

A Practical Guide for Working with Weather Datasets, Topic #2:

# Strategies for Processing Large Weather Datasets

DECEMBER | 2022

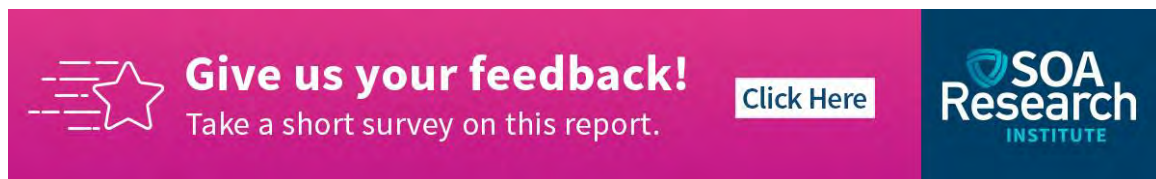






# Strategies for Processing Large Weather Datasets

**AUTHOR** Patrick Wiese, ASA

**SPONSOR** Catastrophe and Climate Strategic  
Research Program Steering Committee

A horizontal banner with a pink-to-blue gradient background. On the left is a white star icon with horizontal lines extending from its left side. To the right of the star is the text "Give us your feedback!" in white, bold font, followed by "Take a short survey on this report." in a smaller white font. Further right is a white rectangular button with the text "Click Here" in black. On the far right of the banner is the SOA Research Institute logo in white on a dark blue background.

 **Give us your feedback!**  
Take a short survey on this report. [Click Here](#) 

#### **Caveat and Disclaimer**

The opinions expressed and conclusions reached by the authors are their own and do not represent any official position or opinion of the Society of Actuaries Research Institute, the Society of Actuaries or its members. The Society of Actuaries Research Institute makes no representation or warranty to the accuracy of the information.

Copyright © 2022 by the Society of Actuaries Research Institute. All rights reserved.

# CONTENTS

- Executive Summary ..... 4**
- 1. How to Process a Large Data File in Small Chunks ..... 5**
- 2. How to Iteratively Process a Comma-Delimited Text File Using VBA..... 5**
- 3. How to Iteratively Process a Fixed-Width Text File Using VBA ..... 8**
- 4. How to Iteratively Process a NetCDF File Using R ..... 9**
- 5. A Speed Test Comparing VBA, R, C++, and Python ..... 13**
- 6. Options for Dealing with Limited Hard Drive Space ..... 15**
- 7. A Preview of the Next Paper in this Series ..... 17**
- Appendix A. Reading Large Text Files: Examples in VBA, R, Python and C++ ..... 17**
- Appendix B. How to Install R, Python, or C++ on Your PC ..... 18**
- Feedback ..... 18**
- About The Society of Actuaries Research Institute ..... 19**

# Strategies for Processing Large Weather Datasets

## Executive Summary

Many weather datasets exceed 100 gigabytes, and some are much larger. While climate scientists have access to servers that can store and process massive weather datasets, actuaries and other risk analysts may lack this infrastructure. To analyze weather data, therefore, a risk analyst may be faced with the prospect of using a standard personal computer (PC). A PC rarely offers more than several hundred gigabytes of available storage space and 16 to 32 gigabytes of RAM (active memory for running applications and programs). Given these constraints, a researcher using a PC will need a clever strategy for working with large weather datasets. This paper describes techniques for analyzing large weather datasets despite the memory and storage limitations imposed by a PC.

To overcome a PC's memory constraints, we present illustrative computer programs that iteratively loop through a large data file, processing the data in small chunks, analogous to eating a large meal in small bites. This technique can be programmed in many different languages, and this paper presents examples written in R, VBA, C++, and Python. Because the illustrative programs are short and simple, no prior programming experience is necessary to grasp the key ideas.

To address the limited storage space available on a PC's hard drive, we present a case study that involves the analysis of 1500 gigabyte weather dataset. Various techniques were used to make the data easier to store, including the following: jettisoning data for geographic regions that were not required for the analysis; restructuring the data to eliminate excessive temporal resolution; and rounding each observation to two decimal places to eliminate excess precision and reduce data storage requirements.

The computer programs referenced in this paper can be downloaded from the same web page on which this paper is available. These programs are as follows: (1) "create\_text\_file.xlsm", a VBA program that generates CSV and fixed-width text files that contain synthetic precipitation data; (2) four programs which read and tabulate the synthetic data: "speed\_text.xlsm" (written in VBA), "speed\_test.r" (written in R), "speed\_test.py" (written in Python), and "speed\_test.cpp" (written in C++); and (3) "read\_netcdf.r" (written in R), which demonstrates how to loop through a netCDF file, reading and processing the data in small "slices", thereby placing little demand on a PC's limited RAM.



**Give us your feedback!**

Take a short survey on this report.

[Click Here](#)

