NASA generates and provides heaps of data to the scientific community. Not all  
of it is looking out at the stars. Some of it is looking back at us here on  
Earth. NASA’s Earth science program observes, understands and models the  
Earth system. We can use these data to discover how our Earth is changing,  
to better predict change, and to understand the consequences for life on Earth.

The Earth science program includes the  
[Prediction of Worldwide Energy Resource (POWER)](https://power.larc.nasa.gov/)  
project, which was initiated to improve upon the current renewable energy data  
set and to create new data sets from new satellite systems. The POWER project  
targets three user communities: 1) Renewable Energy (SSE), 2) Sustainable  
Buildings (SB) and 3) Agroclimatology (AG) and covers 140+ different  
parameters.

**How Did This Package Happen?**

I first became acquainted with the POWER data when I joined the GIS lab  
at the [International Rice Research Institute  
(IRRI)](https://irri.org/mapping) in 2011. We commonly used the  
agroclimatology data from POWER to map and model rice projects. In most  
of the work I did, I used it with the EPIRICE model, but it was also used  
for other crop modelling. Since then I have used the POWER data in projects with  
EPIRICE myself and have worked with other researchers who use it for crop  
simulation modelling exercises. The data were a great resource for  
agricultural modelling, even if it was a bit coarse at 1˚ x 1˚ (covering roughly  
12225 km2 or 4721 mi2 per grid cell at the equator, more  
as you approach the poles), the full data set had global coverage, offering data  
where we often needed it in areas that lacked good weather station coverage.

Because I had used the data and I knew plenty of others used the data; in 2017 I  
started writing nasapower to interface with the POWER website and run  
queries to get the data from the server but only for agricultural weather data  
(AG community), as this was my main (really only) interest. This created a  
simplified procedure for downloading the data in place of using the point and  
click interface of the website for repeated queries.

**What Happened Next**

However, in late 2017 very shortly after the package was reviewed, the  
POWER team announced that the POWER data would be getting a makeover.  
The data would be served at 0.5˚ x 0.5˚ and use a new  
[API](https://power.larc.nasa.gov/docs/v1/) to access them. This  
complicated things. I had never worked with an official API before. I  
had only worked with FTP servers, which are at best a very rudimentary  
API, but easy to work with in R. So I went back to the reviewers and  
editor and suggested that I needed to rewrite the whole package due to  
the major changes that were coming. The editor, Scott Chamberlain,  
readily agreed and over the next several months I learned how to write  
an R package that is an API client, a proper API client.

Thankfully rOpenSci has a great package, crul, that makes this much easier.  
So I  
got down to business writing the new version of nasapower using crul.

I quickly found out that the easy part was using crul; the hard part  
was validating user inputs. I decided early in the process of writing  
the package to validate the users’ input on the client side before even  
querying the server to make sure that only well-formed requests were  
sent. That way I could provide the user with feedback on what may have  
been entered incorrectly to make it easier for them to correct rather  
than relying on the server’s response.

There are over 140 different parameters for three “communities”  
(AG, SSE and SB), that the POWER data provides. One of the first issues I  
encountered was how to validate the users’ requests against what was available.

Power JSON File

**POWER JSON file**

Using purrr::map2 and jsonlite::fromJSON() read the JSON file into R creating a single, nested list and reorder it alphabetically by parameter name.

library(purrr)

library(jsonlite)

##

## Attaching package: 'jsonlite'

## The following object is masked from 'package:purrr':

##

## flatten

temporal\_api <- c("HOURLY", "DAILY", "MONTHLY", "CLIMATOLOGY")

community <- c("AG", "RE", "SB")

# create all combinations

vals <- expand.grid(temporal\_api, community, stringsAsFactors = FALSE)

# create equal length vectors for purrr::map2

temporal\_api <- vals[, 1]

community <- vals[, 2]

base\_url <- "https://power.larc.nasa.gov/api/system/manager/parameters?"

power\_pars <- map2(.x = temporal\_api,

.y = community,

.f = ~ fromJSON(paste0(

base\_url,

"temporal=",

.x,

"&community=",

.y

)))

names(power\_pars) <- paste(temporal\_api, community, sep = "\_")

# create a list of vectors for each temporal API/par combination for easier

# checking and validation

parameters <- vector(mode = "list", length = length(power\_pars))

names(parameters) <- names(power\_pars)

for (i in names(power\_pars)) {

parameters[[i]] <- names(power\_pars[[i]])

}

parameters <- map(.x = parameters, .f = sort)

Formatting dates and checking those given all the different ways we enter them  
proved to be another challenge entirely taking nearly 100 lines of code.

Along with learning how to write an API client package, one of the  
methods I used in this package that I had not made full use of before  
was lists. Internally in that 100+ lines of code there are several  
inter-related checks and values that are provided for the query. By  
using lists I was able to return more than one value from a function  
after it was checked and validated and then provide that to the function  
I created to generate the request that crul sends, *e.g.*, the  
longitude, latitude and whether it is a single point or  
region. Here’s the code below

#' Adds %notin% function

#'

#' Negates `%in%`` for easier matching.

