write this local time is Eastern Standard Time which is five hours  
earlier than UTC (universal  
time code). At first I thought the time code offset at the end of the text string would take care of everything.  
In fact, it is useless.  
As near as I can tell, the internal data has no indication of time zone.[1](https://johngoldin.com/post/2020-02-15-apple-health-export1/#fn1)  
The UTC offset is attached to the date and time information when the data is exported.  
Every single time stamp in the exported dataset has the same “-0500” offset,  
which merely represent my local offset at the time the export was done.  
When I exported the data during Eastern Daylight Savings, all of the offsets appeared as “-0400”.  
In fact, the exported data has no information about time zone or daylight savings.  
One should think of the actual time stamp as being in universal time (UTC).  
The export creates a character string which displays the date and time  
in the time zone where and when the export is created and attaches an offset to UTC.

Here is a detailed example.  
When I go into the Activity app on my iPhone I see that it claims I started a walk in England  
on 9/1/2019 at 05:51:58. I’m not that much of an early riser.  
I know from my travel diary that I actually got started five hours later than that  
at 10:51:58 local time (British Summer Time) because I had to take a bus before  
I could start the walk. When I exported the workout data last October during  
Daylight Savings, the exported  
date showed “2019-09-01 05:51:58 -0400”. When I export the data now during standard  
time it shows “2019-09-01 04:51:58 -0500”. If I feed either of those  
character strings into lubridate::as\_datetime I get “2019-09-01 09:51:58 UTC”.  
In absolute terms, there’s no ambiguity about when the observation was added to the data.  
The ambiguity is the local time as it appeared on my watch at the moment the data was added.  
Sometimes that matters and sometimes it doesn’t.

When does it matter?  
A number of items like resting heart rate are recorded once per day.  
But because of time zone issues, you can end up with two on one day and none on another.  
Also, there may be situations where you want to look at patterns over the course of a day.  
At one point I wanted to look at whether there were periods when my heart rate was  
unusually high during normal sleeping hours.  
I looked for a heart rate above 120 between the hours of 11PM and 6AM.  
I got hits for when I was hiking in a different time zone because the  
time stamp appeared to be during those night time hours when in fact the local time  
was shifted five or seven hours because of the different time zone.

The time stamp issues are tricky. If these kinds of situations are not a problem for you,  
then ignore them and skip this section. Your life will be simpler.

I use the lubridate package to deal with the time stamps.  
R relies on a Unix-based standard for dates and time called  
[POSIX](https://en.wikipedia.org/wiki/POSIX) that is implemented as a class  
called POSIXct. You can see lots of references to POSIXct in the lubridate documentation.  
The as\_datetime function in lubridate allows you to add a tz parameter that  
specifies the time zone.  
A significant difficulty is that in R the time zone is stored as an *attribute* of the vector  
rather than as part of the data.  
If you have a vector of datetime data,  
the time zone attribute applies to the entire vector, not to individual elements  
in the vector.  
If you want to store time zones that vary within the vector, you need to store them in a separate  
vector, and that’s not part of the R standard for handling dates and times.  
You’re on your own. The lubridate package includes some functions to help convert vectors  
from one time zone to another and to deal somewhat with  
daylight savings.  
But it does not automatically help with a vector that contains datetime information from  
varying time zones (as well as different daylight savings issues).

As I searched the web for tips on how to approach this issue, I discovered that  
there’s a population of people who are working hard to maintain a streak  
in filling their activity rings in the Apple Activity app.  
Some of those individuals get frustrated because they are tripped up  
by movement across time zones or even changes to daylight savings.

**My Treatment of Time Zones and the Apple Health Export**

Here I describe the way I handled the time zone issue.  
There will be three steps.  
1. Identify when I was in a different time zone.  
2. Attach a time zone to each row of health\_df.  
3. Use the time zone information to attach a local start and end time to each observation.

The first task is to figure out when I was outside of my home time zone.  
To do that I need a data frame that shows me the time when  
I arrived in another time zone.  
I created a table that contains the date and time my  
watch switched to a new time zone.  
In my case that means identifying airplane trips  
that landed in a different time zone.  
As a bonus, I will describe in detail how I  
used data exported from TripIt to create most  
of the table of time zone changes. Later I converted  
that information to UTC and then used the table to  
establish the time zone in each of my 4+ million rows  
of data from the Apple Health Export.

