Language-agnostic data analysis workflows and reproducible research

Andrew John Lowe

Wigner Research Centre for Physics,

Hungarian Academy of Sciences

28 April 2017

Overview

- This talk: language-agnostic (or polyglot) analysis workflows
- I'll show how it is possible to work in multiple languages and switch between them without leaving the workflow you started
- Additionally, I'll demonstrate how an entire workflow can be encapsulated in a markdown file that is rendered to a publishable paper with cross-references and a bibliography (and with raw a LaTeX file produced as a by-product) in a simple process, making the whole analysis workflow reproducible

Which tool/language is best?

- TMVA, scikit-learn, h2o, caret, mlr, WEKA, Shogun, ...
- C++, Python, R, Java, MATLAB/Octave, Julia, ...
- Many languages, environments and tools
- People may have strong opinions
- A false polychotomy?
- Be a polyglot!

Approaches

- Save-and-load
- Language-specific bindings
- Notebook
- knitr

Save-and-load approach

Flat files

- Language agnostic
- Formats like text, CSV, and JSON are well-supported
- Breaks workflow
- Data types may not be preserved (e.g., datetime, NULL)
- New binary format Feather solves this

Feather

Feather: A fast on-disk format for data frames for R and Python, powered by Apache Arrow, developed by Wes McKinney and Hadley Wickham

```r
# In R:
library(feather)
path <- "my_data.feather"
write_feather(df, path)
```

```python
# In Python:
import feather
path = 'my_data.feather'
df = feather.read_dataframe(path)
```

Other languages, such as Julia or Scala (for Spark users), can read and write Feather files without knowledge of details of Python or R

RootTreeToR

- RootTreeToR allows users to import ROOT data directly into R
- Written by Adam Lyon (Fermilab), presented at useR! 2007
- cdcvs.fnal.gov/redmine/projects/roottreetor
- Requires ROOT to be installed, but no need to run ROOT

```
# Open and load ROOT tree:
rt <- openRootChain("TreeName", "FileName")
N <- nEntries(rt) # number of rows of data
# Names of branches:
branches <- RootTreeToR::getNames(rt)
# Read in a subset of branches (varsList), M rows:
df <- toR(rt, varsList, nEntries=M)
# Use writeDFToRoot to write a data.frame to ROOT
```

root_numpy

root_numpy is a Python extension module that provides an efficient interface between ROOT and NumPy

```
import ROOT
from root_numpy import root2array, root2rec, tree2rec
from root_numpy.testdata import get_filepath
filename = get_filepath('test.root')
# Convert a TTree in a ROOT file into a NumPy
# structured array:
arr = root2array(filename, 'tree')
# Convert a TTree in a ROOT file into a NumPy
# record array:
rec = root2rec(filename, 'tree')
# Get the TTree from the ROOT file:
rfile = ROOT.TFile(filename)
intree = rfile.Get('tree')
# Convert the TTree into a NumPy record array:
rec = tree2rec(intree)
```

Language-specific approaches

rPython (using Python from R)

```
library(rPython)
python.call( "len", 1:3 )
a <- 1:4
b <- 5:8
python.exec( "def concat(a,b): return a+b" )
python.call( "concat", a, b)
python.assign( "a", "hola hola" )
python.method.call( "a", "split", " " )
```

rpy2 (using R from Python)

```
from rpy2.robjects.lib.dplyr import (filter,
                                     mutate,
                                     group_by,
                                     summarize)
dataf = (DataFrame(mtcars) >>
         filter('gear>3') >>
         mutate(powertoweight='hp*36/wt') >>
         group_by('gear') >>
         summarize(mean_ptw='mean(powertoweight)'))
dataf
```

Others

- Calling Python from Julia: PyCall
- Calling R from Julia: RCall
- Calling Python from Excel: xlwings
- Calling Python from Ruby: Rupy
- ...