#'

#' @param x A character string to match.

#' @param table A table containing values to match `x` against.

#'

#' @return A function to use for checking if something is not in a table

#'

#' @noRd

`%notin%` <- function(x, table) {

# Same as !(x %in% table)

match(x, table, nomatch = 0L) == 0L

}

#' Check dates for validity when querying API

#'

#' Validates user entered date values against `lonlat` and `temporal\_average`

#' values

#'

#' @param dates User entered `dates` value.

#' @param lonlat User entered `lonlat` value.

#' @param temporal\_average User entered `temporal\_average` value.

#'

#' @return Validated dates in a list for use in .power\_query()

#'

#' @noRd

.check\_dates <- function(dates, lonlat, temporal\_average) {

if (is.null(dates) & temporal\_average != "CLIMATOLOGY") {

stop(

call. = FALSE,

"\nYou have not entered dates for the query.\n"

)

}

if (temporal\_average == "INTERANNUAL") {

if (length(unique(dates)) < 2) {

stop(

call. = FALSE,

"\nFor `temporal\_average = INTERANNUAL`, at least two (2) years ",

"are required to be provided.\n"

)

}

if (any(nchar(dates) > 4)) {

dates <- unique(substr(dates, 1, 4))

}

if (dates[[2]] < dates[[1]]) {

message(

"\nYour start and end dates were reversed. ",

"They have been reordered.\n"

)

dates <- c(dates[2], dates[1])

}

return(dates)

}

if (temporal\_average == "DAILY") {

if (is.numeric(lonlat)) {

if (length(dates) == 1) {

dates <- c(dates, dates)

}

if (length(dates) > 2) {

stop(

call. = FALSE,

"\nYou have supplied more than two dates for start and end.\n"

)

}

# put dates in list to use lapply

dates <- as.list(dates)

# check dates as entered by user

date\_format <- function(x) {

tryCatch(

# try to parse the date format using lubridate

x <- lubridate::parse\_date\_time(

x,

c(

"Ymd",

"dmY",

"mdY",

"BdY",

"Bdy",

"bdY",

"bdy"

)

),

warning = function(c) {

stop(

call. = FALSE,

"\n",

x,

" is not a valid entry for date. Enter as YYYY-MM-DD.\n"

)

}

)

as.Date(x)

}

# apply function to reformat/check dates

dates <- lapply(X = dates, FUN = date\_format)

# if the stdate is > endate, flip order

if (dates[[2]] < dates[[1]]) {

message(

"\nYour start and end dates were reversed. ",

"They have been reordered.\n"

)

dates <- c(dates[2], dates[1])

}

# check date to be sure it's not before POWER data start

if (dates[[1]] < "1981-01-01") {

stop(

call. = FALSE,

"\n1981-01-01 is the earliest available data from POWER.\n"

)

}

# check end date to be sure it's not \_after\_

if (dates[[2]] > Sys.Date()) {

stop(

call. = FALSE,

"\nThe data cannot possibly extend beyond this moment.\n"

)

}

dates <- lapply(dates, as.character)

dates <- gsub("-", "", dates, ignore.case = TRUE)

}

}

}

#' Check community for validity when querying API

#'

#' Validates user entered `community` values against `pars` for availability.

#'

#' @param community User entered `community` value.

#' @param pars User entered `pars` value.

#'

#' @return Validated community and pars for use in .power\_query()

#'

#' @noRd

.check\_community <-

function(community, pars) {

if (community %notin% c("AG", "SB", "SSE")) {

stop(

call. = FALSE,

"\nYou have provided an invalid `community` value.\n"

)

}

# check to make sure community is appropriate for given pars

for (i in pars) {

if (community %notin% parameters[[i]]$community) {

stop(

call. = FALSE,

"\nYou have entered an invalid value for `community` for ",

"the supplied `pars`. One or more `pars` are not, available for ",

"`", community, "`, please check.\n"

)

}

}

}

#' Check pars for validity when querying API

#'

#' Validates user entered date values against `lonlat` and `temporal\_average`

#' values

#'

#' @param dates User entered `dates` value.

#' @param lonlat User entered `lonlat` value.

#' @param temporal\_average User entered `temporal\_average` value.

