Math3101/Math5303
Computational Mathematics

A/Prof William McLean,
School of Maths and Stats, UNSW
Session 1, 2017

Version: April 4, 2017

Outline

Writing to a text file

The simplest way is usually to redirect standard output:

```
$ julia table_stdout.jl > table.txt
```

Here, the source file `table_stdout.jl` contains code to write a table of values of $J_0(x)$ to standard output.

```
1  const N = 10
2  x = linspace(0, 1, N+1)
3  @printf("%5s %18s\n", "# x", "J(x)")
4  for j = 1:N+1
5      @printf("%5.2f %18.14f\n", x[j], besselj(0,x[j]))
6  end
```

The hash character # signifies a header line, so another script that reads the file knows to process the first line differently from the ones that follow.

The `@printf` macro takes a C-style format string as its first argument, and is convenient for printing tables. The format placeholder `%5.2f` means that the corresponding floating-point parameter is printed in a field of width 5 with 2 digits after the decimal point.

Instead of calling the `@printf` macro, we could call the `println` function.

```
1 for j = 1:N+1
2     println(x[j], ' ', besselj(0,x[j]))
3 end
```

However, the numbers will not be aligned properly.

Note that `println(...)` is the same as `print(...,'\n')`.

The `writedlm` function provides a simple way to write a matrix to a file.

```
const N = 10
x = linspace(0, 1, N+1)
writedlm("table.txt", [x besselj(0,x)], ' ')
```

The `dlm` in the name refers to delimiter, which is the character specified in the third argument; in this example, a space `' '`.

Changing `' '` to a comma `','` would produce a csv file.

There is a matching function to read the numbers in a file and return the corresponding matrix:

```
table = readdlm("table.txt", Float64)
```

More flexible data handling

Writing and reading floating point data as text is fine for a small table of numbers, but becomes inefficient for large arrays because of the type conversions involved. In addition, conversion between binary and decimal representations generally results in rounding errors.

An industrial strength alternative is provided by the Hierarchical Data Format (HDF5) standard, accessed in Julia via the JLD package.

```
1  using JLD
2
3  a = rand(1000)
4  b = rand(100,100)
5  c = sprand(1000,1000, 0.01)
6
7  save("mydata.jld", "a", a, "b", b, "c", c)
```

After saving some variables in a script, you can load them again in another script.

```
using JLD

myvars = load("mydata.jld")
a = myvars["a"]
b = myvars["b"]
c = myvars["c"]
```

The load function returns a dictionary whose keys are the labels written by the save function.

