TITLE

Digital Forensic Data Analysis Using Mathematica

/STAND FIRST

A Tutorial

/DISCLAIMER

Neither of our respective employers has endorsed or vetted this article. It has been done entirely without any involvement from on behalf.

/BODY COPY

/Introduction

Today, digital forensic investigators and analysts likely use a multitude of software tools during the course of a typical investigation. Some are free and open source, others free and closed source (freeware), and others commercial in nature. Unfortunately, many of the commercial forensic software are typically employed are expensive, sometimes excessively so, often for software capable of performing only a few data processing or analysis tasks.

Mathematica, a commercial mathematics and symbolic computing software framework, is an entirely different creature from what investigators and analysts typically use. It is unlike anything in the investigator's software toolbox. Complete with very extensive data processing and analysis capabilities, it includes wide-ranging support for image processing and analysis.

While our ultimate goal is to show you how Mathematica can be used for wide ranging forensic image analysis, a tutorial is required because it is very different from what most investigators are used to. So, in this first of several articles, we will provide a hands-on Mathematica tutorial to get you up and running and to prepare you for doing real analysis and data processing in our follow-up articles.

/Why use it?

We use Mathematica because it is uniquely qualified to perform complex image processing and analysis, all from a single software framework. But why Mathematica? Because it is a first-rate data processing and analysis facility, with a moderate learning curve. For digital forensic analyses, Mathematica does not require any special add-ons or third-party products to provide the necessary features and capabilities investigators and analysts will likely need.

Other similar commercial frameworks are available, many of which require the purchase of additional packages, thus dramatically increasing the base software package's price. Various open source alternatives exist, some of which range from very good to excellent. However, none of them provides the extensive image processing, general data processing, and analytical capabilities of Mathematica.

In short, do not use it for what other specialized tools already do so well; use it for what they can't do.

Throughout this series of articles, we will be looking at how to get started with Mathematica and introduce various concepts using easy to read code. Mathematica supports various programming paradigms, including functional and procedural, but where possible, we'll be using procedural as this is what most people are familiar with. Anyone who has ever written even simple programs in C will be able to understand what it is we are trying to do. For those interested in functional programming, which is in many regards more capable than procedural languages, Mathematica's built-in documentation is a great place to get started, but it has a sufficiently steep learning curve.

/Expectations

In each article, we will provide running code that can be run directly in one's own instance of Mathematica. We wanted to use a single software framework, which would not require add-on components, to perform both analysis and data processing, and the code is an essential part of this. Of course, it also makes the articles easier to read because the code is simple enough to be readily understood, at least for the procedurally based code.

However, because of length concerns for this article we cannot provide many program output listings; we have incorporated only those essential to the reading of this article. To see the outputs you will need to a copy of Mathematica to play with.

In subsequent articles, we will be examining issues including multilingual OCR (i.e., text extraction) and language recognition, image content identification, and image classification according to user-specified sample sets. From these articles, you will have all the knowledge you need to do so much more. However, at this point, it has come to our attention that there are limitations to Mathematica's OCR and language recognition, but we are endeavouring to find ways to work around these issues when we explore these topics in detail. We also plan to write several non-image processing based articles

All these capabilities and so much more are readily available to the user, including a very extensive documentation library that details the thousands of functionalities available in Mathematica. Moreover, many of its built-in features can be readily parallelized with little to no additional work on the part of the user. However, what we will be looking at in these articles will be done in serial fashion, so the parallelization of the various processing loops (i.e., FOR, WHILE, DO), if possible (which we have not explored), is left for others to implement.

Additional topics can always be looked at. As time goes by, we will occasionally try to write follow-up articles exploring other Mathematica capabilities. And of course, if you have suggestions or comments, all you have to do is drop us a line; just use the email address found at the end of the article.

/Things to consider

Mathematica's learning curve can be steep so prior experience with similar software, image processing, data analysis or "big data" will definitely be helpful. Fortunately, any procedural or functional programming experience will significantly lessen the curve.

Mathematica can also be very taxing on available system resources. That is why high-end workstations, with lots of memory and sufficient CPU power, are highly recommended. These additional capabilities will significantly help to improve the overall experience. To its benefit, out of the box, Mathematica supports 8 CPU cores and places no limit on the amount of memory it can use. However, the actual version of Windows in use will ultimately dictate just how much memory can be used. Consult [1] for more information regarding the Windows' ability to support varying sizes of memory.

It is available for commercial, student, academic, online and home use. A trial version is readily downloadable for a limited period to allow would-be users an opportunity to assess its suitability and integration into existing workflows and software toolboxes. Whichever version is used, they all have the same data processing and analysis features.

/Test computer system specs

To perform and test all the functionality that we will write about in this series of articles, we needed a capable test system to experiment with to ensure that our code is both accurate and functional.

For our tests, we used a customized workstation equipped with 2 Xeon 2630v3 processors running at 2.40 GHz, with 128 GiB RAM running atop a SuperMicro X10-DAL-I motherboard. Swap was placed onto a Kingston SV300S3 SSD raw partition, also set at 128 GiB. An NVidia GeForce 720 2 GiB video card supported system graphics. Five internal 1 TB hard drives were used for various data storage needs, but no RAID was configured. The system was installed using Fedora 23 x64 Linux, then heavily customized and updated, currently running kernel 4.4.3-300 SMP.