#'

#' @return Validated pars for use in .power\_query()

#'

#' @noRd

.check\_pars <-

function(pars, temporal\_average, lonlat) {

# check pars to make sure that they are valid

if (any(pars %notin% names(parameters))) {

stop(

call. = FALSE,

paste0(

"\n", pars[which(pars %notin% names(parameters))],

" is/are not valid in 'pars'.\n"

)

)

}

# check to make sure temporal\_average is appropriate for given pars

for (i in pars) {

if (temporal\_average %notin% parameters[[i]]$include) {

stop(

call. = FALSE,

"\nYou have entered an invalid value for `temporal\_average` for ",

"the supplied `pars`. One or more `pars` are not, available for ",

"`", temporal\_average, "`, please check.\n"

)

}

}

# make sure that there are no duplicates in the query

pars <- unique(pars)

# check pars to make sure < allowed

if (length(pars) > 3 & temporal\_average == "CLIMATOLOGY") {

stop(

call. = FALSE,

"\nYou can only specify three (3) parameters for download when ",

"querying CLIMATOLOGY.\n"

)

}

if (length(pars) > 20 & temporal\_average != "CLIMATOLOGY") {

stop(

call. = FALSE,

"\nYou can only specify 20 parameters for download at a time.\n"

)

}

# calculate how many lines to skip in the header to read the CSV from server

if (temporal\_average == "CLIMATOLOGY") {

skip\_lines <- length(pars) + 7

} else {

skip\_lines <- length(pars) + 9

}

# all good? great. now we format it for the API

pars <- paste0(pars, collapse = ",")

pars <- list(pars, temporal\_average, skip\_lines)

names(pars) <- c("pars", "temporal\_average", "skip\_lines")

return(pars)

}

#' Check lonlat for validity when querying API

#'

#' Validates user entered `lonlat` values and checks against `pars` values.

#'

#' @param lonlat User entered `lonlat` value.

#' @param pars User entered `pars` value.

#'

#' @return A list called lonlat\_identifier for use in .power\_query()

#'

#' @noRd

.check\_lonlat <-

function(lonlat, pars, temporal\_average) {

bbox <- NULL

if (is.numeric(lonlat) & length(lonlat) == 2) {

if (lonlat[1] < -180 | lonlat[1] > 180) {

stop(

call. = FALSE,

"\nPlease check your longitude, `",

paste0(lonlat[1]),

"`, to be sure it is valid.\n"

)

}

if (lonlat[2] < -90 | lonlat[2] > 90) {

stop(

call. = FALSE,

"\nPlease check your latitude, `",

paste0(lonlat[2]),

"`, value to be sure it is valid.\n"

)

}

identifier <- "SinglePoint"

lon <- lonlat[1]

lat <- lonlat[2]

} else if (length(lonlat) == 4 & is.numeric(lonlat)) {

if ((lonlat[[3]] - lonlat[[1]]) \* (lonlat[[4]] - lonlat[[2]]) \* 4 > 100) {

stop(

call. = FALSE,

"\nPlease provide correct bounding box values. The bounding box\n",

"can only enclose a max of 10 x 10 region of 0.5 degree values\n",

"or a 5 x 5 region of 1 degree values, (i.e. 100 points total).\n"

)

} else if (any(lonlat[1] < -180 |

lonlat[3] < -180 |

lonlat[1] > 180 |

lonlat[3] > 180)) {

stop(

call. = FALSE,

"\nPlease check your longitude, `",

lonlat[1],

"`, `",

lonlat[3],

"`, values to be sure they are valid.\n"

)

} else if (any(lonlat[2] < -90 |

lonlat[4] < -90 |

lonlat[2] > 90 |

lonlat[4] > 90)) {

stop(

call. = FALSE,

"\nPlease check your latitude, `",

lonlat[2],

"`, `",

lonlat[4],

"`, values to be sure they are valid.\n"

)

} else if (lonlat[2] > lonlat[4]) {

stop(

call. = FALSE,

"\nThe first `lat` value must be the minimum value.\n"

)

} else if (lonlat[1] > lonlat[3]) {

stop(

call. = FALSE,

"\nThe first `lon` value must be the minimum value.\n"

)

}

identifier <- "Regional"

bbox <- paste(lonlat[2],

lonlat[1],

lonlat[4],

lonlat[3],

sep = ","

)

} else if (temporal\_average == "CLIMATOLOGY") {

identifier <- "Global"

} else {

stop(

call. = FALSE,

"\nYou have entered an invalid request for `lonlat`.\n"

)

}

if (!is.null(bbox)) {

lonlat\_identifier <- list(bbox, identifier)

names(lonlat\_identifier) <- c("bbox", "identifier")

} else if (identifier == "Global") {

lonlat\_identifier <- list("Global")

names(lonlat\_identifier) <- "identifier"

} else {

lonlat\_identifier <- list(lon, lat, identifier)

names(lonlat\_identifier) <- c("lon", "lat", "identifier")

}

return(lonlat\_identifier)

}

#' Query the POWER API

#'

#' Constructs and sends a query to the POWER API using validated values from

#' previous functions in this file.

#'

#' @param community A validated value for community from `check\_community()`.

#' @param lonlat\_identifier A list of values, a result of `check\_lonlat()`.

#' @param pars A validated value from `check\_pars()`.

#' @param dates A list of valuse, a result of `check\_dates()`.

#' @param outputList A value of either "CSV" or "ICASA" that tells the API the

#' desired format in which to return the data.

