Paper 201-2023

How Do We Git There?

Best Practices for Using Git With SAS®

Joe Matise, NORC at the University of Chicago

ABSTRACT

Git is a powerful tool for managing version control that is commonly used in professional programming environments as well as by researchers, academics, and students across the globe. It can be, however, somewhat overwhelming for new programmers, particularly those who do not have exposure to it through other languages.

In this presentation, we will give a brief introduction to Git, tour a few tools for helping to manage Git workflows, and explain best practices for using Git in a SAS programming environment.

This presentation is aimed at novice Git users, regardless of SAS programming level; it requires no previous experience with SAS or Git.

INTRODUCTION

Git is a powerful solution for source code version control that is widely used in software development at all levels, from open-source software hosted on the popular Github repository to many Fortune 500 companies, and has an 97% marketshare among professional developers according to the 2022 Stack Overflow Developer Survey.

SAS Developers can be more productive, more collaborative, and more organized programmers by using Git in conjunction with their daily workflow. Git increases productivity by helping a developer to organize different streams of code, manage versions, and more easily spot changes. Developers using Git are more collaborative due to the ability to work on one program or set of programs at the same time, effortlessly merging their changes together and spotting areas they need to discuss. When used properly, Git provides an excellent way to organize code through the use of branches, allowing developers to move from testing to production or to manage different variants of code with minimal work needed.

Modern SAS user interfaces have enhanced Git integration, from the built-in Git support in Enterprise Guide, SAS Studio, and Data Integration Studio to Git functions in the SAS data step language. Beyond integration, SAS programs and projects can also be managed in Git the old-fashioned way: through the command line or various first- or third-party Git GUI tools, such as SourceTree, GitKraken, or GitHub Desktop.

WHAT IS GIT?

Git is a “free and open source distributed version control system” (https://git-scm.com), developed in 2005 by the creator of Linux, Linus Torvalds, initially to manage the source code for the Linux kernel. It allows a developer to periodically “commit” code to a repository, which keeps track of the changes to that code across each commit. It includes features such as branching, merging, and the ability to identify who made a change (“blame”).

DISTRIBUTED, YET CENTRAL

Git differs from older source control methods in that you store your changes on your local folder, in a *local repository*. For some users, that local repository might be literally on their own machine; for other users, the local repository might be in a network file share (for example, if you are running SAS Studio, your server most likely cannot see your local machine). Either way, the storage is local to the code – usually in a file that resides in the same folder as the root folder of the project – and usually accessible only to that user. The local repository keeps track of all changes to the code and should be updated frequently – ideally several times a day, any time a meaningful change is made.

How do users collaborate, then? Git uses a central server to store a copy of the repository. This server, called a remote repository, hosts a copy of those versions that its users have decided to share with each other. This remote repository might be updated every few days, or less or more often depending on your collaboration schedule. This process requires authenticating to the server, sending it a copy of your changes, and asking the server to verify your changes do not conflict with anyone else's code changes that have happened since you pulled the data down last.

USING GIT: THE BASICS

The primary unit in Git is the repository, or repo, which is the local data store that holds your code. You can think of this as a small, local database which your code is stored in with each differential from a previous version stored under a different version number. Repos usually hold only code and similar text-based files; you do not typically store data files in Git, though it is possible to do so. Since Git is unable to comprehend the contents of a binary file (such as an excel file or a sas7bdat dataset), you lose most of the advantage of Git when storing such a file, in addition to the significant space required to store a larger dataset.

In order to version your code, you perform an action called a commit, which tells Git to take a snapshot of the code you select and store the differences from the last commit. You can commit your code as often as you want – typically this would be at least as often as a major change is made to it.

Since repositories are local, you need to connect that repository to a remote server if you want to share it with other users. Rather than communicate to the remote server each time you commit, you have a separate action called pushing, which takes the current status of the repository and syncs it to the server. There is a related action called a pull, which is when you ask the server to sync the status of a repository with your local copy. This allows you to get changes other users made to the code.

These three actions – commit, push, pull – are by far the most common actions taken, and much of the time will be the only actions you use in a day. There are, however, several other actions you can take; the main ones we will discuss here are branching and merging.

Branching is the process of taking a copy of a repository and splitting its history from the main code's history. This is very helpful for a number of reasons: it allows you to easily create separate versions, or to make changes to code in a separate space from where another user might be working without having to worry about their changes until later. Branches might be short term – created just to test a fix to one feature – or might contain an entire version and never be merged back to the main branch.