At the time of this writing, the most recent version of Mathematica is 10.4.0.0, which was fully installed to the system disk.

Non-Mathematica commands used in this article will work under Linux and UNIX and should work under Mac OS X. However, for Windows systems, while the concepts are the same third-party applications will be required.

/Handling the data processing load: Benchmarking with Mathematica

The first order of business is to baseline the underlying system's performance. Benchmarking will give the user a "feel" for how quickly they can expect CPU-bound processing and analysis to proceed. It is important to determine this information because processing many thousands of images may take a considerable amount of time. Thus, to ascertain your system's overall Mathematica based processing capabilities, copy and run **Code Listing 1** into a new Mathematica notebook.

```
(* BENCHMARKING NOTEBOOK *)

(* Pre-Benchmark Actions *)
ClearSystemCache[]
ClearAll["Global`*"]

(* Run Single CPU Core Benchmark *)
Needs["Benchmarking`"]
BenchmarkReport[]

(* Run Parallel Benchmark (Default=8 cores/kernels
   Mathematica will complain about running Benchmarkreport[]
   sequentially. It still produces meaningful results. *)
Parallelize[BenchmarkReport[]]
```

Code Listing 1: Benchmarking code

Looking over **Code Listing 1**, the overall functionality should be clear. We use "`(*  *)`" to insert comments to improve the code's overall readability. Those who have done C/C++/Java programming will remember using "`//`" or "`/*  */`" for program commenting and documentation.

We run `Benchmark[]` twice, the first unmodified while the second is run inside of `Parallelize[]`. The purpose of running the two benchmarks is to highlight the differences between parallel and non-parallelized operations. The first instance runs on a single CPU core while the second runs on up to 8 cores. It could be more if your version of Mathematica supports more than the default number of cores or less if your system has less than 8 cores.

The `Needs[]` statement is responsible for loading the necessary Mathematica package from disk required for system benchmarking.

Mathematica runs one kernel instance per CPU core, up to its maximum default amount (8), or less, depending on the actual number of cores. The Mathematica kernel is responsible for performing the actual work.

To run this or any other code in a Mathematica notebook, bring the mouse to bear somewhere within the code, not above or below, click inside the code just once, and then press SHIFT+ENTER, which will run the code. The two benchmarking instances will execute a series of calculations that will give an overall feel for how quickly Mathematica will run on the current system.

The most important information resulting from `BenchmarkReport[]`is its graphical system comparison, which will indicate how well the current system performed against other modern x86 systems. The current system is highlighted in blue. The larger the number resulting from the benchmark the better the performance is. Example benchmarks were run against one and eight CPU cores, as shown in **Figure 1** and **Figure 2**, respectively, using our test system.

Our test system performed well in the single core benchmark category, shown in in **Figure 1**. When examining **Figure 2**, we see that there are no multi-core comparisons available. Nevertheless, its results give an appreciation for a how multi-core system taking advantage of parallelized operations may fare.

/Sources of information when starting out

Mathematica is neither difficult to start up nor to begin rudimentary data analysis, as its extensive help library can be consulted at any time. The majority of questions that beginning investigators and analysts will have can usually be answered by looking at Mathematica's extensive documentation library, available both in the product itself and online. In addition, many books have been and continue to be written on Mathematica, its extensive list of features, how to program with it or solve a wide-ranging series of scientific and technical problems (e.g., math, physics, engineering, etc.). There are also various online forums dedicated to Mathematica and technical support is available directly from Wolfram, which is included for Premier Service subscribers.

There is also the Wolfram Demonstration Project that currently boasts more than 10,000 demos. They represent a very wide spectrum of science, engineering, math and technology. In looking at their code and then running them you will become better at recognizing what the different Mathematica functions do and how to string them together to write useful programs.