My TripIt data is missing one overseas trip. I needed to  
add that trip manually. Here is a tribble that creates a table  
to describe a trip to Greece via a two day stop  
in Amsterdam. If you have no  
data in TripIt then the manual table would have to  
describe all your trips to a different time zone.

manual\_timezone\_changes <-

tibble::tribble(

~local\_arrive, ~local\_timezone,

"2018-04-18 09:15:00", "Europe/Amsterdam",

"2018-04-20 21:00:00", "Europe/Athens",

"2018-04-30 15:23:00", "America/New\_York"

) %>%

mutate(local\_arrive = as\_datetime(local\_arrive))

The local\_arrive column records when I arrived in a new time zone and  
is the scheduled arrival time for the fight to that time zone.  
For example, the first line represent a flight that I took  
from JFK to Amsterdam that arrived at 9:15 the next morning  
in Amsterdam time. My watch was on New York time until I turned off  
airplane mode after arrival in Amsterdam so I am focusing on  
the arrival time. One  
could do similar lines of data for arrivals by car, train, or boat.  
Later we will see how to transform this data so that it can  
be applied to the rows of the Apple Health Export. First,  
let’s go into the details of how to complete this table  
using data from TripIt.

**Using TripIt Data to Track Your Plane Flights**

I will use TripIt to get most of my flight information.

OK, here we go. Remember that I warned you that adjusting the time  
stamps would involve complications.

TripIt API offers a very simple way of authenticating the API that should only be used for testing and development purposes…. Note that this authentication scheme is off by default for every TripIt user.

I sent them an email to request simple authentication, and TripIt support responded the next day.  
From then on I could use httr functions to get the data from TripIt via information  
from documentation of the TripIt API.

I did one GET call from httr to get a list of my TripIt trips. Next I used the purrr package  
to extract data from the nested JSON lists returned by TripIt. In particular, I used the map function  
to get TripIt “air” objects for each trip ID. Individual airplane flights are “segments”  
with each air object. For example, a trip might be two connecting flights to get to the  
destination and two flights to return home, each represented by a “segment”.  
I always feel like I’m a few steps away from thorough understanding of  
purrr and tend to rely on a certain amount of trial and error to get a sequence of flatten  
and map call that extract what I need.

I end up with a  
data frame that has the scheduled departure and arrival for each flight and conveniently provides  
the time zone for each airport. Note that in practice this data might not be perfect. It is scheduled  
flights only and would not account for cancelled flights or even the time of a delayed flight. So keep  
that in mind before you  
try to use this data to examine something detailed such as whether your heart rate is elevated during takeoffs and landings.

I took a quick detour and explored whether I could use the FlightAware API to get the actual arrival  
times. It is now easy to get free access to limited FlightAware data. But the API calls are  
oriented to retrieving current rather than historical data so I can’t use it to find out about  
my past flights.

Once I have the trip ID’s, I use trip ID to fetch the flight  
information. I used the RStudio View() function to examine  
the results I got back from calls to the TripIt API.  
I get one trip at a time, fetch the  
air segments, and then bind them together with purrr::map\_dfr. I  
fetched more than the minimum columns I needed partly out of curiosity over  
what was in the TripIt data. It’s interesting to see all my flight information  
in one table, although much is not directly relevant to the task at hand.

# get list of trips

trips\_list <- GET\_tripit("<https://api.tripit.com/v1/list/trip/past/true/false>")

trip\_ids <- trips\_list$Trip %>% map\_chr("id")

GET\_air <- function(trip\_id) {

atrip <-

GET\_tripit(

paste0(

"<https://api.tripit.com/v1/get/trip/id/>",

trip\_id,

"/include\_objects/true"

) )

air\_trip <- atrip[["AirObject"]][["Segment"]]

flights <- dplyr::tibble(

trip\_id = trip\_id,

trip\_start = atrip[["Trip"]][["start\_date"]],

start\_date = air\_trip %>% purrr::map("StartDateTime") %>% map\_chr("date"),

start\_time = air\_trip %>% purrr::map("StartDateTime") %>% map\_chr("time"),

start\_timezone = air\_trip %>% purrr::map("StartDateTime") %>% map\_chr("timezone"),

start\_city = air\_trip %>% purrr::map\_chr("start\_city\_name"),

end\_date = air\_trip %>% purrr::map("EndDateTime") %>% map\_chr("date"),

end\_time = air\_trip %>% purrr::map("EndDateTime") %>% map\_chr("time"),

end\_timezone = air\_trip %>% purrr::map("EndDateTime") %>% map\_chr("timezone"),

end\_city = air\_trip %>% purrr::map\_chr("end\_city\_name"),

airline = air\_trip %>% purrr::map\_chr("marketing\_airline"),

code = air\_trip %>% purrr::map\_chr("marketing\_airline\_code"),

number = air\_trip %>% purrr::map\_chr("marketing\_flight\_number"),

aircraft = air\_trip %>% purrr::map\_chr("aircraft\_display\_name"),

distance = air\_trip %>% purrr::map\_chr("distance"),

duration = air\_trip %>% purrr::map\_chr("duration")