Merging is the opposite of branching – you take two branches and combine them, working out any overlapping differences as you go. Merging can occur in other situations as well, but most commonly involves multiple branches.

A typical daily workflow with Git might go something like this:

- Create a branch from main, name it “v2_test” on the remote server
- Pull and check out that branch to your local machine
- Make some modifications to the code
- Commit those changes to the repository
- Push those changes to the remote server
- Create a Pull Request, which begins the merge process and asks for review of your changes
- Once the changes are approved, merge the branch back to main.

STARTING OUT

When starting out with Git, the first thing to do is to initialize the repository. This creates the repository, creates some default preference files, and tells Git what folder(s) contain the code you want to store.

Once that repository is initialized, you can then commit code, connect the repository to a remote server, push that code to the server and share it with your co-workers!

How you structure your folders prior to initializing a repository will drive how the repository is itself structured, and thus what your team will see when they check out your code. It is important, therefore, to be highly organized in how you structure your folders.

CODE IS CODE, DATA IS NOT CODE (EXCEPT WHEN IT IS)

The number one thing to consider is how to store code and data, when they are normally stored nearby. While some SAS programmers will store their data far away – in a database, perhaps – it is common to have code and data live near each other. Since data does not go into Git, the first structural guideline is to have a data folder that is separate from your code folder. When you initialize the repository, you will initialize it in the code folder, not the higher level folder that contains both. In the image below, you would initialize the repository in the folder named “src”, not in the folder named “Project”. This ensures your data (and other non-code items) are not accidentally stored in Git – which not only would be wasteful, it could violate your company’s security policies or agreements with your clients.

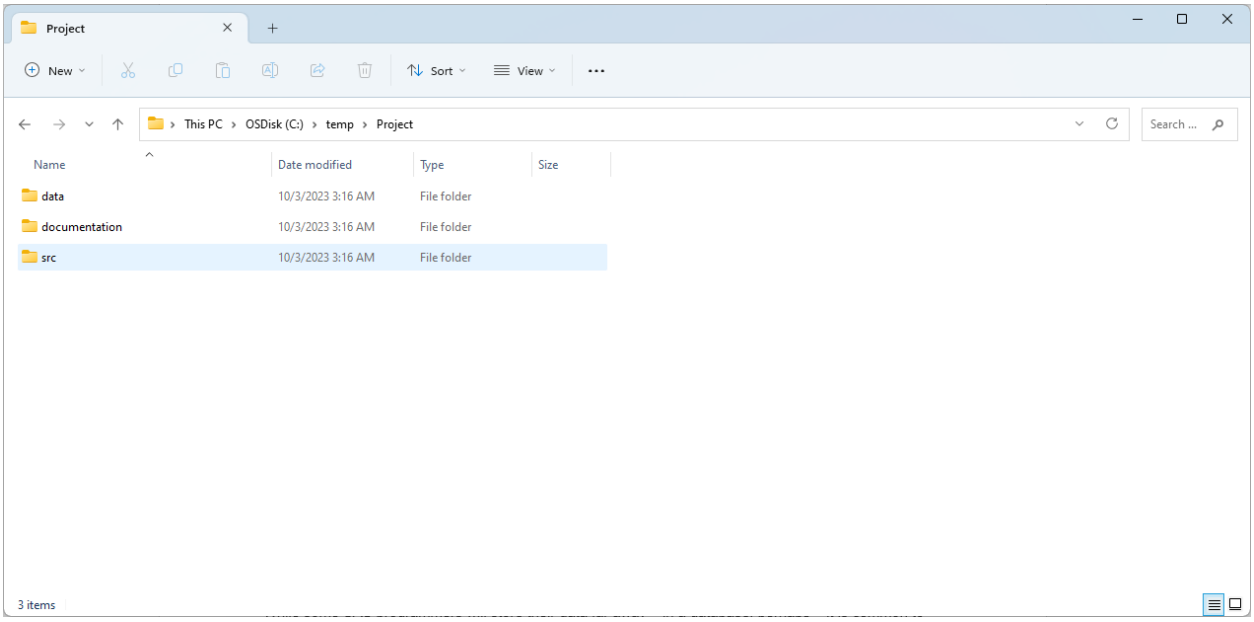

Figure 1. Project Folder

After initializing Git in the src folder, you will be presented with an empty folder with one hidden directory inside (if you enable Windows to show hidden folders):