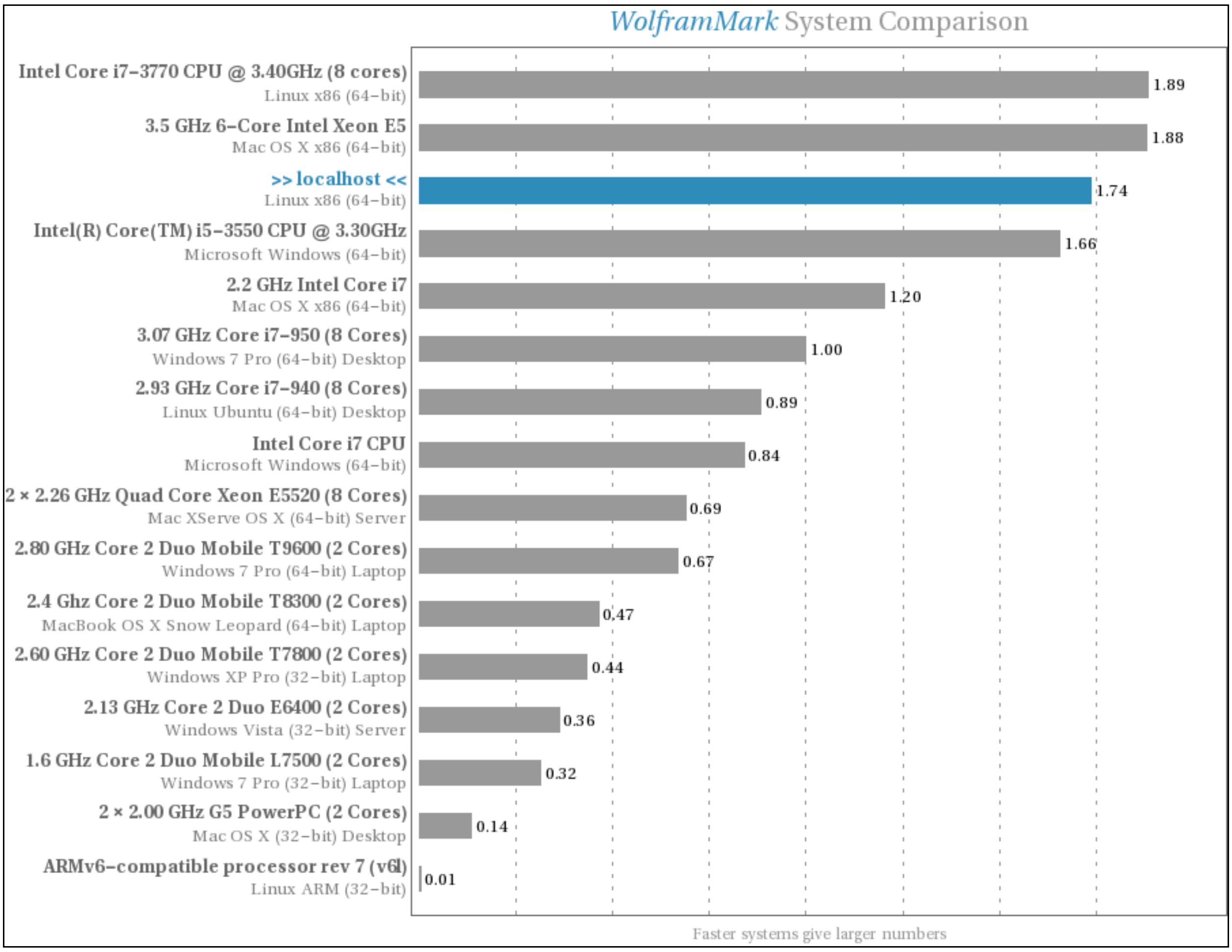

Figure 1: Benchmarking results for a single core

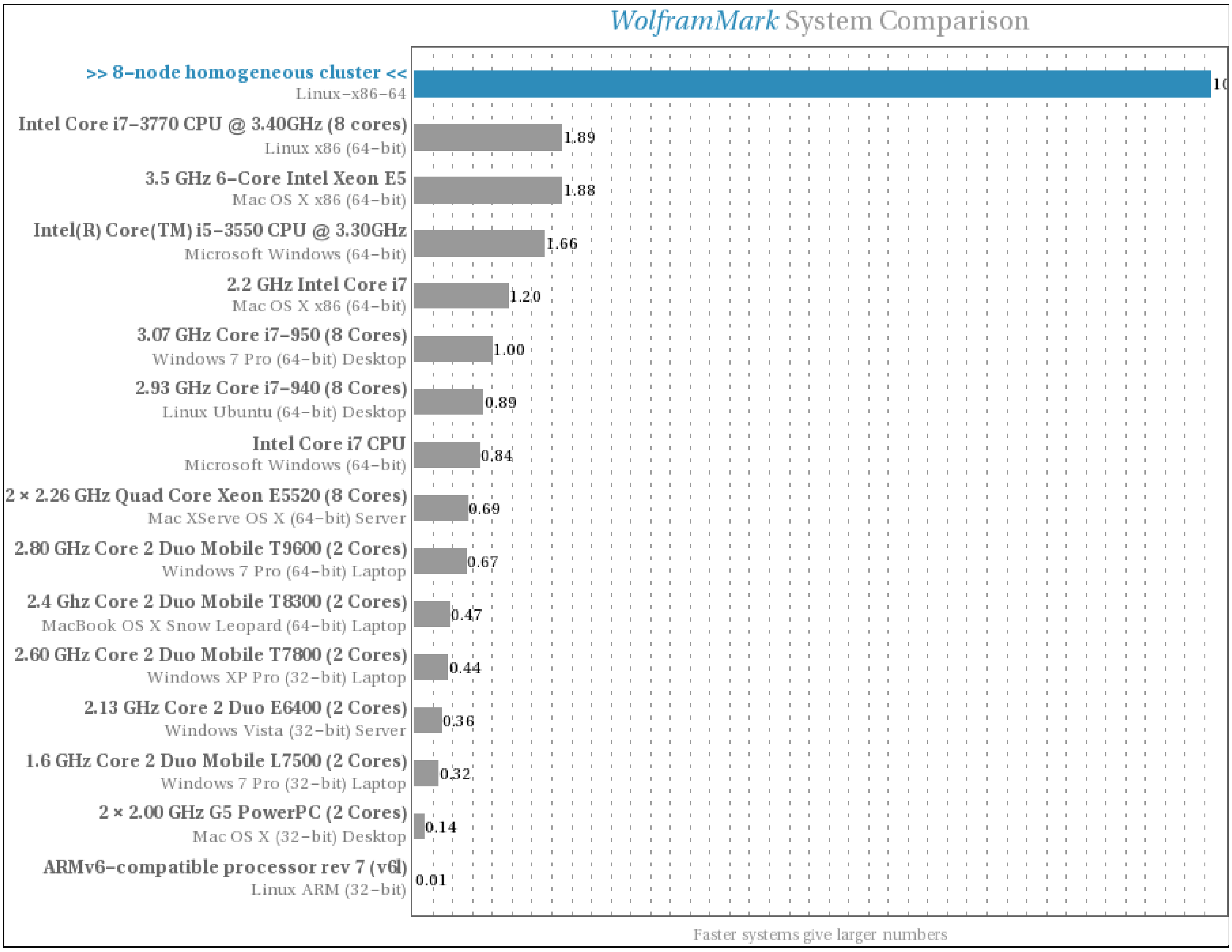

Figure 2: Benchmarking results for eight cores

/Working with file formats

The first thing to do before writing a useful program is to identify which file formats are supported by the current version of Mathematica. In a new notebook, run the simple code shown in **Code Listing 2**.

```
$ImportFormats
```

Code Listing 2: Querying currently supported file formats

Running **Code Listing 2** a listing of the currently supported file formats will be sent as output to the notebook, as shown in **Output Listing 1**.

```
Out[1] = {"3DS", "ACO", "Affymetrix", "AgilentMicroarray", "AIFF",
       "ApacheLog", "ArcGRID", "AU", "AVI", "Base64", "BDF", "Binary",
       "Bit", "BMP", "Byte", "BYU", "BZIP2", "CDED", "CDF", "Character16",
       "Character8", "CIF", "Complex128", "Complex256", "Complex64",
       "CSV", "CUR", "DBF", "DICOM", "DIF", "DIMACS", "Directory", "DOT",
       "DXF", "EDF", "EPS", "ExpressionML", "FASTA", "FASTQ", "FCS",
       "FITS", "FLAC", "GenBank", "GeoTIFF", "GIF", "GPX", "Graph6",
       "Graphlet", "GraphML", "GRIB", "GTOPO30", "GXL", "GZIP",
       "HarwellBoeing", "HDF", "HDF5", "HIN", "HTML", "ICC", "ICNS",
       "ICO", "ICS", "Integer128", "Integer16", "Integer24", "Integer32",
       "Integer64", "Integer8", "JCAMP-DX", "JPEG", "JPEG2000", "JSON",
       "JVX", "KML", "LaTeX", "LEDA", "List", "LWO", "MAT", "MathML",
       "MBOX", "MDB", "MESH", "MGF", "MIDI", "MMCIF", "MOL", "MOL2",
       "MP3", "MPS", "MTP", "MTX", "MX", "NASACDF", "NB", "NDK", "NetCDF",
       "NEXUS", "NOFF", "OBJ", "ODS", "OFF", "OGG", "OpenEXR", "Package",
       "Pajek", "PBM", "PCX", "PDB", "PDF", "PGM", "PLY", "PNG", "PNM",
       "PPM", "PXR", "QuickTime", "Raw", "RawBitmap", "RawJSON",
       "Real128", "Real32", "Real64", "RIB", "RSS", "RTF", "SCT", "SDF",