)

}

GET\_air\_mem <- memoise::memoise(GET\_air)

# I used memoise because TripIt typically doesn't let me fetch

# all the data in a single call. With memoise, for each call

# I only fetch the items it doesn't remember.

flying <- trip\_ids %>% map\_dfr(GET\_air\_mem)

I used memoise with the GET\_air\_mem  
function because TripIt generally doesn’t give me all my flights  
on one try. It seems to limit how much data it will return.  
Because of memoise the Get\_air\_mem remembers the results  
of previous successful fetches from TripIt and only  
asks for things flights it doesn’t already know about.

I ended up with a tibble with one row per flight.  
I selected the two columns I needed and used bind\_rows to combine  
them with the manual data I created above.  
I focused on the ending time and location  
of each flight. That’s when I assume my watch gets changed to a new time zone and  
life in that time zone begins. Life in that time zone ends when life in the next time  
zone begins. I wrote a short function to convert the local scheduled time  
in the flight information into UTC time comparable to what is in the health export.

So after I bind together rows from TripIt and the manual rows above, I  
set utc\_until = lead(utc\_arrive), until\_timezone = lead(local\_timezone).  
I set the last utc\_until to now()

local\_to\_utc <- function(dt, timezone) {

# UTC of this local time. Note: not vectorized

if (is.character(dt)) dt <- as\_datetime(dt)

tz(dt) <- timezone

xt <- with\_tz(dt, "UTC")

tz(xt) <- "UTC"

return(xt)

}

arrivals <- flying %>%

mutate(local\_start = ymd\_hms(paste0(start\_date, start\_time)),

local\_arrive = ymd\_hms(paste0(end\_date, end\_time))) %>%

filter(start\_timezone != end\_timezone) %>% # only trips that change time zone matter

select(local\_arrive, local\_timezone = end\_timezone) %>%

# manual additions here

bind\_rows(manual\_timezone\_changes) %>%

mutate(utc\_arrive = map2\_dbl(local\_arrive, local\_timezone, local\_to\_utc) %>% as\_datetime) %>%

arrange(utc\_arrive) %>%

mutate(utc\_until = lead(utc\_arrive), until\_timezone = lead(local\_timezone))

arrivals$utc\_until[nrow(arrivals)] <- with\_tz(now(), "UTC")

arrivals$until\_timezone[nrow(arrivals)] <- Sys.timezone() # I don't really need this.

# Check that I have arrivals set up properly so that my last arrival is in my home time zone, otherwise warn

if (arrivals$local\_timezone[nrow(arrivals)] != Sys.timezone()) warning("Expected to end in Sys.timezone:",

Sys.timezone(), " rather than ",

arrivals$local\_timezone[nrow(arrivals)])

**Assign a Time Zone to Each Row of the Data**

The next step was to associate a time zone with each of the 4+ million rows in health\_df.  
I thought this would be difficult and very slow. I was wrong!  
The fuzzyjoin package  
by Dave Robinson  
came to the rescue. I can’t do a simple join between flights and data rows because there is  
not an exact match for the time stamps. Instead I want to join the time stamps in health\_df  
with a start and end time stamp for each row in the arrivals table. It turns out  
fuzzyjoin has that covered.

The fuzzyjoin package provides a variety of special joins that do not rely on an exact match.  
In this case, what I needed was the interval\_left\_join function which joins tables based  
on overlapping intervals. The help for interval\_left\_join explains that this function requires  
the IRanges package available from Bioconductor This was the first time I have used anything from the Bioconductor repository. I’m  
impressed by the speed of interval\_left\_join. I thought it would be impractical to run it  
on the full dataset, but it feels fast. I also used it to relate rows in  
health\_df to rows in workout\_df, but I’ll describe that in Part II.

health\_df <- health\_df %>%

mutate(utc\_start = as\_datetime(startDate),

utc\_end = as\_datetime(endDate)) %>%

filter(![is.na](http://is.na)(utc\_start)) %>%

interval\_left\_join(arrivals %>%

select(utc\_arrive, utc\_until, timezone = local\_timezone),

by = c("utc\_start" = "utc\_arrive", "utc\_start" = "utc\_until"))

Do a quick report to check whether the assignment of time zone makes sense.  
If data documenting the transition from one time zone to  
another is missing the whole table will be off kilter. If  
there’s something wrong with the arrivals table, it should show up as an  
unexpected result in Table.