#'

#' @return A tidy data.frame() of requested 'POWER' data

#'

#' @noRd

.power\_query <- function(community,

lonlat\_identifier,

pars,

dates,

outputList) {

power\_url <- # nocov start

"https://power.larc.nasa.gov/cgi-bin/v1/DataAccess.py?"

client <- crul::HttpClient$new(url = power\_url)

user\_agent <- "nasapower"

# check status

status <- client$get()

status$raise\_for\_status() # nocov end

if (lonlat\_identifier$identifier == "SinglePoint") {

query\_list <- list(

request = "execute",

identifier = lonlat\_identifier$identifier,

parameters = I(pars$pars),

startDate = dates[[1]],

endDate = dates[[2]],

userCommunity = community,

tempAverage = pars$temporal\_average,

outputList = outputList,

lon = lonlat\_identifier$lon,

lat = lonlat\_identifier$lat,

user = user\_agent

)

}

if (lonlat\_identifier$identifier == "Regional") {

query\_list <- list(

request = "execute",

identifier = lonlat\_identifier$identifier,

parameters = I(pars$pars),

startDate = dates[[1]],

endDate = dates[[2]],

userCommunity = community,

tempAverage = pars$temporal\_average,

bbox = I(lonlat\_identifier$bbox),

outputList = outputList,

user = user\_agent

)

}

if (lonlat\_identifier$identifier == "Global") {

query\_list <- list(

request = "execute",

identifier = lonlat\_identifier$identifier,

parameters = I(pars$pars),

userCommunity = community,

tempAverage = pars$temporal\_average,

outputList = "CSV",

user = user\_agent

)

}

# send the query

tryCatch({

response <- client$get(query = query\_list, retry = 6)

txt <- jsonlite::fromJSON(response$parse("UTF-8"))

raw\_power\_data <- file.path(tempdir(), "power\_data\_file")

}, # nocov start

error = function(e) {

e$message <-

paste("\nSomething went wrong with the query. Please try again.\n")

# Otherwise refers to open.connection

e$call <- NULL

stop(e)

}

) # nocov end

if ("messages" %in% names(txt) & "outputs" %notin% names(txt)) {

stop(

call. = FALSE,

unlist(txt$messages)

)

}

if ("csv" %in% names(txt$output)) {

if (outputList == "CSV") {

curl::curl\_download(txt$output$csv,

destfile = raw\_power\_data,

mode = "wb",

quiet = TRUE

)

meta <- readLines(

raw\_power\_data,

pars$skip\_lines

)

meta <- meta[-c(1, pars$skip\_lines)] # remove "HEADER ..." lines

meta <- gsub(

pattern = "-99",

replacement = "NA",

x = meta

)

power\_data <- readr::read\_csv(raw\_power\_data,

col\_types = readr::cols(),

na = "-99",

skip = pars$skip\_lines

)

# put lon before lat (x, y format)

power\_data <- power\_data[, c(2, 1, 3:ncol(power\_data))]

# if the temporal average is anything but climatology, add date fields

if (pars$temporal\_average == "DAILY") {

power\_data <- .format\_dates(power\_data)

}

attr(power\_data, "class") <- c(

"POWER.Info",

"tbl\_df",

"tbl",

"data.frame"

)

# add attributes for printing df

attr(power\_data, "POWER.Info") <- meta[1]

attr(power\_data, "POWER.Dates") <- meta[2]

attr(power\_data, "POWER.Location") <- meta[3]

attr(power\_data, "POWER.Elevation") <- meta[4]

attr(power\_data, "POWER.Climate\_zone") <- meta[5]

attr(power\_data, "POWER.Missing\_value") <- meta[6]

attr(power\_data, "POWER.Parameters") <- paste(meta[8:length(meta)],

collapse = ";\n "

)

return(power\_data)

}

} else if ("icasa" %in% names(txt$output)) {

curl::curl\_download(txt$output$icasa,

destfile = raw\_power\_data,

mode = "wb",

quiet = TRUE

)

power\_data <- readLines(raw\_power\_data)

return(power\_data)

} else {

stop(

call. = FALSE,

"Your requested data was not returned. Check",

"<https://power.larc.nasa.gov/> and/or try again later."

)

}

}

#' Prints Power.Info object.

#'

#' @param x POWER.Info object

#' @param ... ignored

#' @export

print.POWER.Info <- function(x, ...) {

if (!is.null(attr(x, "POWER.Info"))) {

cat(

attr(x, "POWER.Info"), "\n",

attr(x, "POWER.Dates"), "\n",

attr(x, "POWER.Location"), "\n",

attr(x, "POWER.Elevation"), "\n",

attr(x, "POWER.Climate\_zone"), "\n",

attr(x, "POWER.Missing\_value"), "\n",

"\n",

"Parameters: \n",

attr(x, "POWER.Parameters"), "\n",

"\n"

)

format(x)

}

NextMethod(x)

invisible(x)

}

#' Format date columns in POWER data frame

#'

#' Formats columns as integers for DOY and adds columns for year, month and day.

#'

#' @param NASA A tidy data.frame resulting from .power\_query().

#'

#' @return A tidy data frame of power data with additional date information

#' columns.