       "SDTS", "SDTSDEM", "SFF", "SHP", "SMILES", "SND", "SP3", "Sparse6",
       "STL", "String", "SurferGrid", "SXC", "Table", "TAR",
       "TerminatedString", "Text", "TGA", "TGF", "TIFF", "TIGER", "TLE",
       "TSV", "UnsignedInteger128", "UnsignedInteger16",
       "UnsignedInteger24", "UnsignedInteger32", "UnsignedInteger64",
       "UnsignedInteger8", "USGSDEM", "UUE", "VCF", "VCS", "VTK", "WAV",
       "Wave64", "WDX", "WebP", "XBM", "XHTML", "XHTMLMathML", "XLS",
       "XLSX", "XML", "XPORT", "XYZ", "ZIP"}
```

Output Listing 1: Currently supported file formats

From this list, identify the file types you want to work with. For example, let’s say we only want to look at JPEG, PNG and GIF file formats, typical of images used on the web, and which would likely reside on an Internet connected computer. Mount the evidence disk image on the current computer system read-only using an appropriate mount command (e.g., raw disk image, EWF, AFF, VMDK, etc.) and mounting it to a corresponding mount point (e.g., /media/evidence_disk).

Many different techniques and tools exist for file format identification. The one we chose to work with is easily scripted, relying on core Linux/UNIX tools. In order to generate a list of file format “types” it is necessary to identify the file formats we wish to work with. In Linux/UNIX, the default file format authority is the *file* command, which in turn relies on a file called *magic* (location of this file varies by installation), which in turn stores a myriad collection of file signatures. Various signatures exist therein, but the ones that in our opinion are most easily scripted are MIME based. The MIME types for JPEG, PNG and GIF are “image/jpeg,” “image/png” and “image/gif,” respectively. Of course, other commands can be used, such as ImageMagick’s *identify* tool, which is also readily scripted.

