Scala as a platform for statistical computing and data science

Darren Wilkinson
http://tinyurl.com/darrenjw
School of Mathematics & Statistics
Newcastle University, UK

Statistical computing languages
RSS, London, 21st November 2014

Talk outline

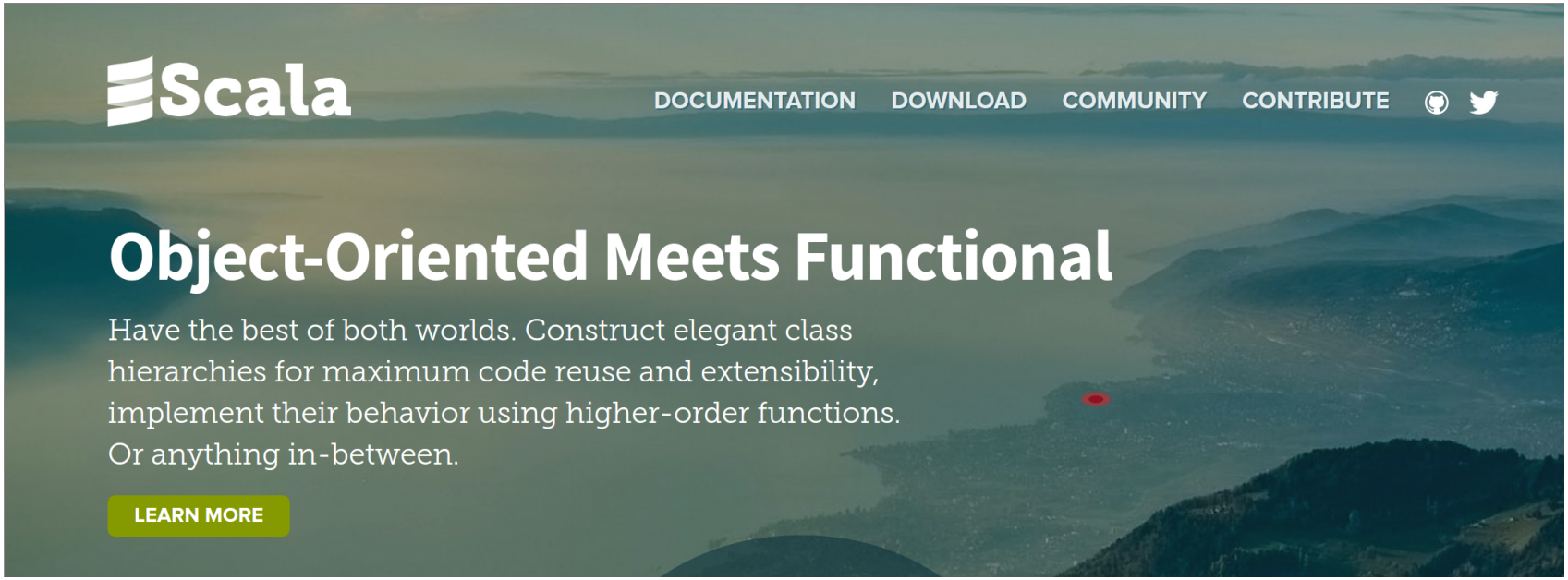
- Requirements for a platform for statistical computing
- Background: Scala, Java and the JVM
- Functional programming, immutable data structures and parallelisation
- Quick introduction to the Breeze scientific library
- Examples: Monte Carlo and Linear regression
- Wrap-up

Feature wish list

It should:

- be a general purpose language with a sizeable user community and an array of general purpose libraries, including good GUI libraries, networking and web frameworks
- be free, open-source and platform independent, fast and efficient with a strong type system, and be statically typed with good compile-time type checking and type safety
- have a good, well-designed library for scientific computing, including non-uniform random number generation and linear algebra
- have reasonable type inference and a REPL for interactive use
- have good tool support (including build tools, doc tools, testing tools, and an intelligent IDE)
- have excellent support for functional programming, including support for immutability and immutable data structures and monadic design
- allow imperative programming for those (rare) occasions where it makes sense
- be designed with concurrency and parallelism in mind, having excellent language and library support for building really scalable concurrent and parallel applications