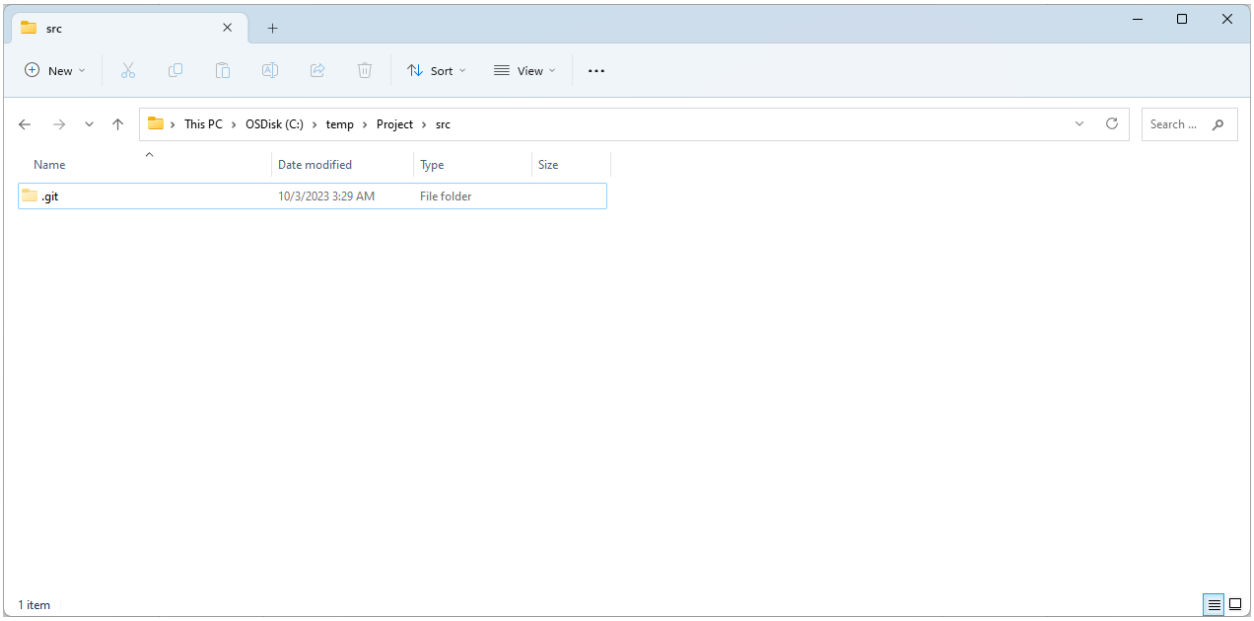

Figure 2. Source Code Folder with Git initialized

That hidden directory contains the git repository, as well as some configuration details.

Inside the src folder, you should further organize your code into subfolders that separate different kinds of code. This not only is a good organizational practice in general, but it will make it easier to understand the merge comparison, and may help if you have different people with different specialties that are likely to be editing code that is relevant to them.

For example, you might create folders for SAS, Powershell, and Python. You further might separate your primary SAS programs from your macros, to make it easier to port the macros to other projects.