Notebook approach

Jupyter Notebook

- Each notebook has one main language
- Cells can contain other languages through “magic”
- For example: ggplot2 in Jupyter Notebook
- ROOTbooks: Notebooks running ROOT Jupyter kernel
- Assume that most people here have already seen these

Beaker Notebook

- Each notebook can contain any language
- Many languages are supported
- Auto translation of data (copied)
- `http://beakernotebook.com/`

Figure 1: "A universal translator for data scientists"

Reproducible research

Typical research pipeline

1. Measured data $\xrightarrow{\text{processing code}}$ analysis data
2. Analysis data $\xrightarrow{\text{analysis code}}$ computational results
3. Computational results $\xrightarrow{\text{presentation code}}$ plots, tables, numbers
4. Plots, tables, numbers + text $\xrightarrow{\text{manual editing}}$ **ARTICLE**
5. (Maybe) put the data and code on the web somewhere

Challenges with this approach

- Onus is on researchers to make their data and code available
- Authors may need to undertake considerable effort to put data and code on the web
- Readers must download data and code individually and piece together which data go with which code sections, etc.
- Typically, authors just put stuff on the web
- Readers just download the data and try to figure it out, piece together the software and run it
- Authors/readers must manually interact with websites
- There is no single document to integrate data analysis with textual representations; *i.e.* data, code, and text are not linked
- *Your experiment may impose restrictions on what kind of data can be shared publicly. Nevertheless, making an analysis reproducible benefits both your collaboration colleagues and your future self!*

Literate Statistical Programming

- Original idea comes from Donald Knuth
- An article is a stream of **text** and **code**
- Analysis code is divided into text and code "chunks"
- Presentation code formats results (tables, figures, *etc.*)
- Article text explains what is going on
- Literate programs are *weaved* to produce human-readable documents and *tangled* to produce machine-readable documents

knitr

- **knitr** is a system for literate (statistical) programming
- Uses R as the programming language, but others are allowed
- Benefits: text and code are all in one place, results automatically updated to reflect external changes, *workflow supports reproducibility from the outset*
  - Almost no extra effort required by the researcher after the article has been written
- Like Notebooks, but output is publication quality, and you get the LaTeX also as a freebie

R Markdown

This is an R Markdown presentation. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see `http://rmarkdown.rstudio.com`

It looks like LaTeX, or to be more precise, it's a **Beamer** presentation, but this slide was created like:

```
## R Markdown

This is an R Markdown presentation. Markdown is a simple
formatting syntax for authoring HTML, PDF, and MS Word
documents. For more details on using R Markdown see
<http://rmarkdown.rstudio.com>

It looks like \LaTeX, or to be more precise, it's a
**Beamer** presentation, but this slide was created like:
```

R Markdown *(continued)*

The Markdown syntax has some enhancements. For example, you can include LaTeX equations, like this:

$$i\hbar\gamma^\mu\partial_\mu\psi - mc\psi = 0 \tag{1}$$

We can also add tables and figures, just as we would do in LaTeX:

Table 1: Effectiveness of Insect Sprays. The mean counts of insects in agricultural experimental units treated with different insecticides.

Count	**Spray**
A	14.500000
B	15.333333
C	2.083333
D	4.916667
E	3.500000
F	16.666667

Example code chunk

A code chunk using R is defined like this:

````
```{r}
print('hello world!')
```
````

A code chunk using some other execution engine is defined like this:

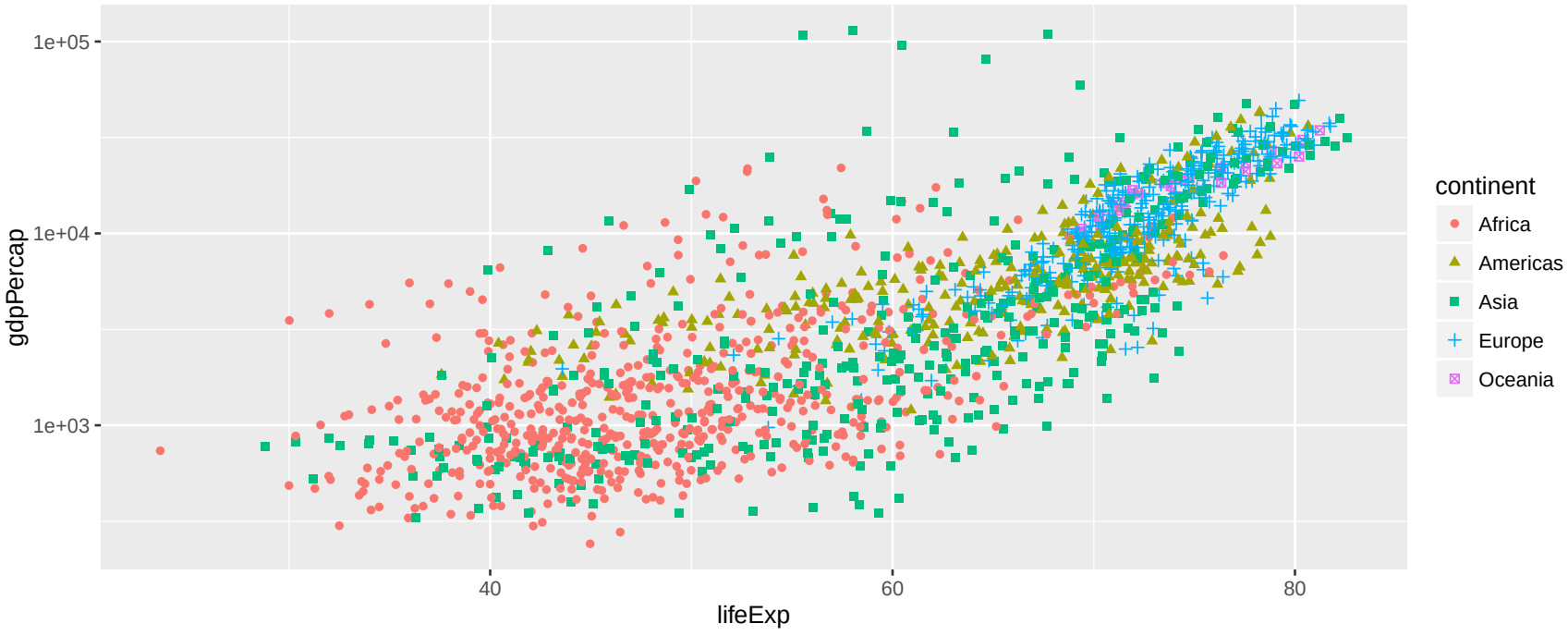
````
```{<execution-engine>}
print('hello world!')
```
````

For example:

````
```{python}
x = 'hello, python world!'
print(x.split(' '))
```
````

Example R code chunk

```
ggplot(
  data = gapminder, aes(x = lifeExp, y = gdpPercap)) +
  geom_point(
    aes(color = continent, shape = continent)) +
  scale_y_log10()
```



Code execution engines

- Although R is the default code execution engine and a first-class citizen in the knitr system, many other code execution engines are available; this is the current list: *awk, bash, coffee, gawk, groovy, haskell, lein, mysql, node, octave, perl, psql, python, Rscript, ruby, sas, scala, sed, sh, stata, zsh, highlight, Rcpp, tikz, dot, c, fortran, fortran95, asy, cat, asis, stan, block, block2, js, css, sql, go*
- I've already shown examples of R and Python code chunks earlier in this talk
- Except for R, all chunks are executed in separate sessions, so the variables cannot be directly shared. If we want to make use of objects created in previous chunks, we usually have to write them to files (as side effects). For the bash engine, we can use Sys.setenv() to export variables from R to bash.
- Code chunks can be read from external files, in the event that you don't want to put everything in a monster-length markdown document (use **bookdown**)

FORTRAN

Define a subroutine:

```
C Fortran test
      subroutine fexp(n, x)
      double precision x
C  output
      integer n, i
C  input value
      do 10 i=1,n
         x=dexp(dcos(dsin(dble(float(i)))))
  10  continue
      return
      end
```

FORTRAN *(continued)*

Call it from R:

```
# Call the function from R:
res = .Fortran("fexp", n=100000L, x=0)
str(res)
## List of 2
##  $ n: int 100000
##  $ x: num 2.72
```