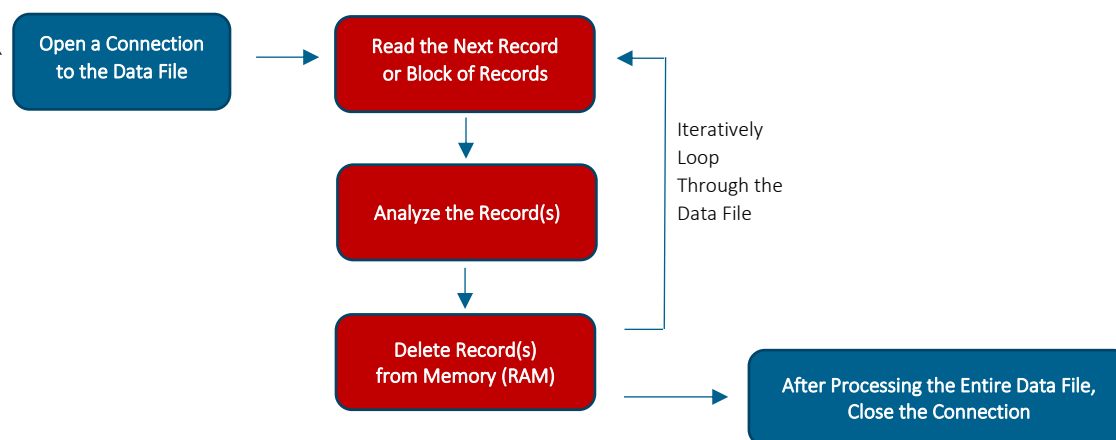
**SOA**  
**Research**  
INSTITUTE

## 1. How to Process a Large Data File in Small Chunks

Due to limited RAM, one cannot load a large weather dataset in its entirety into a PC’s active memory. Instead, the data must be processed iteratively, one record at a time, or one block of records at a time. In a computer program, an iterative process can be performed using a “loop” that repeatedly runs a set of instructions until a stopping condition – such as reaching the end of a data file -- is met.

Exhibit 1

### THE BASIC STRUCTURE OF A COMPUTER PROGRAM FOR LOOPING THROUGH A LARGE DATA FILE



For each iteration of the loop, a record (or block of records) is analyzed, the results are saved, and the record(s) are purged from RAM, thereby freeing up memory to read the next record (or record block).

Weather data is typically stored as comma-delimited text files, fixed-width text files, or netCDF files<sup>1</sup>. Each of these three file types can be read and processed iteratively, one small chunk at a time, but the required computer programming syntax varies from one file type to another. In the pages that follow, we illustrate how to loop through each of these three file types. Because VBA is included with Excel, and because Excel is ubiquitous in the workplace, we use VBA to illustrate how to loop through text files. However, for illustrating how to read NetCDF files, VBA is a poor choice because it lacks a netCDF library. In contrast, R has an excellent netCDF library; therefore, we use R to illustrate how to loop through netCDF files.

Obviously, syntax varies from one computer programming language to another. But there are key features – such as loops for repetitively executing a set of commands – that are common across most languages. Thus, an example presented in one programming language may be helpful even if your preference is to work in a different language.

## 2. How to Iteratively Process a Comma-Delimited Text File Using VBA

In a comma-delimited text file, data elements are separated by commas. This file type is sometimes referred to as “CSV” which stands for “comma separated values”. To illustrate how to read a CSV file using VBA, we will begin by creating synthetic precipitation data and storing it in CSV format. To generate the synthetic dataset, please download “Create\_Text\_File.xlsm” from the SOA’s weather guide home page. Open this Excel file, enable the macros, and go to the tab named “Create a Synthetic Dataset”. In cell B5, enter a name for the data file that will be

<sup>1</sup> NetCDF is a binary file format that is commonly used to store scientific data that has a repetitive structure involving several dimensions such as latitude, longitude, and time.

produced, such as “Synthetic\_Data.csv”. Then press the button labeled “Generate File with Synthetic Precipitation Data”. This will execute a VBA macro that produces the synthetic data. If you wish to examine the macro, right-click on the “Generate File” button, left-click on “Assign Macro”, and left-click on “Edit”.

By default, the macro’s input parameters are set to produce a 90-megabyte CSV data file consisting of one million monthly records, each with 30 daily precipitation observations. For the sake of simplicity, every month is assumed to have 30 days. A few of the synthetic records are presented in Exhibit 2:

#### Exhibit 2

##### SYNTHETIC DAILY PRECIPITATION DATA (INCHES) IN CSV FORMAT

```
Station_ID,Year,Month,Day1,Day2,Day3,Day4,Day5 .... with additional fields up to Day30\r\n
1437,1960,1,0,0,.45,0,.81,0,0,0,0,0,.3,0,0,.22,0,0,0,.18,0,.15,.3,0,.26,0,.84,0,.4,0,2.11\r\n
1437,1960,2,0,0,0,0,.96,1.75,0,0,0,0,.34,0,.25,.08,0,0,0,0,1.24,.1,0,0,0,0,1.27,0,.37,0\r\n
1437,1960,3,.03,.09,0,.36,0,0,.03,0,.35,0,0,.49,1,.38,.91,.21,0,0,.27,0,.48,0,.01,0,.41,0,.22,0,0,.98\r\n
```

The first field of the synthetic data is the numerical ID of a fictional weather station, followed by the year and month of the precipitation observations. The remainder of each record contains 30 daily precipitation observations -- one observation for each day of the month -- separated from each other by commas. The character sequence “\r\n” appears at the end of each record. These characters – which are invisible if the file is opened in a text file editor such as “Notepad” – signify the end of a line of text or data.

After generating the CSV file, go to the Excel worksheet named “Read the Data in Chunks”. This worksheet executes a VBA macro that loops through the data file, reading one record in each cycle of the loop. The parameter in cell B9 indicates whether the program should simply read the data, or if it should both read the data and tabulate it, producing, as an output, a frequency distribution of daily precipitation amounts. On a typical PC, reading one million synthetic records takes only a couple of seconds, while both reading and tabulating one million records takes about 20 seconds. All else equal, the more computations a computer program performs, the longer it will take to run. Go ahead and run the program, experimenting with different settings for cell B9.

To view the VBA code that processes the synthetic dataset, right-click on the “Run” button, left-click on “Assign Macro”, and left-click on “Edit”. In Exhibit 3, a simplified version of the program is presented. In this program, “FileName” is a variable that stores the name of the file to be opened, such as “Synthetic\_Data.csv”. In VBA, it is possible to open multiple files simultaneously. Each file must be assigned a unique number. The data file is opened and is assigned number “1”. Opening a file for “input” indicates that the computer program will read the data in the file, while opening a file for “output” indicates that the program will send information to the file.