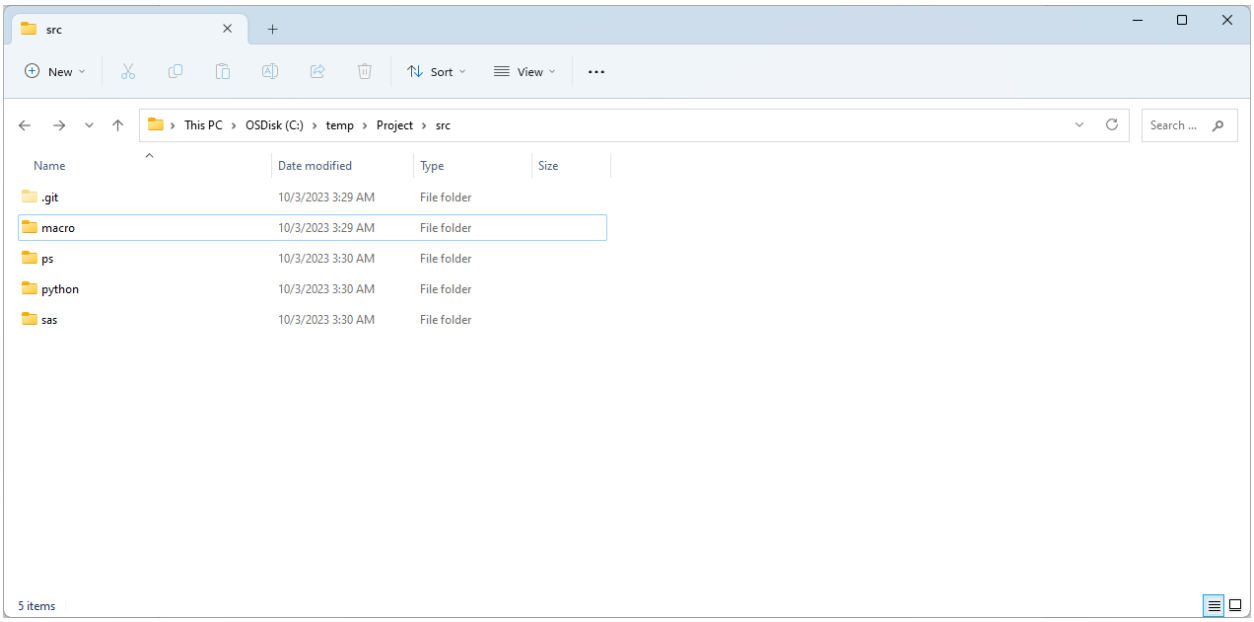

Figure 3. Source Code Folder with various folders for programs

A few other folders make sense at this level as well. Configuration files are helpful to separate from the other code elements, as they might need to be updated periodically; so a config directory makes sense. I also like to have an env directory, which is short for environment. This is a specialized kind of configuration folder: it stores files that are specific to the environment, such as test or production, that are not stored in Git. For example, you might store database connection information that is specific to the environment, or API tokens, or similar files.

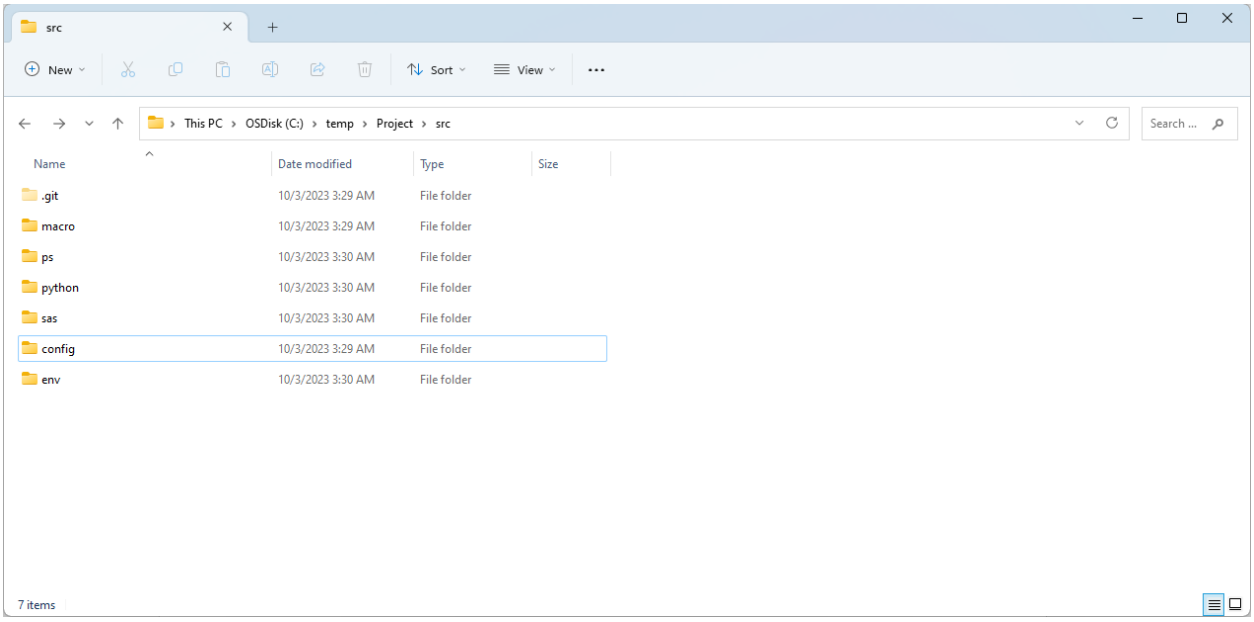

Figure 4. Source Code Folder with configuration and environment folders

You exclude the folder from being tracked in git by using a .gitignore file; this file contains one or more patterns that Git will check when evaluating what files to track. Any file that matches that pattern in that subfolder will be ignored.