Scala



http://www.scala-lang.org/

History and background

- The name Scala derives from "Scalable Language"
- It is a hybrid object-oriented/functional language
- It supports both functional and imperative styles of programming, but functional style is idiomatic
- It is statically typed and compiled — compiling to Java bytecode to run on the JVM
- It was originally developed by Martin Odersky, one of the original authors of the Java compiler, `javac` as well as Java generics, introduced in Java 5.
- Development driven by perceived shortcomings of the Java programming language

Scala usage

- Scala is widely used in many large high-profile companies and organisations — it is now a mainstream general purpose language
- Many large high-traffic websites are built with Scala (eg. Twitter, Foursquare, LinkedIn, Walmart, Gilt, The Guardian, etc.)
- Scala is widely used as a Data Science and Big Data platform due to its speed, robustness, concurrency, parallelism and general scalability (in addition to seamless Java interoperability)
- Scala programmers are being actively recruited by many high profile data science teams

Static versus dynamic typing, compiled versus interpreted

- It is fun to quickly throw together a few functions in a scripting language without worrying about declaring the types of anything
- But for any code you want to keep or share with others you end up adding lots of boilerplate argument checking code that would be much cleaner, simpler and faster in a statically typed language
- Scala has a strong type system offering a high degree of compile-time checking, making it a safe and efficient language
- By maximising the work done by the compiler at build time you minimise the overheads at runtime
- Coupled with type inference, statically typed code is actually more concise than dynamic code

Functional versus imperative programming

- Functional programming offers many advantages over other programming styles
- In particular, it provides the best route to building scalable software, in terms of both program complexity and data size/complexity
- Scala has good support for functional programming, including immutable named values, immutable data structures and for-comprehensions
- Many languages are attempting to add functional features retrospectively (eg. lambdas in C++, lambdas, streams and the option monad in Java 8, etc.)
- Many new and increasingly popular languages are functional, and several are inspired by Scala (eg. Apple's Swift is essentially a cut down Scala, as is Red Hat's Ceylon)

Using Scala

- Scala is completely free and open source — the entire Scala software distribution, including compiler and libraries, is released under a BSD license
- Scala is platform independent, running on any system for which a JVM exists
- Easy to install `Scala` and `scalac`, the Scala compiler, but not really necessary
- The "simple build tool" for Scala, `sbt`, is all that is needed to build and run most Scala projects, and this can be bootstrapped from a 1M Java file, `sbt-launch.jar`

Versions, packages, platform independence, cloud

- All dependencies, including Scala library versions, and associated "packages", can be specified in a `sbt` build file, and pulled and cached at build time — no need to "install" anything, ever — most basic library/package versioning problems simply disappear
- This is particularly convenient when scaling out to virtual machines and lightweight containers (such as Docker) in the cloud
- All that is required is a container with a JVM, and you can either build from source or push a redistributable binary package

IDEs

- There are several very good IDEs for Scala
- In general, IDEs are much better and much more powerful for statically typed languages — IDEs can do lots of things to help you when programming in a statically typed language which simply aren't possible when using a dynamically typed language
- I use the "Scala IDE" (http://scala-ide.org/), which is based on Eclipse, but other possibilities, including IntelliJ (https://www.jetbrains.com/idea/features/scala.html), which some prefer
- If you use an IDE, your code will (almost) always compile first time

Reproducible research

- Reproducible research is very important — others should be able to run your code and reproduce your results
  - Many within the statistics community have come to associate reproducible research with dynamic documents and literate programming — nothing could be further from the truth!
  - Can more-or-less guarantee that R code and documentation written with the latest trendy literate programming framework will not build and run in 2 years time (due to incompatible library and package version changes)...
- `sbt` build files specify the particular versions of Scala and any associated dependencies required, and so projects should build and run reproducibly without issues for many years
- Standard code documentation format, Scaladoc, an improved Scala version of Javadoc, and standard testing frameworks such as ScalaTest.