Haskell

```
[x | x <- [1..10], odd x]
```

```
## [1,3,5,7,9]
```

C

C code is compiled automatically:

```
void square(double *x) {
  *x = *x * *x;
}
```

```
## gcc -std=gnu99 -I/usr/share/R/include -DNDEBUG       -fpi
## gcc -std=gnu99 -shared -L/usr/lib/R/lib -Wl,-Bsymbolic-f
```

Test the `square()` function:

```
.C('square', 9)
```

```
## [[1]]
## [1] 81
```

Rcpp

The C++ code is compiled through the **Rcpp** package; here we show how we can write a function in C+ that can be called from R:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector timesTwo(NumericVector x) {
  return x * 2;
}
```

```
# In R:
print(timesTwo(3.1415926))
```

```
## [1] 6.283185
```

cat

A special engine cat can be used to save the content of a code chunk to a file using the `cat()` function, defined like this:

````
```{cat engine.opts=list(file='source.cxx')}
// Some code here...
```
````

BASH

```
echo hello bash!
echo 'a b c' | sed 's/ /\|/g'
# We can also compile the source.cxx file that we
# created in the previous code chunk; maybe compile
# against ROOT libraries, etc.
# Then run the application
```

```
## hello bash!
## a|b|c
```

We can run anything that we can run from the command line, including stuff we built in previous code chunks

Data exchange

Since the Python engine executes code in an external process, exchanging data between R chunks and Python chunks is done via the file system. If you are exchanging data frames, you can use the **Feather** package for very high performance transfer of even large data frames between Python and R:

```
import pandas
import feather
# Read flights data and select flights to O'Hare
flights = pandas.read_csv("flights.csv")
flights = flights[flights['dest'] == "ORD"]
# Select carrier and delay columns and drop rows with
# missing values
flights = flights[['carrier','dep_delay','arr_delay']]
flights = flights.dropna()
print flights.head(10)
# Write to feather file for reading from R
feather.write_dataframe(flights, "flights.feather")
```

Data exchange *(continued)*

```
library(feather)
library(ggplot2)

# Read from feather and plot
flights <- read_feather("flights.feather")
ggplot(flights, aes(carrier, arr_delay)) +
  geom_point() +
  geom_jitter()
```

Knitron

- The **knitron** package, available on GitHub but not yet on CRAN, is intended to allow users the ability to use IPython/Jupyter and `matplotlib` in R Markdown code chunks and render them with `knitr`
- According to the author's website, "Knitron works by lazily starting a global IPython kernel the first time a code chunk gets rendered by knitr and this kernel is reused for all consecutive chunks. This way all the computation done in any previous chunk is available in the current chunk, providing R-like behaviour for Python".

Knitron example

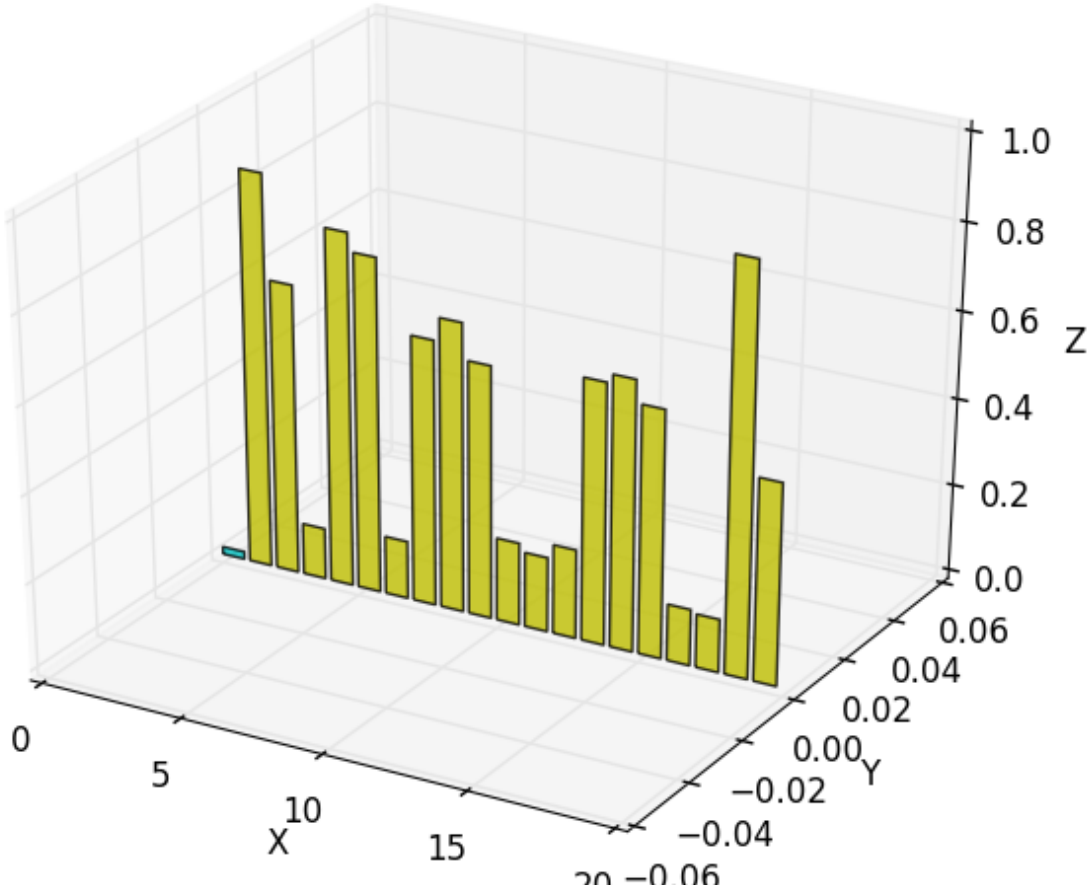
```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
for c, z in zip(['r', 'g', 'b', 'y'], [30, 20, 10, 0]):
  xs = np.arange(20)
ys = np.random.rand(20)
cs = [c] * len(xs)
cs[0] = 'c'
ax.bar(xs, ys, zs=z, zdir='y', color=cs, alpha=0.8)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
plt.savefig("myplot.png")
```