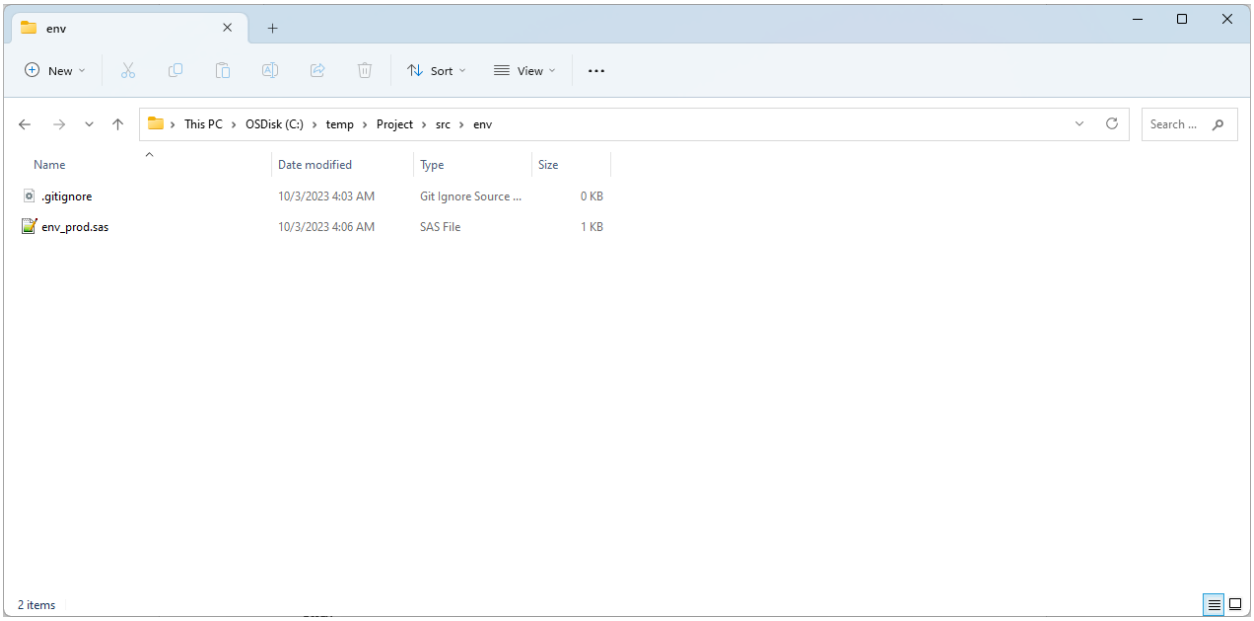

Figure 5. Environment folder with .gitignore example

One other consideration when using Git is that your folder structure will persist across different checkouts of the code, but the absolute path will not necessarily be the same. A user should be able to check out code into any directory they wish, and expect it to still function. As such, when referring to code (such as macros) that is included in this folder structure, you should either use relative paths, or include a reference to the path in the environment file if it is impossible to use relative paths. Alternately, you may be able to determine the path of the code itself while executing; for example, in SAS 9.4 there are macro variables that indicate the path of the current program in some environments.

WHAT ABOUT NON-CODE FILES?

Data-driven SAS programming often includes files that are not programs *per se*, but are nonetheless important elements of your application. These files may change over time, and you may desire to track them as they do.

If those files are stored as binary file formats, such as excel files, you may store them in Git alongside your programs, but you will not gain the advantages of Git – as you cannot see the difference over time, only the entire program. This is not ideal, although of course it sometimes is the best compromise.

Better would be to store the files as text formats – such as .csv files. This allows them to be tracked in Git just as you would your program files, and see the differences as they change (also allowing them to be stored more efficiently).

One option might be to keep an Excel copy of the file to simplify the user experience, but as part of your program extract that to a text file format. This would allow you to get the best of both worlds – ease of use and ease of tracking.