Data structures and parallelism

- Scala has an extensive "Collections framework" (http://docs.scala-lang.org/overviews/collections/overview.html), providing a large range of data structures for almost any task (Array, List, Vector, Map, Range, Stream, Queue, Stack, ...)
- Most collections available as either a mutable or immutable version — idiomatic Scala code favours immutable collections
- Most collections have an associated parallel version, providing concurrency and parallelism "for free" (examples later)

Functional approaches to concurrency and parallelisation

- Functional languages emphasise immutable state and referentially transparent functions
- Immutable state, and referentially transparent (side-effect free) declarative workflow patterns are widely used for systems which really need to scale (leads to naturally parallel code)
- Shared mutable state is the enemy of concurrency and parallelism (synchronisation, locks, waits, deadlocks, race conditions, ...) — by avoiding mutable state, code becomes easy to parallelise
- The recent resurgence of functional programming and functional programming languages is partly driven by the realisation that functional programming provides a natural way to develop algorithms which can exploit multi-core parallel and distributed architectures, and efficiently scale

Category theory

Dummies guide:

- A "collection" (or parametrised "container" type) together with a "map" function (defined in a sensible way) represents a functor
- If the collection additionally supports a (sensible) "apply" operation, it is an applicative
- If the collection additionally supports a (sensible) "flattening" operation, it is a monad (required for composition)
- For a "reduce" operation on a collection to parallellise cleanly, the type of the collection together with the reduction operation must define a monoid (must be an associative operation, so that reductions can proceed in multiple threads in parallel)

Scala ecosystem

- Akka — actor-based concurrency framework (inspired by Erlang)
- Spark — scalable analytics library, including some ML (from Berkeley AMP Lab)
- Algebird — abstract algebra (monoid) support library (from Twitter)
- Scalding — cascading workflow library (from Twitter)
- Storm — streaming analytics library (from Twitter)
- Scalaz — category theory types (functors, monads, etc.)
- Breeze — scientific and numerical library (including non-uniform random number generation and numerical linear algebra)
- Saddle — data library (data frames, etc.)

Large ecosystem of software libraries and developers using Scala in the big data space...

Breeze

- Breeze is a Scala library for scientific and numerical computing — https://github.com/scalanlp/breeze
- Includes all of the usual special functions, probability distributions, (non-uniform) random number generators, matrices, vectors, numerical linear algebra, optimisation, etc.
- For numerical linear algebra it provides a Scala wrapper over netlib-java, which calls out to a native optimised BLAS/LAPACK if one can be found on the system (so, will run as fast as native code), or will fall back to a Java implementation if no native libraries can be found (so that code will always run)
- Blog post: Brief introduction to Scala and Breeze for statistical computing

Example: Monte Carlo (0)

- Integrate the standard normal PDF over $[-5, 5]$ by simple uniform rejection sampling on the rectangle $[-5, 5] \times [0, 0.5]$.
- First import some objects into the namespace:

```
import scala.math._
import breeze.stats.distributions.Uniform
import breeze.linalg._
import Scala.annotation.tailrec

// R-like functions for Uniform random numbers
def runif(l: Double,u: Double) = Uniform(l,u).sample
def runif(n: Int, l: Double, u: Double) =
    DenseVector[Double](Uniform(l,u).sample(n).toArray)
```

Example: Monte Carlo (1)

The idiomatic Breeze solution would be to use vectorised code similar to that you would use to solve the problem in R or Python

```
val N=100000
def f(x: Double): Double =
     math.exp(-x*x/2)/math.sqrt(2*Pi)

def mc1(its: Int): Int = {
  val x = runif(its,-5.0,5.0)
  val y = runif(its,0.0,0.5)
  val fx = x map {f(_)}
  sum((y :< fx) map {xi => if (xi == true) 1 else 0})
}
println(5.0*mc1(N)/N)
```

This works fine, but will exhaust available RAM for sufficiently large N

Example: Monte Carlo (2)

In this case, better, faster and more memory efficient to use a tail call (which will actually compile down to a while loop)

```
def mc2(its: Long): Long = {
  @tailrec def mc(its: Long,acc: Long): Long = {
    if (its == 0) acc else {
      val x = runif(-5.0,5.0)
      val y = runif(0.0,0.5)
      if (y < f(x)) mc(its-1,acc+1) else
                                    mc(its-1,acc)
    }
  }
  mc(its,0)
}
println(5.0*mc2(N)/N)
```