#'

#' @noRd

#'

.format\_dates <- function(NASA) {

# convert DOY to integer

NASA$DOY <- as.integer(NASA$DOY)

# Calculate the full date from YEAR and DOY

NASA <- tibble::add\_column(NASA,

YYYYMMDD = as.Date(NASA$DOY - 1,

origin = as.Date(paste(

NASA$YEAR, "-01-01",

sep = ""

))

),

.after = "DOY"

)

# Extract month as integer

NASA <- tibble::add\_column(NASA,

MM = as.integer(substr(NASA$YYYYMMDD, 6, 7)),

.after = "YEAR"

)

# Extract day as integer

NASA <- tibble::add\_column(NASA,

DD = as.integer(substr(NASA$YYYYMMDD, 9, 10)),

.after = "MM"

)

}  
By doing this I was able to simplify the parameters that the user had to  
enter when constructing a query in their R session because the API  
dictates by default many values that cannot co-occur.

**Are We Doing it Right?**

There were a few hang-ups along the way. At one point the POWER team  
made a change to the API response while the package was in review.  
Initially it provided an option for just a CSV file that was properly  
rectangular. Suddenly it changed overnight with no warning to offering a  
CSV file with information in a “header” as they called it. To me a  
header in a CSV file is just column names. This was more. This was  
metadata about the data itself. The quickest way to deal with this was  
to simply read only the CSV file portion of the returned file. However,  
I thought it might be useful to users to include the metadata that POWER  
was now providing in this format as well. So I learned something else  
new, how to modify the S3 print() method function to include the  
metadata in the console along with the weather or climate data.

Later, in early 2019 I ran into an issue with the method I’d used to  
validate user inputs for community when building the query that was  
sent. Looking  
into the JSON file I used as suggested by Ben Raymond, I found the issue. The  
POWER team did not list the AG community as having this data available. I  
contacted them thinking is was a mistake in the file since the data are in  
indeed available. After some back and forth with the NASA-POWER team I never  
really was clear on how I should handle this from their perspective aside from  
their suggestion of just looking at their webpage to see what was available.  
Additionally, when I checked, it appeared that the data were available for all  
three communities. So I ended up dropping the check for community from the  
user input validations because the community specified only affects the units  
that the data are reported in. I still check user inputs for the correct parameter specification and  
temporal matches, but assume all communities offer the same data. It’s not  
optimal but until the POWER team can deliver a more robust way for us to check  
against their records it will have to do.

**Finally, It (Mostly?) Works!**

The package is relatively simple as far as functionality. It only  
fetches data and reformats it for use in R or crop modelling software.

**The Three Functions**

The first function, get\_power(), is the main function that is used and  
offers the greatest flexibility. Say you want to know the maximum and  
minimum temperatures for Death Valley in California, USA every day for  
the last 35 years, you can easily do this with nasapower using  
coordinates for the valley floor.

If you are not sure what the parameters are that you need to get the  
data you can always have a look at the list of available parameters  
using ?parameters. That shows that T2M\_MIN and T2M\_MAX  
are the minimum and maximum temperatures at 2 meters. We will use those  
to construct the query.

library(nasapower)

dv <- get\_power(community = "AG",

lonlat = c(-117.2180, 36.7266),

dates = c("1983-01-01", "2018-12-31"),

temporal\_average = "DAILY",

pars = c("T2M\_MIN", "T2M\_MAX"))

dv

## NASA/POWER SRB/FLASHFlux/MERRA2/GEOS 5.12.4 (FP-IT) 0.5 x 0.5 Degree Daily Averaged Data

## Dates (month/day/year): 01/01/1983 through 12/31/2018

## Location: Latitude 36.7266 Longitude -117.218

## Elevation from MERRA-2: Average for 1/2x1/2 degree lat/lon region = 1192.53 meters Site = na

## Climate zone: na (reference Briggs et al: http://www.energycodes.gov)

## Value for missing model data cannot be computed or out of model availability range: -99

##

## Parameters:

## T2M\_MIN MERRA2 1/2x1/2 Minimum Temperature at 2 Meters (C) ;

## T2M\_MAX MERRA2 1/2x1/2 Maximum Temperature at 2 Meters (C)

##

## # A tibble: 13,149 x 9

## LON LAT YEAR MM DD DOY YYYYMMDD T2M\_MIN T2M\_MAX

##

## 1 -117. 36.7 1983 1 1 1 1983-01-01 -4.73 7.24

## 2 -117. 36.7 1983 1 2 2 1983-01-02 -3.33 7.72

## 3 -117. 36.7 1983 1 3 3 1983-01-03 0.05 11.8

## 4 -117. 36.7 1983 1 4 4 1983-01-04 1.27 14.9

## 5 -117. 36.7 1983 1 5 5 1983-01-05 2.55 15.9

## 6 -117. 36.7 1983 1 6 6 1983-01-06 2.63 17.1

## 7 -117. 36.7 1983 1 7 7 1983-01-07 3.2 17.2

## 8 -117. 36.7 1983 1 8 8 1983-01-08 1.96 18.3

## 9 -117. 36.7 1983 1 9 9 1983-01-09 0.7 14.0

## 10 -117. 36.7 1983 1 10 10 1983-01-10 1.3 17.6

## # … with 13,139 more rows

In the metadata header you can see information about where the data  
comes from and what dates have been queried and returned as well as the  
elevation data.