After opening the data file for input, a “do/while” loop is launched. The contents of the loop are repeatedly executed until the end of the data file is reached (“EOF” means “end of file”). In each cycle of the loop, another record is read using the command “Line Input #1, MonthlyRecord”. The “Line Input” command reads data until hitting the next “\r\n” character sequence, signifying that the end of a line has been reached. The line of data is stored in “MonthlyRecord” which is a “string” variable that can hold a sequence of letters, numbers, and/or other characters. A month’s worth of daily precipitation data is stored in “MonthlyRecord”.

To process the synthetic precipitation data – for example, to compute a frequency distribution for daily precipitation amounts – each monthly data record needs to be split into separate daily observations. In VBA, a string variable loaded with CSV data can be divided into separate fields using the “split” command, with a comma specified as the

field delimiter<sup>2</sup>. In this example, the resulting data fields are stored in the “DailyData” array. Recall that the first three fields of the synthetic data are station id, year, and month. These fields are stored in DailyData(0), DailyData(1), and DailyData(2), respectively. The 30 days of precipitation data run from DailyData(3) to DailyData(32).

After splitting the data into separate fields, a “for/next” loop is executed. This loop is embedded within the outer do/while loop. This inner loop cycles through each of the 30 days of data, counting those days on which there was non-zero precipitation.

In each cycle of the outer loop, the next monthly record is read, the monthly data is split into daily data, and the daily data is tabulated. The variable “RecordCount” tracks the number of records that have been read, and the variable “DaysWithPrecip” tracks the number of days on which precipitation occurred. After reaching the end of the data file, the do/while loop is terminated and the tabulation results are output to the screen.

### Exhibit 3

#### VBA CODE FOR READING AND TABULATING THE SYNTHETIC DATA IN CSV FORMAT

```

Dim Header as string
Dim MonthlyRecord as string 'a string variable holds a sequence of characters
Dim DailyData as variant 'a variant is flexible variable that can be recast into an array that holds multiple values

FileName = range("B6") 'get the name of the data file (the file name is stored in cell B6)
Open FileName For Input As #1 'create connection to data file
Line Input #1, Header 'read the file header which contains field names
RecordCount = 0

Do While EOF(1) = False 'initiate a do/while loop

    RecordCount = RecordCount + 1
    'read the next line of data and store it in the variable "MonthlyRecord"
    Line Input #1, MonthlyRecord

    'split the monthly record into separate data fields using commas as the field delimiter;
    'as a result of the split, the variable "DailyData" is transformed into an array that holds each of the separate fields
    DailyData = Split(MonthlyRecord, ",")

    'loop through the 30 days of daily data and count days with zero precipitation
    For d = 1 To 30
        If DailyData(d+2) > 0 then DaysWithPrecip = DaysWithPrecip + 1
    Next d

Loop

Close #1 'the data file has been fully processed, so close the connection

TotalDays = RecordCount * 30
PctWithPrpc = round(100*DaysWithPrecip / TotalDays, 2)

'output results
Msgbox("Total Monthly Records: " & RecordCount)
Msgbox("% of Days With Precip: " & PctWithPrecip)

```

Computer code is in blue font and comments are in red font

<sup>2</sup> “Split” can also be applied to a data string that uses tabs or any other character as the field delimiter. For example, if semicolons were used as the field delimiter, then the data string could be split into separate fields as follows: `DailyData = split(MonthlyRecord, ";")`

### 3. How to Iteratively Process a Fixed-Width Text File Using VBA

The previous VBA example reads and processes a data file in CSV format. Another format for text files is “fixed width”. Under this format, field delimiters such as commas are unnecessary. Instead, each field spans a fixed number of characters; consequently, its location is consistent across all records. This consistency permits the data to be read and processed by a computer program without the need for field delimiters.

Worksheet 1 of “Create\_Text\_File.xlsm” can generate a fixed-width data file by setting cell B8 to “Fixed Width” and then running the macro. The resulting file will have the format shown below:

#### Exhibit 4

##### SYNTHETIC DAILY PRECIPITATION DATA (INCHES) IN FIXED WIDTH FORMAT

Station_ID	YYYY	MM	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12	D13	D14	D15	\r\n
1651	1960	1	0.00	0.00	0.00	1.22	1.42	0.00	0.03	0.00	0.00	0.00	0.09	0.00	0.00	0.00	0.00	\r\n
1651	1960	2	0.00	0.00	0.11	0.00	0.00	0.00	0.00	0.10	0.00	0.00	1.35	0.00	0.00	0.94	1.09	\r\n
1651	1960	3	0.00	0.00	0.00	0.00	0.00	0.68	0.00	0.00	0.79	0.00	0.23	0.00	0.00	0.03	0.00	\r\n

Fields D16 through D30 – corresponding to precipitation on days 16 through 30 of each month -- were omitted for sake of brevity

In this example, columns 1 through 10 hold the station id, columns 12 through 15 hold the year, columns 17 and 18 hold the month, columns 20 through 23 hold precipitation on day 1, columns 25 through 28 hold precipitation on day 2, etc. For clarity, an empty space was inserted between each of the fields; thus, in effect, this is a space-delimited file in addition to being a fixed-width file. However, spaces are not needed in a fixed-width file; in fact, spaces might be viewed as wasteful, leading to an unnecessary increase in file size. In our example, however, spaces are a useful addition to make the data easier to visually decipher.