This works fine for any N and is faster than the previous version

Example: Monte Carlo (3)

Trivial to parallelise the previous version by mapping it over a parallel collection

```
def mc3(its: Long): Long = {
  val NP = 8 // Max number of threads to use
  val N = its/NP // assuming NP|its
  (1 to NP).toList.par.map{x => mc2(N)}.sum
}
println(5.0*mc3(N)/N)
```

This kind of code is typically more-or-less as fast (sometimes faster!) than native C/MPI code...

Example: Monte Carlo (4)

Timings for 10^7 iterations on my laptop:

```
> run
[info] Running MonteCarlo
Running with 10000000 iterations
Idiomatic vectorised solution
0.999661
time: 2957.277005ms
Fast efficient (serial) tail call
1.000262
time: 1395.82964ms
Parallelised version
1.000463
time: 337.361038ms
Done
```

Example: linear regression (1)

- Read a CSV file with 3 columns — 2 numeric and one binary factor — regress first (numeric) column on other two — plots and summary statistics
- This is simple exploratory data analysis for a small data set — I would use R for this!
- No reason why Scala can't do this kind of thing, but not the interest of current Scala developers
- Two standard implementations of multiple linear regression in Scala that I know of:
  - Breeze: `leastSquares` (LAPACK call to `dgels`)
  - Spark MLlib: `LinearRegressionWithSGD` (stochastic gradient descent)
- Both work perfectly well for obtaining least squares estimates of regression coefficients, but not much in the way of statistical diagnostics.

Example: linear regression (2)

- Not one to shirk a challenge, I spent a couple of evenings writing a few classes for linear regression, building on Saddle and Breeze...

```
import regression._
import scala.math.log
import org.saddle.io._
import FrameUtils._

val file = CsvFile("data/regression.csv")
val df = CsvParser.parse(file).withColIndex(0)
println(df)
framePlot(getCol("Age", df), getCol("OI", df))
```