With the MIME types selected, entirely from the command line, we are able to obtain a full listing of all JPEG, PNG and GIF files located within a mounted evidence disk image. The command, found in **Command Line Listing 1**, was run from a terminal window.

```
$ find /media/evidence_disk -type f -print0 | xargs -0 file -i | grep -P
‘(image\/jpeg;|image\/png;|image\/gif;)’ | cut -d “:” -f 1 | sort >
/output_directory/file_listing.txt
```

Command Line Listing 1: Find and identify all JPEG, PNG, and GIF image files atop a mounted evidence disk

After running **Command Line Listing 1**, we obtain an output file containing a file listing with which we can use to do some real data processing in Mathematica.

/Basic forensic data processing using Mathematica

Before we can get any practical work done in Mathematica, we need to devise a way to read in a flat file (i.e. text file) that contains a listing of the images we want to process. We can readily do this without having to worry about testing for EOF, unlike many other programming languages. Mathematica supports typical file names that include many whitespace and ASCII characters. Of course, there are exceptions, and these may vary by supported platform and Mathematica version.

Using just a few simple lines, we can open a given text file for reading, determine how many images are to be processed, open the desired image file for reading and then do processing against it, one at a time, using a FOR loop. However, before this can be done, an input stream must be defined for reading in a text file. These are shown in **Code Listing 3**.

```
(* Clear everything *)
ClearSystemCache[]
ClearAll["Global`*"]

(* Define file to read in *)
inputfile = "input.txt";

(* Identify how many image files will be processed from input file*)
n = Length@Import[inputfile, "Lines"];
StringForm["Number of files to be processed from input file: ``\n", n]

(* Open input stream *)
```

```
inputstream = OpenRead[inputfile];

(* For loop to be used for processing *)
For [i = 0, i < n, i++;
  filename = Read[inputstream, String];
  Print[filename]]

(* Close input file stream *)
Close[inputstream];
```

Code Listing 3: Basic file input-based processing in Mathematica

`ClearSystemCache[]` and `ClearAll[]` are used to clear Mathematica's internal system of all previous results, from other programs or the current one (being rerun as it were) while the latter clears all the values, information and meta-information associated with all symbols. These functions are good practice to use in order to reduce Mathematica's memory footprint, and should always be placed at the top of a program.

Opening a file for reading is straightforward and closely resembles the approach taken in C. The name of the file is placed into a variable and read line by line from that file; assuming it is a text file, `OpenRead[]` is used against the opened stream. Determining the number of files to be read from the input text file is as simple as running `Length[]`against the importing of the text file itself, whose result is then stored in a variable. Notice that variable types do not need to be defined (e.g., int, float, char, etc.).

Many languages provide `printf()` like output formatting – in Mathematica, `StringForm[]` is just one of many options that can be used for formatting output. It lets us define additional output formatting including tabs "`\t`" and newlines "`\n`".

Just as `fgets()` is used in C to read a line from a file, we can use `Read[]` . It all depends on exactly what it is we want to do, but here, our goal is to read in one full line at a time. For unformatted output, `Print[]` helps keep to things simple.

As with most languages, loops are used to accomplish a repetitive series of tasks. For our first program, a simple FOR loop is best. Mathematica's version of the FOR loop and its functionality should be recognizable to anyone with basic programming experience.

`Close[]` functions similarly to C's `close()` function. It is good form to close all file streams at the end of a program, just as it is in most other languages.

In C-like languages where we use "`{ }`" to denote program structure and substructure, in Mathematica we use "`[ ]`". Thus, it is important to ensure that for each "`[`" open square bracket a "`]`" close square bracket is used elsewhere correspondingly, and the number of each used must be the same. Finally, we use "`;`" to denote that the current line of code's output is to be suppressed.

/Running multiple lines of code from a loop

Running multiple actions in serial, as done in most loops, is easy to implement. We can spice up what we have shown in **Code Listing 3**. Just after "`Print[filename]`" we append an additional action. This is shown in **Code Snippet 1**.

```
Print[filename]
Print[IntegerString[FileHash[filename, "SHA256"], 16]]]
```

Code Snippet 1: First actual data processing

Code Snippet 1 causes Mathematica to print out the read in filename on one line followed by the SHA256 hash of that file on the next. File hashing is done by `FileHash[]`. All output is, by default, sent to STDOUT, which is the running notebook. Other hashing schemes are currently supported and can be identified by consulting Mathematica's help library.

/Keeping track of time (CPU and overall time)

To determine how much actual CPU time a given unit of code takes, for example, to hash many thousands of files, change the FOR loop found in **Code Listing 3** to what is shown in **Code Snippet 2**.

```
Timing[For [i = 0, i < n, i++;
  filename = Read[inputstream, String];
  Print[filename]
  Print[IntegerString[FileHash[filename, "SHA256"], 16]]]]
```

Code Snippet 2: Actual CPU time used to perform work

When the program has finished running `Timing[]` will return the number of seconds the CPU spent on processing, regardless if the program is serial or parallelized.

In our benchmark against 90,808 files, representing 71,941,450,725 bytes of data, it took about 547 seconds of CPU time to process. Although Mathematica is not as fast as a dedicated file-hashing tool, but in all fairness, we were using a SHA256 hashing scheme which is considerably slower than SHA1.