The tibble contains a few columns not in the original data, but that  
can make it easier to work within R. The original data only include YEAR  
and DOY. Looking at the data returned, there are:

* LON = the queried longitude as a double;
* LAT = the queried latitude as a double;
* YEAR = the queried year as a double;
* MM = the queried month as a double,
* DD = the queried day as a double,
* DOY = the day of year or Julian date as an integer,
* YYYYMMDD = the full date as a date object and the requested parameters,
* T2M\_MIN = the minimum temperature at 2 meters above the Earth’s surface as a double, and
* T2M\_MAX = the maxiumum temperature at 2 meters above the Earth’s surface as a double.

**Visualising the Data**

To visualise these data I will use ggplot2, but first I need to gather the data  
into long format using tidyr’s gather().

library(tidyr)

library(ggplot2)

library(hrbrthemes)

dv\_long <-

tidyr::gather(dv, key = "T2M",

value = "Degrees",

c("T2M\_MIN", "T2M\_MAX"))

ggplot(dv\_long, aes(x = YYYYMMDD, y = Degrees,

colour = Degrees)) +

geom\_line() +

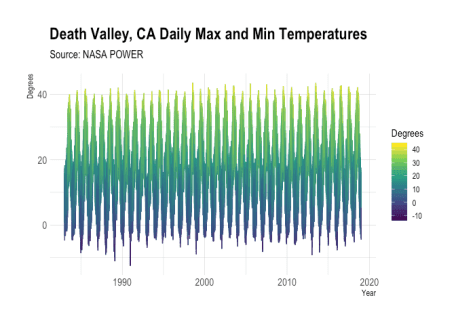
xlab("Year") +

ggtitle(label = "Death Valley, CA Daily Max and Min Temperatures",

sub = "Source: NASA POWER") +

scale\_colour\_viridis\_c() +

theme\_ipsum()

Figure 1: Daily temperature extremes at 2 meters above the Earth’s surface for the grid cell covering Death Valley, California, USA for the time-period from 1983 to 2018.

That is quite a swing in air temperatures from well over 40˚ C to well  
below 0˚ C throughout the year. I was going to put together a comparison  
with station data using GSODR but  
instead found a good reason why you might want to use nasapower to get  
POWER data. When I checked for stations nearby this specified point,  
there were two in the GSOD database, 746190-99999 and 999999-53139.  
However, neither one of them offers weather data for this time period.  
The station, 746190-99999, only offers data from 1982 to 1992 and  
999999-53139 only provides data from 2004 to 2019. This nicely  
illustrates one of the advantages of the POWER data, it is available for  
any point on the globe. One of the weaknesses you might have noticed if  
you looked at the elevation data in the metadata header is that it shows  
the elevation of Death Valley is over 1000 m above sea level when we  
know that it is 86 m below sea level. This is due to the fact that the  
data are the average of a 0.5˚ by 0.5˚ area or roughly  
3000 km2 as opposed to a single point that a station can  
provide.

**Crop Modelling From Space**

This can be advantageous as well, however. Some agricultural scientists work  
with models that predict crop yields in response to changes to the crop,  
weather, farmer inputs or even climate change. Crop yield modelling often  
uses daily weather data covering large relatively continuous areas that  
are less affected by things like elevation, so for these purposes, the  
POWER data can be very useful. Another reason the POWER data are useful  
could be that the same data set is available for many areas, making  
model output comparisons easier.

If you do any crop modelling work you are likely familiar with the  
Decision Support System for Agrotechnology Transfer  
[DSSAT](https://dssat.net/) platform. The new POWER API  
provides properly formatted [ICASA](https://dssat.net/data/standards_v2) files,  
which are the format that DSSAT uses. Naturally I took advantage of this and  
added a function, create\_icasa(), to download and save ICASA files for use in  
crop simulations.

But, being in Toowoomba, Queensland, I had to acknowledge another crop  
simulation model, the Agricultural Production Systems sIMulator  
[APSIM](https://www.apsim.info/). APSIM was developed here and has similar  
functionality to DSSAT. However, the POWER API did not offer properly formatted  
APSIM .met files. So, wrote a function, create\_met(), that takes advantage of  
the POWER data API and the R APSIM package to generate the proper weather  
.met files since many APSIM users, use R in their modelling pipeline, *e.g.*,  
APSIMBatch and apsimr.

Both of these functions simply download data and write the values to  
disk in a specialised file format that these crop modelling platforms  
use, therefore I have declined to illustrate their usage in this blog  
post.