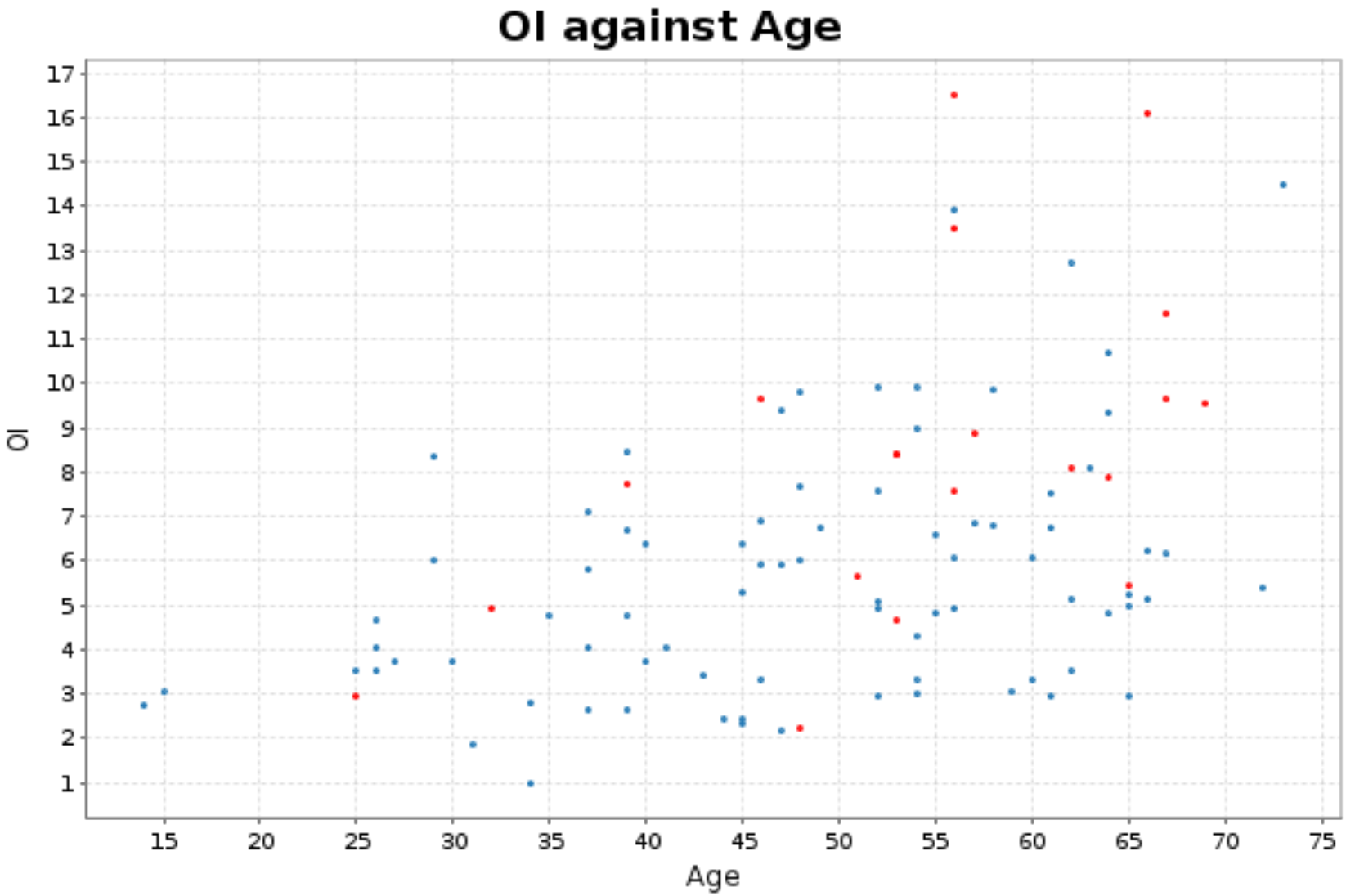
Example: linear regression (3)

```
scala>  val df = CsvParser.parse(file).withColIndex(0)
df: org.saddle.Frame[Int,String,String] =
[101 x 3]
          OI  Age     Sex
        ----  --- ------
  1 ->     5   65 Female
  2 ->  3.75   40 Female
  3 ->   7.6   52 Female
  4 ->  2.45   45 Female
  5 ->   5.4   72 Female
...
 97 ->  8.89   57   Male
 98 ->  16.5   56   Male
 99 ->  4.65   53   Male
100 ->  13.5   56   Male
101 ->  16.1   66   Male

scala>
```

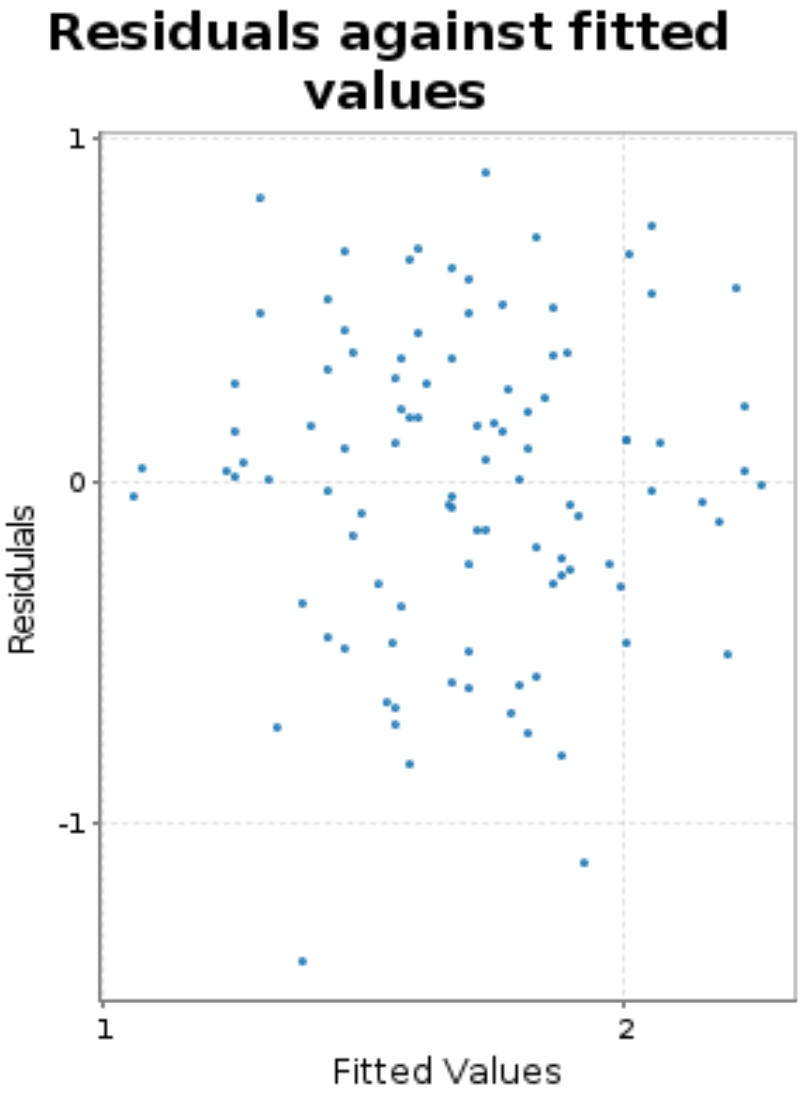
Example: linear regression (4)