Just like a CSV data file, a fixed-width data file can be read line-by-line using VBA’s “Line Input” function. However, splitting each line of data into separate fields cannot be performed with the “split” function; rather, a different approach is required. One option is to use VBA’s “mid” function which has the following syntax:

extracted field = mid(string variable holding a fixed-width record, first column to extract, number of columns to extract)

For example, in the synthetic data shown in Exhibit 4, the year field runs from column 12 to 15, and can be extracted using the following code:

```
data_year = mid(MonthlyRecord, 12, 4)
```

In the VBA program in Exhibit 5, the fixed-width synthetic precipitation data is read line-by-line, and each record is split into its component fields using the “mid” function. After reading a monthly record, the program loops through each day of the month. For each day, the “mid” function extracts the associated data, storing it in the “precip” variable. If daily precipitation greater than zero, the program increments the variable “DaysWithPrecip”. To move onward to the next day of data, the “col” variable is increased by 5, which corresponds to the width of each of the 30 fields that store the daily precipitation data.



## Exhibit 5

## VBA CODE FOR READING THE SYNTHETIC DATA IN FIXED-WIDTH FORMAT

```

Dim precip as single 'this type of variable can store numbers that require decimal places

Do While EOF(1) = False 'initiate a do/while loop

    'read the next line of data and store it in the variable "MonthlyRecord"
    Line Input #1, MonthlyRecord

    station_id = mid(MonthlyRecord, 1, 10)
    data_year = mid(MonthlyRecord, 12, 4)
    data_month = mid(MonthlyRecord, 17, 2)

    'loop through the 30 day period, extracting each day's precipitation data

    col = 20 'the first day of precipitation data begins in column 20
    For d = 1 To 30
        precip = mid(MonthlyData, col, 4)
        If precip > 0 then DaysWithPrecip = DaysWithPrecip + 1
        col = col + 5 'jump forward to the next day of data
    Next d

Loop

TotalDays = RecordCount * 30
PctWithPrpc = round(100*DaysWithPrecip / TotalDays, 2)

'output results
Msgbox("Total Monthly Records: " & RecordCount)
Msgbox("% of Days Without Precip: " & PctWithPrecip)

```

## 4. How to Iteratively Process a NetCDF File Using R

NetCDF files are binary; consequently, if a netCDF file is opened using a text editor (like "Notepad") it will appear to be gibberish. While it is possible to write one's own code to decipher netCDF files, it is far easier to use a library designed for this purpose. In the context of computer programming, a library is pre-written computer code that performs a particular task(s), thus saving a programmer the trouble of reinventing the wheel. "ncdf4" is an R library that can both create and read netCDF files. We rely upon this library in the example that follows.

To learn how to use the ncdf4 library, please download the R program named "read\_netcdf.r" (hereafter referred to simply as "the R program"). Save the program to a subdirectory on your PC, and then open the program. Before running the program, you will need to change the "subdir" variable such that it points to the subdirectory in which you placed "read\_netcdf.r". This program loads the ncdf4 library using the following two lines of code:

```

install.packages("ncdf4")

require("ncdf4")

```

The library needs to be installed only one time; that is, you do not need to install it every time you run the program. Therefore, after running the program once, place a comment symbol ("#") to the left of "install.packages" to remove this line of code from execution. The "require" statement, however, should remain uncommented, because it needs to be executed each time the program is run.

A netCDF file is needed to illustrate how to use R's ncdf4 library. To this end, when you run the program, a small netCDF file named "fake\_data.nc" will be generated. This file will be saved in a subdirectory of your choice. In the program, please specify the desired subdirectory by editing the following line of code:

```
subdir = "C:/YourSubdirectory"
```

Next, run the R program. The program will generate “fake\_data.nc”, saving this file in the specified subdirectory. Like many gridded weather datasets, “fake\_data.nc” is dimensioned by longitude, latitude, and time. But unlike most weather datasets, this dataset is quite small, consisting of only 48 data values that correspond to 4 longitudes, 2 latitudes, and 6 time points (4 \* 2 \* 6 = 48):

Exhibit 6  
THE DATASET IN FAKE\_DATA.NC

Longitude (Degrees E)	Latitude (Degrees N)	Month 1	Month 2	Month 3	Month 4	Month 5	Month 6
250	30	111	112	113	114	115	116
255	30	211	212	213	214	215	216
260	30	311	312	313	314	315	316
265	30	411	412	413	414	415	416
250	35	121	122	123	124	125	126
255	35	221	222	223	224	225	226
260	35	321	322	323	324	325	326
265	35	421	422	423	424	425	426

While a weather dataset contains observations – such as temperature measurements in degrees Fahrenheit – “fake\_data.nc” contains numerical values that illustrate the rectangular dimensions of the data. Each of the dataset’s 48 values consists of 3 digits. The first digit – on the lefthand side of each 3-digit number – runs from 1 to 4, corresponding to the dataset’s 4 longitudes. The second digit runs from 1 to 2, corresponding to the 2 latitudes. The third digit – on the righthand side of each 3-digit number – runs from 1 to 6, corresponding to the 6 different points in time. For example, the value “325” corresponds to the 3<sup>rd</sup> longitude, the 2<sup>nd</sup> latitude, and the 5<sup>th</sup> time point. This numbering system is helpful for demonstrating how to extract specific chunks or subsets of the data. The ability to iteratively extract chunks of data is critical when dealing with a netCDF file that is too large to load into a PC’s memory. On a typical PC, a netCDF file that is over 3 gigabytes will either fail to fully load or will cause the PC to perform sluggishly; therefore, such a file must be processed in small chunks.