**There Is More Than Just Daily Data for Single Cells**

**Retrieving Climatology**

Daily weather data are not the only data offered by this API. Two other options  
exist for the temporal\_average parameter, INTERANNUAL and CLIMATOLOGY.  
INTERANNUAL data provide monthly averages for the same 0.5˚ x 0.5˚ grid as the  
daily data for the time-period the user specifies, while CLIMATOLOGY provides  
0.5˚ x 0.5˚ gridded data of a thirty year time period from January 1984 to  
December 2013.

The CLIMATOLOGY data are the only way to get the entire surface in one query,  
but single cell and regional data are also available for this temporal average.

library(raster)

library(GSODR)

library(nasapower)

## Loading required package: sp

##

## Attaching package: 'raster'

## The following object is masked from 'package:tidyr':

##

## extract

library(viridisLite)

global\_t2m <-

get\_power(

community = "AG",

pars = "T2M",

temporal\_average = "CLIMATOLOGY",

lonlat = "GLOBAL"

)

# view the tibble

global\_t2m

## NASA/POWER SRB/FLASHFlux/MERRA2/GEOS 5.12.4 (FP-IT) 0.5 x 0.5 Degree Climatologies

## 22-year Additional Solar Parameter Monthly & Annual Climatologies (July 1983 - June 2005), 30-year Meteorological and Solar Monthly & Annual Climatologies (January 1984 - December 2013)

## Location: Global

## Value for missing model data cannot be computed or out of model availability range: -99

## Parameter(s):

## T2M MERRA2 1/2x1/2 Temperature at 2 Meters (C)

##

## Parameters:

## NA;

## NA;

## T2M MERRA2 1/2x1/2 Temperature at 2 Meters (C)

##

## # A tibble: 259,200 x 16

## LON LAT PARAMETER JAN FEB MAR APR MAY JUN JUL AUG

##

## 1 -180. -89.8 T2M -29.0 -40.7 -52.9 -57.8 -59.1 -59.6 -61.3 -61.8

## 2 -179. -89.8 T2M -29.0 -40.7 -52.9 -57.8 -59.1 -59.6 -61.3 -61.8

## 3 -179. -89.8 T2M -29.0 -40.7 -52.9 -57.8 -59.1 -59.6 -61.3 -61.8

## 4 -178. -89.8 T2M -29.0 -40.7 -52.9 -57.8 -59.1 -59.6 -61.3 -61.8

## 5 -178. -89.8 T2M -29.0 -40.7 -52.9 -57.8 -59.1 -59.6 -61.3 -61.8

## 6 -177. -89.8 T2M -28.9 -40.7 -52.9 -57.9 -59.1 -59.6 -61.3 -61.8

## 7 -177. -89.8 T2M -28.9 -40.7 -52.9 -57.9 -59.1 -59.6 -61.3 -61.8

## 8 -176. -89.8 T2M -28.9 -40.7 -53.0 -57.9 -59.1 -59.6 -61.3 -61.8

## 9 -176. -89.8 T2M -28.9 -40.7 -53.0 -57.9 -59.1 -59.6 -61.3 -61.8

## 10 -175. -89.8 T2M -28.9 -40.7 -53.0 -57.9 -59.1 -59.6 -61.3 -61.8

## # … with 259,190 more rows, and 5 more variables: SEP , OCT ,

## # NOV , DEC , ANN

# map only annual average temperatures by converting the 15th column to a raster

# object

T2M\_ann <- rasterFromXYZ(

global\_t2m[c(1:2, 16)],

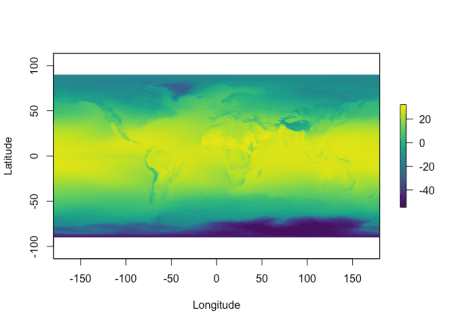
crs = "+proj=eqc +lat\_ts=0 +lat\_0=0 +lon\_0=0 +x\_0=0 +y\_0=0 +ellps=WGS84 +datum=WGS84 +units=m +no\_defs")

# how many unique values

n <- length(unique(global\_t2m$ANN))

# plot the annual average temperature using viridis

plot(T2M\_ann, col = viridis(n = n), xlab = "Longitude", ylab = "Latitude")

Figure 2: Global 30-year annual meteorological (January 1984 – December 2013) average temperature at 2 meters above the Earth’s surface modelled from satellite derived data. You can mostly make out the outlines of the continents and especially the mountain ranges such as the Andes and Rocky Mountains to the left and the Tibetan plateau at about 100˚ longitude (x-axis) and 45˚ latitude (y-axis).