# shows trip by trip to check whether timezone assignment works as expected:

# health\_df %>% #filter(timezone != "America/New\_York") %>%

# mutate(date = as\_date(utc\_start)) %>%

# group\_by(timezone, utc\_arrive) %>% summarise(arrive = min(utc\_start), leave = max(utc\_start),

# days = length(unique(date)),

# n = n())

# Do a quick check whether the distribution of time zones makes sense

health\_df %>% mutate(date = as\_date(utc\_start)) %>% group\_by(timezone) %>%

summarise(dates = length(unique(date)), observations = n()) %>%

kable(format.args = list(decimal.mark = " ", big.mark = ","),

table.attr='class="myTable"',

caption = "Frequency of Days by Time Zone")

| Table 2: Frequency of Days by Time Zone | | |
| --- | --- | --- |
| **timezone** | **dates** | **observations** |
| America/Chicago | 19 | 52,281 |
| America/Los\_Angeles | 15 | 74,973 |
| America/New\_York | 1,800 | 3,458,188 |
| America/Phoenix | 20 | 10,535 |
| Europe/Amsterdam | 3 | 5,105 |
| Europe/Athens | 11 | 19,555 |
| Europe/London | 29 | 574,659 |
| Europe/Rome | 14 | 1,467 |

**Using Row-by-Row Time Zone Data to Adjust Time Stamps**

Now that I have time zone information attached to each row of health\_df,  
I can translate the UTC time into the local time as it appeared on my watch.  
In practice I find it quite hard to wrap my head around exactly  
what is happening with all these time manipulations. Sometimes it feels like  
a science fiction time travel story[2](https://johngoldin.com/post/2020-02-15-apple-health-export1/#fn2).

Remember that for the datetime class  
in R, time zone is an attribute that applies to an entire vector.  
That’s a limitation we need to work around. First we will convert  
the character version of the time stamp (with universal time offset)  
to a datetime class value with UTC as the time zone.

Next I create  
a function exported\_time\_to\_local which will convert the time as  
it appears in the Apple Health Export to the time as it appeared at the  
time the data was originally added to the health data.  
As a side effect the lubridate conversion  
functions will also adjust for daylight savings changes.  
This function  
will apply to data for a single time zone only so that it can be  
vectorized. This is important because it will be applied to millions of rows.

utc\_dt\_to\_local <- function(dt, time\_zone) {

# Adjust a vector of datetime from time zone where data was exported

# to a particular time\_zone that that applies to the whole vector.

tz(dt) <- "UTC"

local <- with\_tz(dt, time\_zone) # now adjust utc to the time zone I want

tz(local) <- "UTC"

# I mark the vector as UTC because I will be row\_bind-ing vectors

# together and all need to have the same time zone attribute.

# Although the vector is marked as UTC,

# in the end I will treat the hour as being whatever the local

# time was that I experienced then.

return(local)

}

Next I will apply the utc\_dt\_to\_local function to the character time stamps  
in the Apple Health Export. The function needs to be applied to a vector with  
the same time zone for all elements in the vector. By doing group\_by(start\_time\_zone)  
before I use the function inside mutate, the function will be applied with a  
different time zone for each group.  
That way the function is vectorized for each group and is reasonably fast.  
I did not group the time zones separately for the start date and the end date. Usually  
they would be in the same tine zone, and even if they are not I want to handle them as if they were.  
All the time stamp vectors end up having the time zone attribute of “UTC”.  
Remember that time zone is a single attribute that has to apply to the whole vector.  
In this case it is labelled UTC, but each time stamp describes the local time  
when and where the observation was recorded. So 16:42 means 4:42 PM in the time zone  
(and daylight savings status) at the place and time of the measurement that goes  
with that time stamp.

health\_df <- health\_df %>%

group\_by(timezone) %>%

# assume end\_date is in the same time zone as start\_date

mutate(local\_start = utc\_dt\_to\_local(utc\_start, first(timezone)),

local\_end = utc\_dt\_to\_local(utc\_end, first(timezone))) %>%

# mutate(end\_time\_zone = get\_my\_time\_zone(endDate)) %>%

# group\_by(end\_time\_zone) %>%

# mutate(end\_date = exported\_time\_to\_local(endDate, first(end\_time\_zone))) %>%

ungroup() %>%

mutate(date = as\_date(utc\_start),

start\_time = as.integer(difftime(local\_start, floor\_date(local\_start, "day"), unit = "secs")) %>% hms::hms()) %>%

arrange(type, utc\_start) %>%

ungroup()