CPU time, of course, is not the same as overall processing time. From the very moment we started the hashing program to the time it stopped, 2193 seconds went by. Overall processing time can be determined by inserting two `UnixTime[]` statements, one line above and one below the FOR loop, and storing the result from these two functions in two variables, as shown in **Code Snippet 3**.

```
starttime = UnixTime[];
Timing[For [i = 0, i < n, i++;
  filename = Read[inputstream, String];
  Print[filename]
  Print[IntegerString[FileHash[filename, "SHA256"], 16]]]]
stoptime = UnixTime[];
StringForm["Processing took `` seconds to complete\n", stoptime-starttime]
```

Code Snippet 3: Determining how long it actually takes to complete processing

By placing the result from each instance of `UnixTime[]` in a separate variable, we subtract the two values to arrive at how much time processing actually took. Where these variables and `UnixTime[]` statements are placed in a program should reflect where the vast majority of processing will take place, as we have done. Opening files for reading and then performing a single read, as shown in **Code Listing 3**, consumes very little processing time, and can thus be safely ignored.

/Handling errors, messages and warnings

Mathematica can be chatty about errors, messages and warnings in a running notebook, so it might be useful to suppress unnecessary output. This is done by placing an `Off[]` statement just above the offending code. For example, when processing filenames with unsupported characters in them, which Mathematica and many other tools and programs cannot handle, `FileHash[]` will trigger a `FileHash::noopen` error. This error is actually a general error (consult the documentation for more information). Thus, to suppress such errors, insert an `Off[]` statement just below the first instance of `UnixTime[]`, as shown in **Code Snippet 4**.

```
starttime = UnixTime[];
Off[General::noopen];
```

Code Snippet 4: Suppressing messages of the specified type from the running notebook

To re-enable the logging of such errors, replace `Off[]` with an `On[]` statement, as shown in **Code Snippet 5**.

```
starttime = UnixTime[];
On[General::noopen];
```

Code Snippet 5: Enabling messages of the specified type from the running notebook

However, when running real programs, getting bogged down in errors, messages and warnings, of which there are many types, it may be convenient to send such output to a text file for post-run processing and analysis. Of course, to do this we have to leave Mathematica's messaging subsystem on. Then, we have to create a new file stream and append Mathematica's `$Messages` subsystem to this file stream, as shown in **Code Snippet 6**.

```
(* Open and setup error message stream *)
errorfile = "errors.txt";
errorstream = OpenWrite[errorfile];
```

```
$Messages = Append[$Messages, errorstream];

...

Close[errorstream];
```

Code Snippet 6: Logging errors, messages, and warnings to dedicated log file

To write to a file stream, it must be opened – we do that with `OpenWrite[]`. But to force all messages to be sent are to the define error stream we must specify where `$Messages` output is to be redirected. Therefore, we have to append it to the error stream. These concepts are very similar to redefining where `stdout` and `stderr`, are sent to.

/Writing output to a file

The more files to be processed the longer the notebook's output will be, which will eventually slow down Mathematica's processing as it not only has to manage the resources attached to the notebook, its ever-growing list of output, on top of the underlying processing or analysis. For this reason, and for the fact that there will be an inordinate amount of output to scroll through, it is suggested when there are more than a few files to process not to print them out to the screen (`stdout`) but rather to an output file.

At the same time, writing output to some arbitrary text file permits us to store permanently the output in a readily accessible format for future analysis. The process of writing to an output file is straightforward, as shown in **Code Snippet 7**.

```
outputfile = "output.txt";
outputstream = OpenWrite[outputfile];
starttime = UnixTime[];
Timing[For [i = 0, i < n, i++;
  filename = Read[inputstream, String];
  WriteString[outputstream, filename, "\t",
IntegerString[FileHash[filename, "SHA256"],16], "\n"]]]
stoptime = UnixTime[];

...

Close[outputstream];
```

Code Snippet 7: Defining and writing some output to a text file

Writing actual data to the file can be done using various functions, but a straightforward one to use is `WriteString[]`, as it permits additional control over the formatting of the output. We can readily add tabs "`\t`" and newlines "`\n`". Other functions for writing to files may make reformatting the output more difficult.

/Printing file output

Sometimes it is useful when working with new code to start with small input files, and from there, verify the generated output, assuming output was redirected to a file. To list the contents of an input file, based on the code we have already looked at, it suffices to place a `FilePrint[]` statement at an appropriate location in the code. An example is shown in **Code Snippet 8**.

```
(* Open input stream *)
inputstream = OpenRead[inputfile];
FilePrint[inputfile]
```

Code Snippet 8: Listing input file contents

From **Code Snippet 8**, we obtain a file listing of 5 files, a short list of examples we chose to initially work with while developing our code, as shown in **Output Listing 2**.

```
Out[5] = /usr/local/Wolfram/Mathematica/10.4/Documentation/English/System/
       ExampleData/spikey2.png
       /usr/local/Wolfram/Mathematica/10.4/Documentation/English/System/
       ExampleData/turtle.jpg
       /usr/local/Wolfram/Mathematica/10.4/Documentation/English/System/
       ExampleData/photo.jpg
       /usr/local/Wolfram/Mathematica/10.4/Documentation/English/System/
       ExampleData/coneflower.jpg
       /usr/local/Wolfram/Mathematica/10.4/Documentation/English/System/
       ExampleData/ocelot.jpg
```

Output Listing 2: Contents of file "input.txt"

From **Output Listing 2**, we readily identify the names and location of the various test files we used.