**Retrieving Regional Data**

If your interests cover a large area, it is possible to retrieve an area of  
cells, rather than a single cell in a query. They can not be more than 100  
points in total for an area of 4.5˚ x 4.5˚. Regional coverage is simply  
specified by providing a bounding box as “lower left (lon, lat)” and “upper  
right (lon, lat)” coordinates, *i.e.*, lonlat = c(xmin, ymin, xmax, ymax) in  
that order for a given region, *e.g.*, a bounding box for the south-western  
corner of Australia:

lonlat = c(112.5, -55.5, 115.5, -50.5).

regional\_t2m <-

get\_power(

community = "AG",

pars = "T2M",

temporal\_average = "CLIMATOLOGY",

lonlat = c(112.5, -55.5, 115.5, -50.5)

)

# view the tibble

regional\_t2m

## NASA/POWER SRB/FLASHFlux/MERRA2/ 0.5 x 0.5 Degree Climatologies

## 22-year Additional Solar Parameter Monthly & Annual Climatologies (July 1983 - June 2005), 30-year Meteorological and Solar Monthly & Annual Climatologies (January 1984 - December 2013)

## Location: Regional

## Elevation from MERRA-2: Average for 1/2x1/2 degree lat/lon region = na meters Site = na

## Climate zone: na (reference Briggs et al: http://www.energycodes.gov)

## Value for missing model data cannot be computed or out of model availability range: -99

##

## Parameters:

## T2M MERRA2 1/2x1/2 Temperature at 2 Meters (C)

##

## # A tibble: 77 x 16

## LON LAT PARAMETER JAN FEB MAR APR MAY JUN JUL AUG

##

## 1 113. -55.2 T2M 2.98 3.38 3.13 2.4 1.65 1.01 0.41 -0.05

## 2 113. -55.2 T2M 3.07 3.47 3.22 2.5 1.75 1.11 0.51 0.05

## 3 114. -55.2 T2M 3.14 3.55 3.32 2.6 1.85 1.21 0.61 0.15

## 4 114. -55.2 T2M 3.2 3.62 3.39 2.69 1.94 1.29 0.69 0.22

## 5 115. -55.2 T2M 3.26 3.69 3.46 2.76 2.01 1.37 0.77 0.28

## 6 115. -55.2 T2M 3.3 3.73 3.51 2.82 2.06 1.43 0.83 0.34

## 7 116. -55.2 T2M 3.32 3.77 3.54 2.86 2.1 1.49 0.88 0.39

## 8 113. -54.8 T2M 3.12 3.51 3.3 2.57 1.82 1.17 0.59 0.12

## 9 113. -54.8 T2M 3.2 3.6 3.39 2.68 1.92 1.26 0.69 0.21

## 10 114. -54.8 T2M 3.27 3.7 3.49 2.78 2.02 1.36 0.78 0.3

## # … with 67 more rows, and 5 more variables: SEP , OCT ,

## # NOV , DEC , ANN

# map only annual average temperatures by converting the 15th column to a raster

# object

T2M\_ann\_regional <- rasterFromXYZ(

regional\_t2m[c(1:2, 16)],

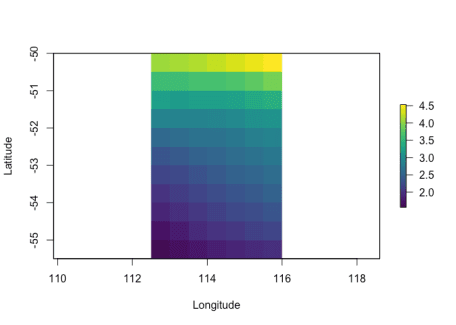
crs = "+proj=eqc +lat\_ts=0 +lat\_0=0 +lon\_0=0 +x\_0=0 +y\_0=0 +ellps=WGS84 +datum=WGS84 +units=m +no\_defs")

# how many unique values

n <- length(unique(regional\_t2m$ANN))

# plot the annual average temperature using viridis

plot(T2M\_ann\_regional, col = viridis(n = n), xlab = "Longitude", ylab = "Latitude")

Figure 3: Regional 30-year annual meteorological (January 1984 – December 2013) average temperature at 2 meters above the Earth’s surface modelled from satellite derived data for the south-western coastal area of Australia, illustrating the maximum allowable cells in a regional query.

**Conclusion**

Even though the package took me well over one year to write and work out  
all of the bugs and has only three functions, I learned quite a bit from  
the experience. The new API really does improve the ability for a  
developer to write a client to interface with the data. Now the  
nasapower package is able to access all of the data available whereas  
the initial version was only able to work with the “AG” community data.

While nasapower does not redistribute the data or provide it in any  
way, if you use the data, we encourage users to follow the requests of  
the POWER Project Team.

When POWER data products are used in a publication, we request the  
following acknowledgement be included: “These data were obtained from  
the NASA Langley Research Center POWER Project funded through the NASA  
Earth Science Directorate Applied Science Program.”

The approach I have used mostly works, but there has been at least one  
example where data are listed as being available in the POWER database  
but querying by the apparently proper pars does not work. Because of  
issues like this, in what appear to be edge cases, I suggest checking  
the web interface if you experience difficulties querying data.  
Hopefully the API itself has settled out a bit and will not have the  
sudden changes that I experienced. The POWER team have been supportive  
of the package, and I have received feedback and interaction from other  
users along the way that helped me by finding and reporting bugs.