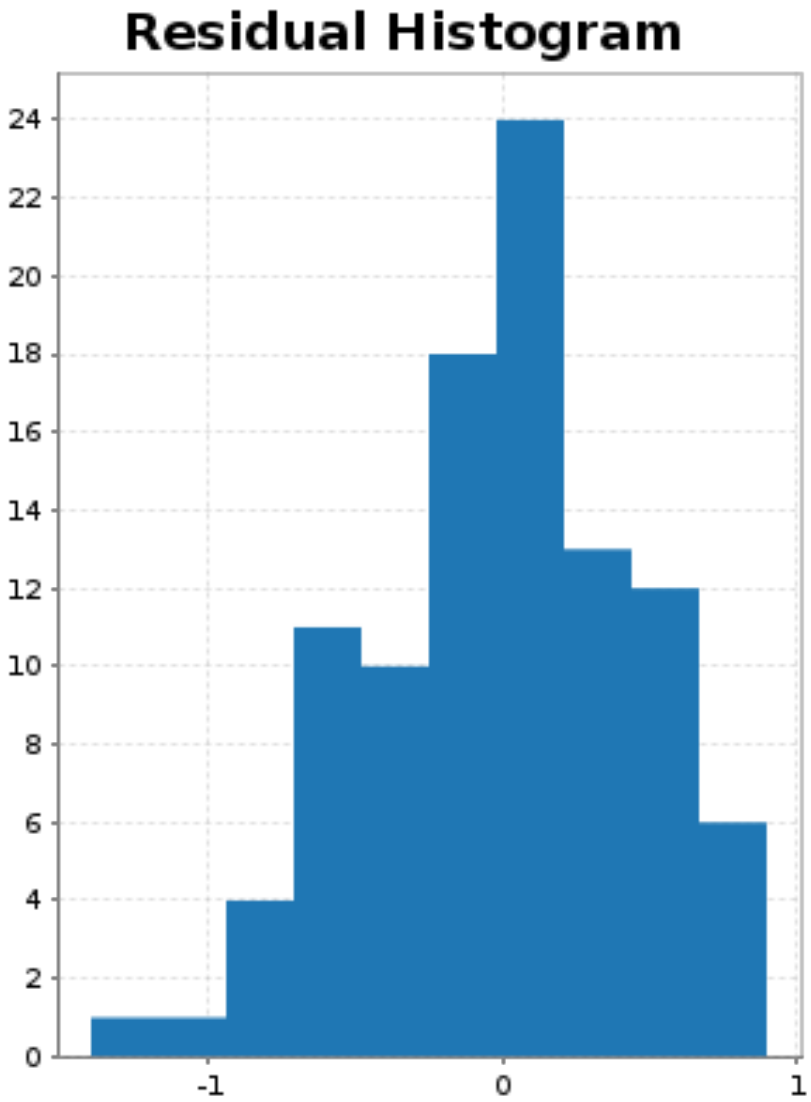
```
val df2 = frameFilter(df, getCol("Age", df), _ > 0.0)
println(df2)
val oi = getCol("OI", df2)
val age = getCol("Age", df2)
val sex = getFactor("Sex", df2)
framePlot(age, oi, sex).saveas("data.png")

val y = oi.mapValues { log(_) }
val m = Lm(y, List(age, sex))
println(m)
m.plotResiduals.saveas("resid.png")

val sum = m.summary
println(sum)
```