After generating “fake\_data.nc”, the R program demonstrates how to open a connection to the file, extract the information in the file header, and extract various slices of data. When embarking on an analysis of a netCDF file, the logical starting point is to examine the header because it describes the data’s key features, including its rectangular dimensions. The following code creates a connection with the netCDF file and outputs the file’s header:

```
nc <- nc_open("fake_data.nc")
print(nc)
```

The header of “fake\_data.nc” appears in Exhibit 7. The header indicates that the data has three dimensions: “lon” of size 4, measured in “degrees east”; “lat” of size 2, measured in “degrees north”; and “time” of size 6, measured in months. In the header, “size” indicates that the data contains 4 longitudes, 2 latitudes, and 6 points in time. The R program outputs the header to “header.txt” and outputs additional file metadata to “dimensions.txt”.

## Exhibit 7

## THE FILE HEADER OF "FAKE\_DATA.NC"

```

1 variables (excluding dimension variables):
  float FakeData[lon,lat,time]
    units: lon_lat_time
    _FillValue: -999

3 dimensions:
lon Size:4
  units: degrees_east
  long_name: lon
lat Size:2
  units: degrees_north
  long_name: lat
time Size:6
  units: months
  long_name: time

```

After outputting the file header, the R program illustrates how to extract slices of the data. This is a critical technique for processing netCDF files that are too large to load into a PC's active memory. The "ncdf4" library makes it easy to efficiently fetch user-defined slices of data. This is accomplished using the "ncvar\_get" function:

```
data_slice <- ncvar_get(nc, varName, start, count)
```

There are several parameters that feed into "ncvar\_get": "nc" is the connection to the netCDF file; "varName" is the name of the variable you wish to extract (for example, "precipitation"); "start" is the location in the rectangular N-dimensional dataset from which you wish to begin the extraction process, and "count" is an N-dimensional vector indicating the length of each dimension to be extracted.

Suppose we wish to extract a data slice from "fake\_data.nc" that contains all latitudes and longitudes for the first time point. This slice is obtained as follows:

```
slice <- ncvar_get(nc, "Fake_Data", c(1,1,1), c(4,2,1))
```

c(1,1,1) corresponds to the dataset's first longitude, the first latitude, and the first point in time, and can be visualized as the corner of a cube; c(4,2,1) stretches outward from this corner, spanning 4 longitudes, 2 latitudes, and 1 time point. Thus, this data slice captures the following 8 data values:

## Exhibit 8

## THE SLICE OF DATA (IN YELLOW) CAPTURED BY NCVAR\_GET(NC, "FAKE\_DATA", C(1,1,1), C(4,2,1))

Longitude (Degrees E)	Latitude (Degrees N)	Month 1	Month 2	Month 3	Month 4	Month 5	Month 6
250	30	111	112	113	114	115	116
255	30	211	212	213	214	215	216
260	30	311	312	313	314	315	316
265	30	411	412	413	414	415	416
250	35	121	122	123	124	125	126
255	35	221	222	223	224	225	226
260	35	321	322	323	324	325	326
265	35	421	422	423	424	425	426

An alternative syntax to capture the same slice of data is as follows:

```
slice <- ncvr_get(nc, "Fake_Data", c(1,1,1), c(-1,-1,1))
```

In the statement above, “-1” indicates that the data should be extracted from the specified starting point up to the upper limit of a particular dimension. Keep in mind that the triplet of values “c(x,y,z)” corresponds to longitude, latitude, and time, respectively. The order of these dimensions is evident from an inspection of the header shown in Exhibit 7. Specifically, the statement “float FakeData[lon,lat,time]” indicates that longitude is the first dimension, latitude is the second, and time is the third.

In the example in Exhibit 9, the full time series of data is extracted for the geographic location corresponding to the third longitude (260E) and the second latitude (35N). In Exhibit 10, a different slice of data is extracted, consisting of 3 longitudes, 1 latitude, and 4 points in time. Lastly, the R program provides additional examples that illustrate how to extract slices of netCDF data.

#### Exhibit 9

THE SLICE OF DATA (IN YELLOW) CAPTURED BY NCVAR\_GET(NC, “FAKE\_DATA”, C(3,2,1), C(1,1,-1))

Longitude (Degrees E)	Latitude (Degrees N)	Month 1	Month 2	Month 3	Month 4	Month 5	Month 6
250	30	111	112	113	114	115	116
255	30	211	212	213	214	215	216
260	30	311	312	313	314	315	316
265	30	411	412	413	414	415	416
250	35	121	122	123	124	125	126
255	35	221	222	223	224	225	226
260	35	321	322	323	324	325	326
265	35	421	422	423	424	425	426

#### Exhibit 10

THE SLICE OF DATA (IN YELLOW) CAPTURED BY NCVAR\_GET(NC, “FAKE\_DATA”, C(2,1,2), C(3,1,4))

Longitude (Degrees E)	Latitude (Degrees N)	Month 1	Month 2	Month 3	Month 4	Month 5	Month 6
250	30	111	112	113	114	115	116
255	30	211	212	213	214	215	216
260	30	311	312	313	314	315	316
265	30	411	412	413	414	415	416
250	35	121	122	123	124	125	126
255	35	221	222	223	224	225	226
260	35	321	322	323	324	325	326
265	35	421	422	423	424	425	426

To iteratively process a large netCDF file, the ncvr\_get function can be placed inside of a loop, as demonstrated in Exhibit 11. In each cycle of the loop, a slice of data is read and processed using ncvr\_get. The loop continues to execute until the entire file has been processed. The program begins by establishing a connection with “fake\_data.nc” and then loops across the data’s time dimension, which runs from 1 to 6. For each point in time, ncvr\_get fetches the data for all longitudes and latitudes. Because there are 4 longitudes and 2 latitudes, 8 data values are fetched each time ncvr\_get is executed. These values are loaded into the variable “slice” which

automatically adopts the form of a 2-dimensional array to match the shape of the extracted data. The data is then output to the screen, and the loop moves onward to the next point in time.