Otherwise, these files can also be treated like data, and stored in the data folder, subject to your organization's data management protocols. This may be simpler, particularly if the files are closely related to the data stored in that space.

TOOLS FOR USING GIT

Git can be thought of as having two software components: the programs that actually perform the actions (commit, pull, push, etc.), and the user interface. Sometimes the user interface will include the basic programs that perform the actions, but I recommend installing it separately so you have control over that installation.

Linux users will find installing Git trivial: it can be installed from their package manager of choice, like most common programs. Ubuntu users, for example, simply need to use apt to install it:

```
sudo apt install git-all
```

Windows users may also use a package manager (such as chocolatey) to install git, but users accustomed to running an installer to install a program may prefer to download it directly. You will want to download Git for Windows, available from https://gitforwindows.org/ as an executable installer. Git for Windows includes a basic user interface and a BASH-like command line shell (called Git BASH) in addition to the core Git tools.

Depending on your preferred remote server platform (discussed later), you may find a Git UI that is appropriate; for example, if you use Bitbucket by Atlassian, they make an IDE called SourceTree that is designed to integrate well with Bitbucket. In general, most of these IDEs are compatible with any Git client or server, though, so use what you feel comfortable with – or don't use anything beyond the basic installation; many users who feel comfortable at the command line do so.

Some choices:

- SourceTree (https://www.sourcetreeapp.com/)
- GitKraken (https://www.gitkraken.com)
- GitHub Desktop (https://desktop.github.com/) (Only compatible with GitHub)

Another option that many users find simpler is to use your IDE (Integrated Development Environment) to interface with Git. SAS developers can connect SAS Studio or Enterprise Guide to their Git repositories; developers who use Visual Studio Code can do the same in that program. Git integration is table stakes for an IDE these days, and most likely your favorite editor supports Git directly. Yes, even UltraEdit users can take advantage of Git integration, and perform most Git functions directly inside the editor! (https://wiki.ultraedit.com/Git)

PROGRAM DESIGN FOR GIT

While it is not necessary to change how your programs are structured to use Git, a more modular program design will be easier to use than using single long programs. Chunking your code into multiple pieces where appropriate will both simplify the merging process and allow for better unit testing, when you have one program per unit, rather than many units in one program.

How you split a program up largely depends on its function. Identify discrete pieces of code you might want to run by themselves; those likely should be separate programs. No, you do not need to use a separate program for each data step; think about what a logical grouping of steps might be such that you frequently would want to run them, then check their output.

One example of this might be a workflow that I would use to perform a data delivery, from extracting data from the survey platform to creating the final delivery dataset and reports.

- Survey_extract.sas
    - Calls several macros to interface with the survey engine
    - Output: the raw survey dataset, transformed to a shape that can be easily processed further
- Data_cleaning.sas
    - Calls several Data Cleaning macros to perform typical data cleaning activities
    - Includes several .csv files that contain instructions for cleaning (such as skip patterns or missing data imputation)
    - Output: a cleaned survey dataset
- Survey_Weighting.sas
    - Calls a macro that performs survey rimweighting
    - Output: a dataset of survey weights
- KPI_Report.sas
    - Creates a simple KPI Report
    - Output: the KPI Report
- Data_Delivery.sas
    - Performs the final data delivery tasks (formatting, labelling, etc.)
    - Calls several macros related to data delivery
    - Output: the final deliverable dataset

This is a reasonable level of structure that splits up the program into logical units that can each be separately run and tested.

CONCLUSION

Using Git with your code or any other language can improve your productivity and simplify collaboration with others, particularly if you follow a few basic guidelines to keep your code well organized and structured. A little effort at the start will go a long way to improving your experience with version control.

ACKNOWLEDGMENTS

I appreciate the support of my manager, David Trevarthen, in attending this conference and in continuing to develop my SAS skills even as I move towards using them less. I also appreciate the support of my colleague Zeke Torres for his continued efforts to convert everyone he can to become a Git user!

RECOMMENDED READING

- *Git and SAS*, Joe Matise, Proceedings of SAS Global Forum 2021; available at https://github.com/snoopy369/presentations/tree/master/Git%20and%20SAS
- *GitRDone,* Joe Matise, Proceedings of WUSS 2019; available at https://github.com/snoopy369/presentations/tree/master/GitRDone

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Joe Matise
NORC at the University of Chicago
matise-joe@norc.org