# Here I'll adjust time for workout\_df as well

workout\_df <- workout\_df %>%

mutate(utc\_start = as\_datetime(startDate),

utc\_end = as\_datetime(endDate)) %>%

filter(![is.na](http://is.na)(utc\_start)) %>%

interval\_left\_join(arrivals %>%

select(utc\_arrive, utc\_until, timezone = local\_timezone),

by = c("utc\_start" = "utc\_arrive", "utc\_start" = "utc\_until"))

workout\_df <- workout\_df %>%

group\_by(timezone) %>%

mutate(local\_start = utc\_dt\_to\_local(utc\_start, first(timezone)),

local\_end = utc\_dt\_to\_local(utc\_end, first(timezone))) %>%

arrange(utc\_start) %>%

ungroup()

# I'm going to focus on health\_df and workout\_df, but I could adjust times in the other df's as well

At this point we have two sets of date and time stamps for health\_df and workout\_df.  
The prefix “utc” is for universal time and  
should be used for sorting by time. The prefix “local” has local time as was shown on the watch when the  
measurement was recorded. Local time and date is what we need if we want to examine time during the day.  
For health\_df there is also a column called start\_time which is just the time of day, without date, expressed as hh:mm:ss.

The two distributions (UTC and local time) mostly overlap,  
but there are some systematic differences.  
I live in the Eastern Time Zone which is five hours later than  
UTC during standard  
time and four hours later during daylight savings.  
As a compromise I  
subtracted 4.5 hours from the UTC time of day to make the distributions fit on the same scale.  
The red scale shows the distribution of observations in  
the export data in terms of UTC time of day and the blue scale shows local time. The blue scale corresponds more  
to my subjective experience of time of day. You can see that the red distribution is “fatter” during the period from  
4AM to 8AM than is the blue scale. That’s primarily because of walking trips in England that are not in  
the Eastern Time. Observations that appear to be at 4AM in terms of UTC – 4.5 are probably happening in  
British Summer Time (because I’m on a hike in England during the summer) which is actually UTC + 1. In terms  
of my experience of local time, they are happening when my watch says 9AM. (The dataset also includes  
a 10 day walking holiday in Greece which is another two hours farther east than England.)

If one looks at sleep time between 00:00 and 06:00, the Local Time distribution (blue) shows a  
consistent pattern of fewer observations because of less activity. A larger proportion of my  
activity outside of the Eastern Time Zone was in Europe rather than in Pacific Time and that’s why  
the UTC distribution seems shifted to the left relative to Local Time. Also, Local Time is responding to  
daylight savings changes and UTC is not. I think that explains why the peaks before and after  
lunch time are higher in the Local Time distribution than in UTC. I’m actually fairly rigid about when  
I take time for lunch (and presumably reduce my physical activity) and taking daylight savings  
into account makes that more clear.

In summary, if I did not correct for time zone and daylight savings the general bi-modal pattern would  
still be apparent. But the translation into local time makes the time of day pattern more clear  
and more accurate, especially if one focuses on hours when I would expect to be asleep.

Figure 1: Density Plot by Time of Observation

**Apple, If You’re Listening…**

There may be a relatively simple item of data that Apple could add to the health data that would  
make it easier (and more accurate) to execute the adjustments for time zone described in this post.  
Whenever time zone on the watch or phone is changed, surely there is a log of that event. If  
those change events were added to the health dataset as a separate item, then one could construct  
an effective way to adjust all the time stamps in the datasets, similar  
to the way I have used the flight arrival times. I don’t know whether there are  
logs to accomplish this retrospectively, but even if it were added going forward  
that would be a big help. It would add very little data to the database.  
Travel and daylight savings are the  
only events that cause me to change the time on my phone or watch.  
Perhaps there are some unusual situations where someone lives close to the border of  
a time zone where they frequently change among cell phone towers in different  
time zones. Perhaps that would add a lot more time change data, but such a person  
really needs a way to adjust for time zone even more than most people. The fancy solution from Apple  
would be if they added time change events to the data and then used that data to adjust the UTC offset  
at the time they produced the Apple Health Export. That would be a great help and make the data  
less confusing. In that case the time zone offsets that already appear in the  
exported data would actually mean something useful and would allow easy translation between  
UTC and local time.

**Conclusion**

This ends Part I. I decided I need to do a separate Part I because this is mostly about  
the machinations to adjust time stamps to retrieve local time. But for most people that’s  
an issue that can be ignored, meaning the bulk of Part I is not relevant for most people.  
In Part II I will examine some of the data elements in more detail and give  
some examples using the data.