In this example, notice that the starting location for `ncvar_get` is `"c(1, 1, t)"`, where `"t"` is the looping variable. In the first cycle of the loop, `"t"` is equal to `"1"`, and, consequently, the data for time point 1 is extracted. In the second cycle of the loop, `"t"` is equal to `"2"`, and the data for time point 2 is extracted. In each cycle of the loop, `"t"` increases by 1. This process continues until we reach the end of the data's time dimension.

#### Exhibit 11

#### VBA CODE FOR ITERATIVELY READING SLICES OF FAKE\_DATA.NC

```
# open a connection to the netcdf file
nc <- nc_open("fake_data.nc")

# fetch the list of longitudes, latitudes and time points
lons <- ncvar_get(nc, "lon")
lats <- ncvar_get(nc, "lat")
dates <- ncvar_get(nc, "time")

# determine the number of time points in the dataset; by design, ndates will be equal to "6"
ndates <- length(dates)

# create longitude and latitude labels that we can use when outputting slices of the data
labels_lon <- paste(lons, "E", sep="")
labels_lat <- paste(lats, "N", sep="")

# loop across all of the points in time
for (t in 1:ndates)
{
  # extract data for all longitudes and latitudes for this particular point in time
  slice <- ncvar_get(nc, varName, c(1, 1, t), c(-1, -1, 1))

  # add row and column names to the 2D slice of data, and then output it to the screen
  colnames(slice) <- labels_lat
  row.names(slice) <- labels_lon
  print(slice)
}

nc_close(nc)
```

When applying this technique to a large netCDF file, experimentation is necessary to find the most efficient approach for slicing the data. Generally, for most netCDF files dimensioned by longitude, latitude, and time, the fastest runtimes are achieved by slicing the data by time, as illustrated in Exhibit 11. In this approach, time is the looping variable, and, for each point in time, `ncvar_get` fetches data across all longitudes and latitudes. Of course, such an approach may be incompatible with some research tasks. Suppose, for example, that our goal is to estimate a linear rate of increase across 50 years of temperature data, separately for each geographic location in the dataset. Given this goal, one cannot slice the data by time; rather, for each longitude/latitude pair, we need the entire 50-year time series. Therefore, the program will have to loop across longitude/latitude pairs, and, for each pair, fetch the associated time series and feed the data into a linear regression.

## 5. A Speed Test Comparing VBA, R, C++, and Python

The preceding examples use R and VBA, but these programming languages are by no means the only options for processing large weather datasets. Indeed, most programming languages can iteratively loop through large datasets. This raises the following questions: for processing large data files, are certain languages better than others? Is a single language sufficient to execute an entire research project? Or might it be advantageous to use one

language for looping through huge datasets and extracting the subset of data needed for a particular project, and a different language for performing statistical analyses of the extracted data?

Comparing the capabilities and performance of different programming languages is a complex topic. For the sake of brevity, we will not attempt a comprehensive discussion of this topic. However, as food-for-thought, we share the results of one experiment in which we measured the performance of VBA, R, Python and C++ with respect to the following task: loop through one million fixed-width synthetic precipitation records, each containing a month's worth of daily data, and calculate the percent of days on which precipitation was greater than zero. This task is by no means a perfect measuring rod for assessing a programming language's power to process large weather databases, and it does not include any statistical analysis; nevertheless, it can provide insight into the relative speeds of different languages.

For each of the four languages, we created a program with the same structure presented in Exhibit 5: an outer loop that cycles through the fixed-width text file reading it one monthly record at a time, and an inner loop that cycles through each monthly record one day at a time. These four programs are available for download on the SOA's weather guide webpage<sup>3</sup>.

We ran each of the four computer programs, tracking the time required to process one million monthly records. Exhibit 12 presents the results. C++ was the clear winner with a runtime of merely 6 seconds. R lagged far behind, with a runtime of 137 seconds (about 23 times the runtime of C++). VBA and Python performed much better than R, but not as well as C++.

#### Exhibit 12

##### TIME REQUIRED TO PROCESS A FIXED-WIDTH TEXT FILE WITH ONE MILLION MONTHLY RECORDS

Programming Language	Runtime in Seconds	Runtime as % of C++ Runtime
C++	6.0	100%
VBA	11.5	192%
Python	22.0	367%
R	137.5	2292%

The large gap between the runtimes for C++ and R is due, in part, to the fact that C++ is a compiled language while R is an interpreted language. When a computer program is "compiled", it is translated into an "exe" (executable) file that is written in machine language that can be executed directly by a computer's CPU (central processing unit). In contrast, an interpreted language does not have a compiler. Instead, as the program runs, the code is "interpreted" on a line-by-line basis. This is an additional computational burden that is absent from compiled code; consequently, interpreted code is usually sluggish compared to compiled code.

While neither VBA nor Python are compiled into machine language, they are compiled into intermediate languages that achieve some of the benefits of machine language. For this reason, VBA and Python tend to be slower than C++, but faster than R.

The results in Exhibit 12 reflect unfavorably on R, but the speed with which a computer program can perform a particular task is not only a function of the language in which it is written, but also the ability and knowledge of the programmer. To optimize the performance of R, it is best to avoid loops that iteratively process small amounts of

<sup>3</sup> The programs are named "speed\_test.r", "speed\_test.py", "speed\_test.cpp", and "speed\_test.xlsm", corresponding to R, Python, C++ and VBA, respectively.

data. Indeed, it was this approach that was employed in our speed test. For the R language, a more efficient approach is to iteratively process larger chunks of data – for example, processing ten thousand records in a single loop cycle, as opposed to processing just one record. This technique will be demonstrated in a future paper.