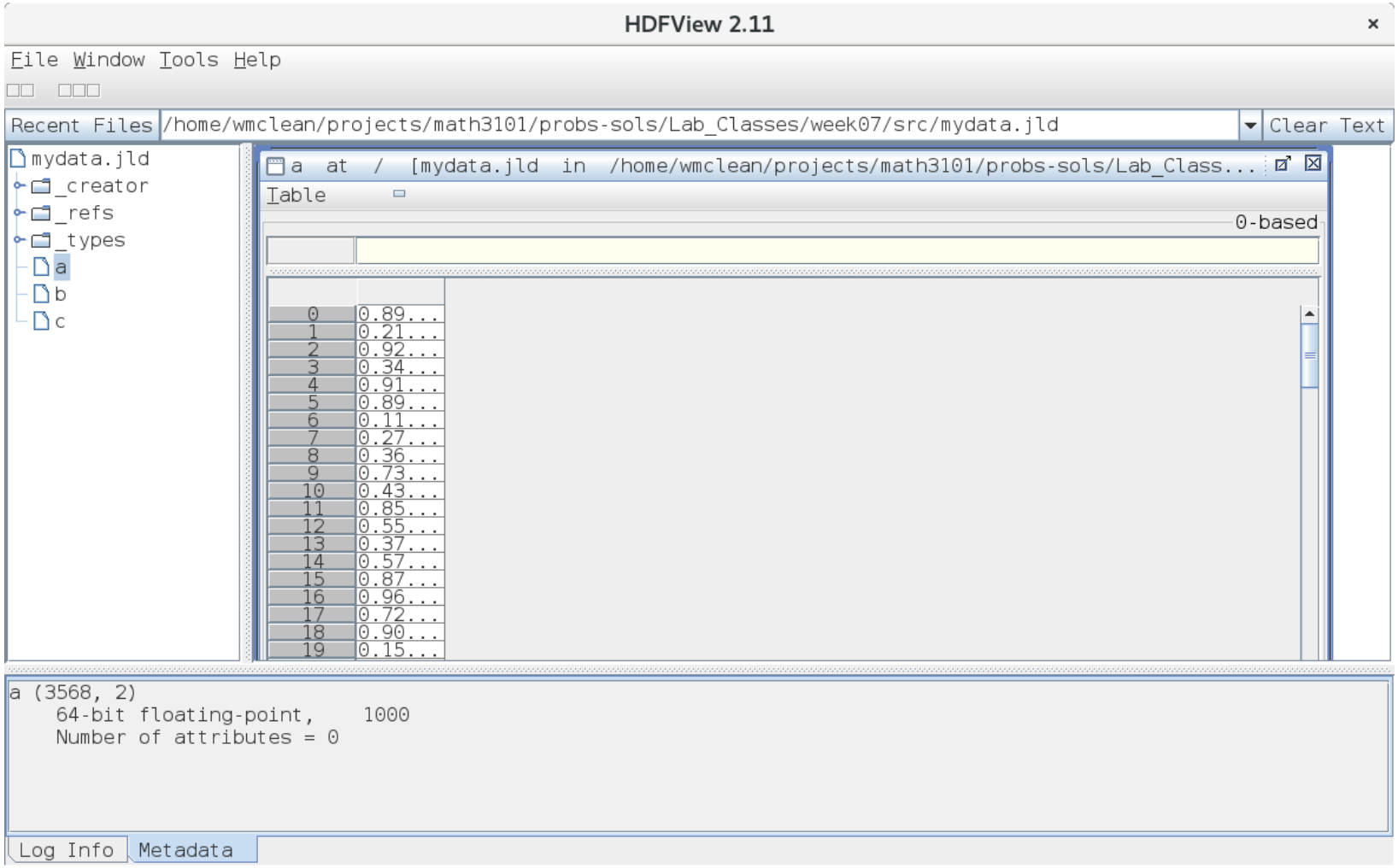
By doing

```
$ file mydata.jld
```

you can verify the file type:

```
mydata.jld: Hierarchical Data Format (version 5) ...
```

Browse a .jld file



Line-by-line file I/O

The following function opens a given file, writes each line to standard output, and then closes the file.

```
1  function showlines(filename::String)
2      infile = open(filename, "r")
3      for line in eachline(infile)
4          write(STDOUT, line)
5      end
6      close(infile)
7      return nothing
8  end
```

Here, `infile` has the type `IOStream`, and the `"r"` argument causes Julia to open the file for reading.

The open function has an alternative method that takes 3 arguments, the first of which is a function. The statement

```
1 open(func, "somefile.txt", "r")
```

is equivalent to

```
1 infile = open("somefile.txt", "r")
2 func(infile)
3 close(infile)
```

This version has the advantage that you do not have to remember to close the file, but having to define a function can be inconvenient if it will be used only once.

A begin-end block allows you to define a multiline anonymous function on the fly. For example,

```
1  open(infile -> begin
2                     for line in eachline(infile)
3                         println(line)
4                     end
5                 end,
6       "load_vars.jl", "r")
```

will print the contents of `load_vars.jl` but with double spacing due to the newline character added by `println` to each line.

Actually, the recommended technique is to use a do-block.

In general,

```
foo(a, b) do x, y
    statements
end
```

is equivalent to defining a function

```
function bar(x, y)
    statements
end
```

and calling

```
foo(bar, a, b)
```

So the example above is most naturally coded as

```
open("load_vars.jl", "r") do infile
    for line in eachline(infile)
        println(line)
    end
end
```

Write to a text file

In our first example, we wrote a table to a file by redirecting standard output. Alternatively, we could write directly to a named file as follows.

```
1  const N = 10
2  open("table.txt", "w") do outfile
3      x = linspace(0, 1, N+1)
4      @printf(outfile, "%5s %18s\n", "# x", "J(x)")
5      for j = 1:N+1
6          @printf(outfile, "%5.2f %18.14f\n", x[j],
               besselj(0,x[j]))
7      end
8  end
```

The "w" argument to the open function tells Julia to open the file for writing; if the file already exists then it will get overwritten.

Exercises

1. Modify your script `pascal.jl` so that it writes to a file called `Pascals_triangle.txt`.
2. Write a Julia function

   ```
   n = grep(pattern, filename)
   ```

   that writes each line in `filename` that matches the given `pattern`, and returns the number `n` of such lines. Hint: use the `Regex` and `ismatch` functions.
3. Using C's "%a" or "%A" format type you can write the exact hexadecimal representation of a floating point number: e.g.,

   ```
   s = @sprintf("%A", sqrt(2))
   ```

   gives `s = "0X1.6A09E667F3BCDP+0"`. You can recover the number by doing `x = parse(Float64, s)`. Write a script that writes a table to a file using the "%A" format, and then reads the numbers back, checking that they are identical to all 64 bits. What happens if you instead use "%20.15e"?