```
## GLib-GIO-Message: Using the 'memory' GSettings backend.
```

Knitron example *(continued)*

Then plot the figure using a standard knitr mechanism, for example:

```
cat("![My matplotlib plot.](myplot.png)")
```



ROOT

- In the R tutorial that I gave at the IML Workshop last month, I spoke about the possibility of using code execution engines other than R, but noted then that there is no way to use ROOT as a code execution engine
- Well, now there is!
- I have written an interface to allow ROOT to be used as a code execution engine
  - Simple proof-of-concept at the moment; nothing too sophisticated at this stage
  - Runs `root -q -b macro.C` and runs the macro
  - If not sufficient, it's always possible to run more complicated analyses, perhaps using stand-alone C++ code using ROOT libraries, or using your experiment's software framework, in a UNIX shell code chunk

ROOT code execution engine example

```
// Builds a graph with errors, displays it and saves it
// as image. First, include some header files (within,
// CINT these will be ignored).

#include "TCanvas.h"
#include "TROOT.h"
#include "TGraphErrors.h"
#include "TF1.h"
#include "TLegend.h"
#include "TArrow.h"
#include "TLatex.h"

void macro1(){
    // The values and the errors on the Y axis
   const int n_points=10;
   double x_vals[n_points]=
           {1,2,3,4,5,6,7,8,9,10};
   double y_vals[n_points]=
```

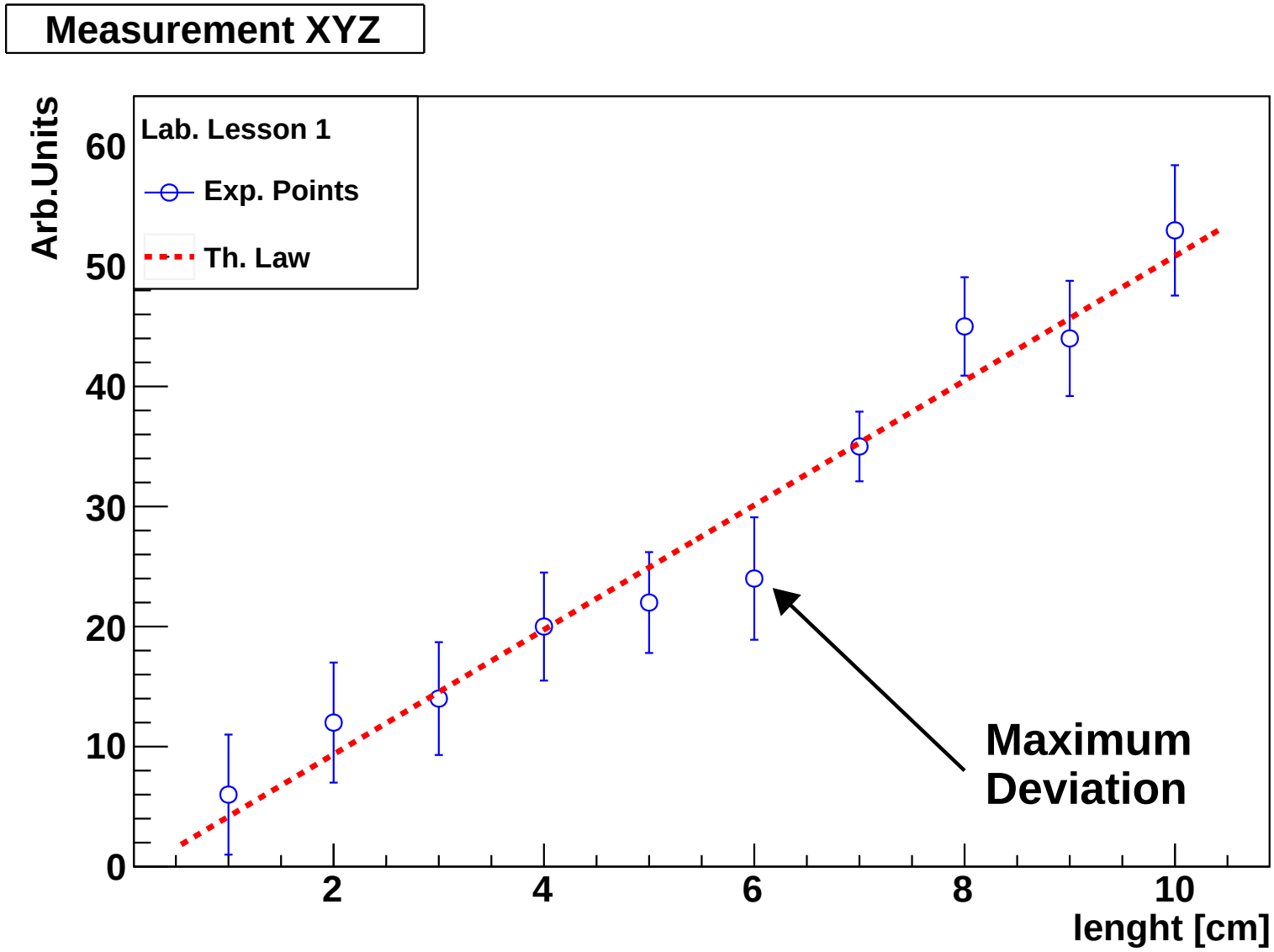


Figure 3: This is the graph generated by the ROOT macro.

Example paper output

- See toy_example_paper.pdf to see example output from **knitr**
- This Beamer presentation, but rendered as a paper instead
- To demonstrate that the necessary infrastructure works
- Contains (hyperlinked) cross-references and bibliography
- Note that I wouldn't have to do anything special to make this happen for a real paper; I ran a single command to run all the "analysis code" and generate the document with all the plots, tables, numerical results, *etc.*
- Raw LaTeX (to submit to journal) generated as a by-product

Messages to take away

- It is already possible to write a reproducible analysis in your favourite programming language and have your paper rendered fit for publication
- You can mix and match programming languages in a single uninterrupted workflow
- There are ways to exchange data between code chunks that may be written in different programming languages
  - ROOT, Python, R, ...
- Works nicely with version control (it's just plain text!)
- You can now embed ROOT code in a reproducible analysis

Acknowledgements

This work is supported by NKFIH OTKA grant K120660.