Speed of execution is but one consideration when evaluating which programming language(s) to use for the analysis of a large dataset. While C++ is fast, it lacks the many libraries available in languages such as R and Python. As discussed earlier, R and Python have libraries to process netCDF files, as well as libraries to perform statistical analysis (such as fitting generalized linear models to data). These libraries can dramatically reduce the time required to design a computer program – for example, a computer program that cycles through temperature data in netCDF format, estimating a linear temperature trend for each geographic grid point.

If one is willing to learn two programming languages, a good combination for the analysis of weather datasets would be C++ and either R or Python. This combination provides speed for preparing and reorganizing large datasets so that the information can then be processed by the statistical packages offered by R and/or Python.

## 6. Options for Dealing with Limited Hard Drive Space

Processing a large data file in small chunks allows a computer program to stay within a PC's RAM limitations (RAM is the active memory that performs calculations on data retrieved from storage). However, one must still contend with a PC's limited space for storing files. A typical PC has a hard drive of between 250 and 1000 gigabytes, but much of this space is consumed by the operating system and software such as Microsoft Office. Furthermore, an actuary or other professional may have hundreds or thousands of work-related files that consume additional space. Consequently, the remaining free space on a PC's hard drive may be insufficient to store a large weather dataset.

To illustrate options for working with a weather dataset that is too large to store on a PC's hard drive, we will describe an analysis recently performed by the Society of Actuaries (SOA) as a part of its ongoing climate research. The goal of the analysis was to estimate long-term temperature trends using ERA5, a high-resolution gridded dataset with worldwide geographic coverage. The dataset has worldwide coverage with over one million geographic grid points, with data in hourly time steps. The SOA's analysis focused on Canada and the United States (U.S.) across the 72-year period from 1950 through 2021. The analysis was performed on a standard laptop with 32 gigabytes of RAM and about 100 gigabytes of available hard drive space.

A year's worth of worldwide ERA5 data for one weather metric (such as temperature) amounts to about 20 gigabytes in netCDF format; thus, 72 years of data amounts to over 1400 gigabytes. Fortunately, while ERA5 data spans the entire surface of the earth, the web interface through which ERA5 data requests are submitted allows users to specify northern, southern, eastern, and western boundaries of the data that they wish to download. Because the SOA's study of temperature trends was focused solely on Canada and the U.S., we submitted a data request for a region running from 25N to 83N and from 52W to 172W (see Exhibit 12). This region captures the area of interest, while excluding areas that lay outside the scope of the analysis. Because the excluded area represents about 90% of the planet's surface, the size of the requested dataset was reduced dramatically, from 1400 gigabytes (for worldwide data) to just 140 gigabytes.

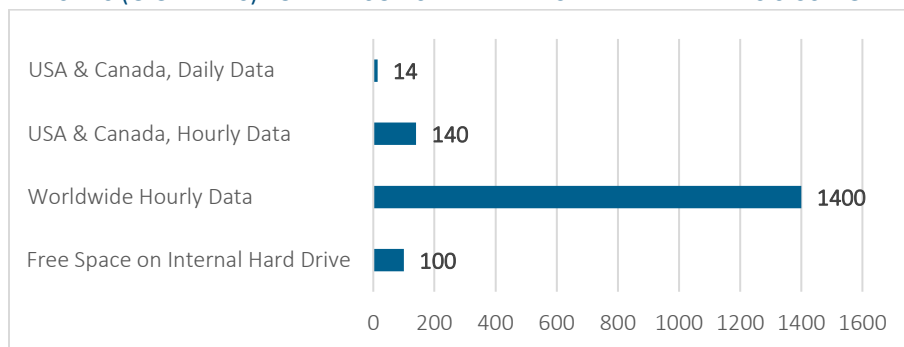
ERA5 data uses hourly time steps, but our temperature-trend research project did not require such high temporal resolution. For our purposes, data in daily time steps was sufficient. Specifically, for each geographic grid point, we needed a time series of daily minimum and daily maximum temperature. One possibility was to convert the raw data from hourly to daily each time we ran tabulations on it – that is, the conversion process would take place “on-the-fly” after reading the hourly data as an input. However, our research plan involved more than ten separate tabulation tasks, each using daily data as an input. Completing these tasks would have required us to repetitively convert the data from hourly to daily time units. This is a time-consuming process to execute across a large dataset. Therefore, it made greater sense to convert the data just one time, and then save it to the hard drive. Using this

approach, we generated two CSV files -- “DailyMaxTemp.csv” and “DailyMinTemp.csv” – each containing temperature data in daily time steps. Each of these two files is merely 7 gigabytes. While the hourly netCDF file is 140 gigabytes, the daily CSV files are merely 14 gigabytes in total. Thus, by reducing the temporal resolution of the data, we dramatically reduced the size of the dataset.

**Exhibit 13**  
**THE RECTANGULAR GEOGRAPHIC REGION SUBMITTED TO THE ERA’S DATA PORTAL**



**Exhibit 14**  
**FILE SIZES (GIGABYTES) FOR THE SOA’S TEMPERATURE TREND ANALYSIS USING ERA5 DATA**



When converting data from netCDF format to CSV format, the resulting values may have more decimal places than needed for a particular analysis. All else equal, the greater the number of decimal places, the greater the size of a CSV file. For our temperature-trend analysis, we rounded the data to 2 decimal places before generating the CSV files. Had we not rounded the data, the resulting CSV files would have been twice as large.