/Our first complete basic Mathematica forensic data processing program

After everything we have looked at, we are ready to present a fully functioning data processing program that can be readily modified for more advanced processing and analysis. It brings together all the topics we have shown and it performs basic forensic data processing (i.e., file hashing). This is shown in **Code Listing 4**.

```
(* Clear everything *)
ClearSystemCache[]
ClearAll["Global`*"]

(* Define files *)
inputfile = "input.txt";
outputfile = "output.txt";
errorfile = "errors.txt";

(* Identify how many image files will be processed from input file *)
n = Length@Import[inputfile, "Lines"];
StringForm["Number of files to be processed from input file: ``\n", n]

(* Define input, output and error streams *)
inputstream = OpenRead[inputfile];
outputstream = OpenWrite[outputfile];
errorstream = OpenWrite[errorfile];
$Messages = Append[$Messages, errorstream];

(* For loop to be used for processing *)
starttime = UnixTime[];
Timing[For [i = 0, i < n, i++;
  filename = Read[inputstream, String];
  WriteString[outputstream, filename, "\t",
IntegerString[FileHash[filename, "SHA256"],16], "\n"]]]
stoptime = UnixTime[];
StringForm["Processing took `` seconds to complete\n", stoptime-starttime]

(* List contents of output file before closing output stream *)
FilePrint[outputfile]

(* Close file streams *)
Close[inputstream];
Close[outputstream];
Close[errorstream];

(* Reset $Messages to avoid unnecessary frustrations *)
(* Quick and Dirty Method *)
$Messages = $Messages[[{1}]];
```

Code Listing 4: Complete program

Looking at **Code Listing 4**, program functionality should be clear. Anyone with even only minor programming experience can readily modify it.

If `$Messages` is not reset back to its original state it will cause unnecessary errors. This is because we have changed `stderr`; we need to rebalance the equation. Resetting is as easy as redefining `$Messages`, as shown in the very last line of **Code Listing 4**.

When the program has finished its processing, a new text file named "output.txt" should have been created containing a list of each filename read in from the input file, a tab and its SHA256 hash. Any errors or messages will be logged to "errors.txt."

After running the program, you should see output similar to **Output Listing 3**, which contains both the name of the file read in and processed along with its corresponding SHA256 hash.

```
Out[18]= /usr/local/Wolfram/Mathematica/10.4/Documentation/English/System/
       ExampleData/spikey2.png       bb60ce413161a5c950ebf8ab476c1cbac04
       00408e90da5bfced735c523279542
       /usr/local/Wolfram/Mathematica/10.4/Documentation/English/System/
       ExampleData/turtle.jpg       6d787b5e12b1e6defb87529b848227404cc451
       251bb55dc8cd9bd0a965f01e6e
       /usr/local/Wolfram/Mathematica/10.4/Documentation/English/System/
       ExampleData/photo.jpg       d06e6263f2124172bd613955c5f78b985c5b7c6
       545491048d349151dbe315840
       /usr/local/Wolfram/Mathematica/10.4/Documentation/English/System/
       ExampleData/coneflower.jpg       c13b7cd74cda57a304a27f9d707dba3d2c
       92075168c30216de36c3280861e80b
       /usr/local/Wolfram/Mathematica/10.4/Documentation/English/System/
       ExampleData/ocelot.jpg       dd83a1f7c1f77e251fdc7b25f01e8664b2c8d4
       e8fc46a3cfdffd62813da8e3ea
```

Output Listing 3: Output from running complete program

/Wrap-up and Conclusion

We have seen quite a few Mathematica functions, which in general, work exactly as their names imply. Using only a few functions, we can build complete programs capable of performing rudimentary forensic data processing. As we introduce additional concepts and capabilities in our follow-up articles, we will build complex data processing programs in only a few lines of code that can be readily used in computer forensic investigations.

Mathematica is unlike anything in the software toolbox employed by investigators and analysts. It has capabilities which make it unparalleled compared to most general data processing and analysis suites. However, the major issue hindering its adoption is that it requires looking at problems differently. Although it already has much functionality, the trick is tuning its capabilities to forensic investigations. That is why we chose to work with image analysis. It is something most investigators and analysts can readily visualize and relate to.

While what we have seen thus far may seem elementary to some, keep in mind that until this very article, there were no readily available tutorials, articles or papers that detailed how to implement batch-based data processing into Mathematica, although this capability has long been there. We will use this information to build the foundation of our data processing capability, which will be expanded upon in our next article, where we will be exploring forensic image-based analysis.

/Acknowledgments

We sincerely express our gratitude to Wolfram Premier Service technical support, specifically Danny Finn and Lin Guo.