After converting the data from hourly to daily time steps, the resulting CSV files were relatively small (see Exhibit 14), allowing us to store them on our laptop’s hard drive as opposed to the external hard drive. Rather than deleting the original hourly netCDF file, we retained it on our external hard drive because it may be useful for future research projects.

This case study illustrates the following key ideas for working with large weather datasets: (1) if possible, when downloading the data, exclude geographic areas and time periods that lie outside of the scope of your analysis; (2) if



the data's temporal or spatial resolution exceeds what is required for your analysis, consider transforming the data to eliminate the excess resolution; (3) if you convert the data from netCDF to CSV format, round each value to eliminate excess precision that isn't required for the analysis; and (4) if the data cannot fit on your internal hard drive, store the data on an external drive, and design your tabulation programs to read the data from that drive.

## 7. A Preview of the Next Paper in this Series

The next paper in this series will provide an overview of the Global Historical Climatology Network Daily (GHCND) dataset, accompanied by an open-source computer program for analyzing the data. GHCND is hosted by the National Oceanic and Atmospheric Administration (NOAA) and is available for download from NOAA's website. GHCND is relatively small – about 30 gigabytes – making it easy to download and store. The data consists of daily temperature, precipitation and wind speed observations collected from over 100,000 land-based weather stations. GHCND provides geographic coverage of much of the world, but the availability of data varies significantly from one country to another, as well as within each country. In general, the density of weather stations is greatest in urban areas and lowest in rural areas.

## Appendix A. Reading Large Text Files: Examples in VBA, R, Python and C++

In the body of this paper, computer code for reading large data files was presented in VBA and R. Also, the VBA, R, C++, and Python computer code used for the speed test summarized in Exhibit 12 is available on our weather guide web page. In addition to these examples, it may be useful to briefly illustrate how closely the four languages (VBA, R, C++, and Python) resemble each other with respect to the syntax for looping through large text files. Below, in each of the four languages, code is presented that opens the synthetic precipitation fixed-width text file and loops through the data reading one monthly record at a time. For sake of brevity, the code does not perform any tabulations; rather, it simply reads the data.

<p><b>VBA</b></p> <pre>Open "Synthetic_Data.txt" for Input as #1 Do While EOF(1) = False     Line Input #1, MonthlyRecord Loop Close #1</pre>	<p><b>C++</b></p> <pre>std::ifstream iFile("Synthetic_Data.txt"); std::getline(iFile, FieldNames); while (std::getline(iFile, MonthlyRecord)) { } iFile.close();</pre>
<p><b>R</b></p> <pre>con = file("Synthetic_Data.txt", "r") EOF &lt;- FALSE while (EOF == FALSE) {     MonthlyRecord &lt;- readLines(con, n = 1)     if ( length(MonthlyRecord) == 0 ) { break } } close(con)</pre>	<p><b>Python</b></p> <pre>file = open("Synthetic_Data.txt", 'r') while True:     MonthlyRecord = file.readline()     if not MonthlyRecord:         break file.close()</pre>

## Appendix B. How to Install R, Python, or C++ on Your PC

VBA is integrated into Excel; therefore, if you have Excel on your PC, then you also have VBA. Thus, you do not need to install VBA to begin using it. In contrast, the other three languages discussed in this paper (R, Python, and C++) must be installed. If you use a PC issued by your employer, you may not have the freedom to install programs on your own; rather, you will need help from an administrator in your IT department.


R, Python, and C++ are available for free, and installation of each requires merely a few minutes. Installation instructions are available at these URLs:

### URLS WITH DOWNLOAD INSTRUCTIONS FOR EACH LANGUAGE

Language	URL with Installation Instructions
VBA	Included with Excel
R	<a href="https://cran.r-project.org/">https://cran.r-project.org/</a>
Python	<a href="https://www.python.org/downloads/">https://www.python.org/downloads/</a>
C++	<a href="https://visualstudio.microsoft.com/vs/community/">https://visualstudio.microsoft.com/vs/community/</a>


Note that C++ is part of a package named “Visual Studio”. This package includes several languages, one of which is C++.

## Feedback



**Give us your feedback!**  
Take a short survey on this report.

[Click Here](#)



## About The Society of Actuaries Research Institute

Serving as the research arm of the Society of Actuaries (SOA), the SOA Research Institute provides objective, data-driven research bringing together tried and true practices and future-focused approaches to address societal challenges and your business needs. The Institute provides trusted knowledge, extensive experience and new technologies to help effectively identify, predict and manage risks.

Representing the thousands of actuaries who help conduct critical research, the SOA Research Institute provides clarity and solutions on risks and societal challenges. The Institute connects actuaries, academics, employers, the insurance industry, regulators, research partners, foundations and research institutions, sponsors and non-governmental organizations, building an effective network which provides support, knowledge and expertise regarding the management of risk to benefit the industry and the public.

Managed by experienced actuaries and research experts from a broad range of industries, the SOA Research Institute creates, funds, develops and distributes research to elevate actuaries as leaders in measuring and managing risk. These efforts include studies, essay collections, webcasts, research papers, survey reports, and original research on topics impacting society.

Harnessing its peer-reviewed research, leading-edge technologies, new data tools and innovative practices, the Institute seeks to understand the underlying causes of risk and the possible outcomes. The Institute develops objective research spanning a variety of topics with its [strategic research programs](#): aging and retirement; actuarial innovation and technology; mortality and longevity; diversity, equity and inclusion; health care cost trends; and catastrophe and climate risk. The Institute has a large volume of [topical research available](#), including an expanding collection of international and market-specific research, experience studies, models and timely research.

Society of Actuaries Research Institute  
475 N. Martingale Road, Suite 600  
Schaumburg, Illinois 60173  
[www.SOA.org](http://www.SOA.org)