E17-0424-0955 - DOCUMENT.docx

/FEATURE IN BRIEF

- /Subject Matter
  Image analysis; symbolic and mathematical programming (procedural and functional)

- /Skill Level
  4-5

- /Requirements
  - Very powerful workstation or desktop computer (i7 (6-core / 12 with hyper-threading) or Xeon (8-core / 16 with hyper-threading) or greater);
  - Lots of memory (64 GiB or greater);
  - Lots of disk space for storing the very large notebook files that Mathematica can generate;
  - Fast disk(s) for reading and writing Mathematica's notebooks;
  - Very stable 64-bit computer operating system, preferably one that support 64+ GiB RAM;
  - Hardware RAID storage if there will be lots of data to be stored and processed;
  - Familiarity with Mathematica or other symbolic computer assisted mathematic such as, but not limited to, Maple, Matlab, GNU Octave, Scilab, R, etc.;
  - Familiarity with batch processing;
  - Knowledge of procedural or functional programming paradigms;
  - A basic knowledge of "big data," how it can be processed and analysed will be helpful.

- /Additional Reading
  Mathematica's documentation is very extensive and up to date. There are many books that have been written on programming with Mathematica as well as interacting and building applications from it. See http://www.wolfram.com/books/?source=nav for a full listing of known books on a variety of Mathematica related subjects, including scientific programming and analysis. A variety of books for users at all levels can be found listed on Wolfram's web site.

  As for digital forensics with Mathematica, there are no known publicly available books or treatises on the subject.

/BOXOUTS

- /Boxout 1 – Writing memory constrained programs
  When working with 32-bit operating systems, the amount of available memory will be more limited than with 64-bit systems. In such cases, it may be necessary for programmers to test for their presence. This can be tested for using Mathematica's `$System` and `$OperatingSystem` functions incorporated into `If[]` comparisons. Additionally, hard memory limits can be set using `MemoryConstrained[]` or `ByteCount[]` for a given calculation or function. Finally, `ClearSystemCache[]` and `ClearAll[]` should be used to clear out Mathematica's current in use memory from previous calculations and metadata.

- /Boxout 2 – Data exporting
  Just as Mathematica supports a wide assortment of file formats for data input, it supports a large number of formats for exporting data to. Use `$ExportFormats` to get a current listing of available export file formats.

- /Boxout 3 – Code reuse

  Keeping track of one's progression in writing programs and developing code will help promote code reuse. This will provide relief when writing similarly themed programs or code. Where possible, reuse data processing methods (including efficient looping system), procedures and templates. However, Mathematica does not directly provide OOP facilities, so do not expect this type of code reuse.

- /Boxout 4 (Type 3) – Running as a script
  Mathematica supports scripting. Scripting is straightforward – all it requires is a sequential series of commands or calculations that you want performed. In short, it is very much like a program, minus any direct interaction with the program. It is started the same way standard shell scripts are, via the command line. The documentation provides sufficient information to get you started.

/SIDEBARS

- /Sidebar 1 – Handling filenames
  When dealing with filenames, Mathematica does not handle filenames that contain non-ASCII characters. It handles most filenames that contain whitespace and punctuation, but even some ASCII characters may require filename escaping (e.g., "\@"). In any event, if Mathematica cannot handle the filename it will generate an error.

- /Sidebar 2 – Where possible, use parallelization
  Some Mathematica functions are highly parallelizable, e.g., factoring prime numbers. While the functions we use do not take immediate advantage of parallelization, modifications to the processing loop may be parallelizable. As you get better at using Mathematica, you will want to find ways to improve performance. The easiest way to parallelize is to use the `Parallelize[]`. In our tests, we determined that running `Timing[Table[Prime[n], {n, 99999999}];]` took 219 seconds while `Timing[Parallelize[Timing[Table[Prime[n], {n, 99999999}]]];]` took 38 seconds, spreading the load across 8 CPU cores.

- /Sidebar 3 – For the daring
  For those interested in going beyond what we have looked at, Mathematica provides lots of functionality that permits the building of interactive programs. Such programs may be suitable in instances where the names of input and output files are not known ahead of time or to prevent the user from modifying program code. This can allow the user to concentrate on running a program, not adjusting it. A good starting place for more information is the built-in documentation.

/PULL-QUOTES

- /Pull-quote 1

  For digital forensic analyses, Mathematica does not require any special add-ons or third-party products to provide the necessary features and capabilities investigators and analysts will likely need.

- / Pull-quote 2

  Mathematica's learning curve can be steep so prior experience with similar software, image processing, data analysis or "big data" will definitely be helpful.

- / Pull-quote 3

  The majority of questions that beginning investigators and analysts will have can usually be answered by looking at Mathematica's extensive documentation library, available both in the product itself and online

/REFERENCES

[1] Microsoft MSDN. Memory Limits for Windows and Windows Server Releases. Technical article. Microsoft MSDN. 2016. https://msdn.microsoft.com/en-ca/library/windows/desktop/aa366778(v=vs.85).aspx#.

E17-0424-0955 - DOCUMENT.docx

/AUTHOR BIOS

R. Carbone has been working for Defence R&D Canada since 2001. He has been working as a digital forensics investigator and researcher for the past seven years and has published numerous articles and case studies in this field. He is also an open source software expert and was the co-author of a Government of Canada study that proved highly influential to federal government policy regarding the adoption of open source software. As a SANS Gold Adviser, he provides guidance to SANS students seeking recertification or working on their Masters' whitepaper requirements. Finally, he is a certified digital forensic investigator, incident handler and malware reverse engineer.
Defence R&D Canada
val-forensics@drdc-rddc.gc.ca
Head Shot – See image DRDC-Graphicbar-3inch-300dpi.jpg