Example: linear regression (5)



Example: linear regression (6)





Example: linear regression (7)

```
scala> m.summary
res6: regression.LmSummary =
Residuals:
[5 x 1]
   Min -> -1.4005
    LQ -> -0.2918
Median ->  0.0308
    UQ ->  0.3211
   Max ->  0.8979
Coefficients:
[3 x 4]
                   OI      SE  t-val  p-val
               ------ ------ ------ ------
(Intercept) -> 0.8292 0.1777 4.6661 0.0000
        Age -> 0.0162 0.0035 4.6027 0.0000
    SexMale -> 0.3189 0.1157 2.7567 0.0058
Model statistics:
[6 x 1]
          RSS -> 20.0999
          RSE ->  0.4552
           df -> 97.0000
    R-squared ->  0.2621
```

Example: linear regression (8)

So we see that Scala could be just as convenient as R and similar languages for basic exploratory data analysis and visualisation, but this would require a little effort from people interested in having this functionality:

- Port the `statsmodels` package from Python
- Port `ggplot` (or similar viz framework)
- Re-write Saddle to have its data frames backed by Breeze matrices, and incorporate into Breeze

Each of these could be accomplished with relatively little effort (eg. a GSoC student or two) if the demand exists

Example: Gibbs sampler

```
class State(val x: Double,val y: Double)

def nextIter(s: State): State = {
     val newX=rngG.nextDouble(3.0,(s.y)*(s.y)+4.0)
     new State(newX,
          rngN.nextDouble(1.0/(newX+1),1.0/sqrt(2*newX+2)))
}

def nextThinnedIter(s: State,left: Int): State = {
   if (left==0) s
   else nextThinnedIter(nextIter(s),left-1)
}

def genIters(s: State,current: Int,stop: Int,thin: Int): State = {
     if (!(current>stop)) {
        println(current+" "+s.x+" "+s.y)
        genIters(nextThinnedIter(s,thin),current+1,stop,thin)
     }
     else s
}
```

Example: Parallel particle filter

```
(th: P) => {
  val x0 = simx0(n, t0, th).par
  @tailrec def pf(ll: LogLik, x: ParVector[S], t: Time,
                  deltas: List[Time], obs: List[O]): LogLik =
    obs match {
      case Nil => ll
      case head :: tail => {
        val xp = if (deltas.head == 0) x else
                 (x map { stepFun(_, t, deltas.head, th) })
        val w = xp map { dataLik(_, head, th) }
        val rows = sample(n, DenseVector(w.toArray)).par
        val xpp = rows map { xp(_) }
        pf(ll + math.log(mean(w)), xpp, t + deltas.head,
                                        deltas.tail, tail)
      }
    }
  pf(0, x0, t0, deltas, obs)
}
```

Summary

- Strong arguments can be made that a language to be used as a platform for serious statistical computing should be general purpose, platform independent, functional, statically typed and compiled
- For basic exploratory data analysis, visualisation and model fitting, R works perfectly well (currently better than Scala)
- Scala is worth considering if you are interested in any of the following:
  - Writing your own statistical routines or algorithms
  - Working with computationally intensive (parallel) algorithms
  - Working with large and complex models for which an out-of-the-box solution doesn't exist in R
  - Working with large data sets (big data)
  - Integrating statistical analysis workflow with other system components (including web infrastructure, relational databases, no-sql databases, etc.)

Links and further reading

- Scala: http://www.scala-lang.org/
- Scala on wikipedia: http://en.wikipedia.org/wiki/Scala_(programming_language)
- My blog: http://darrenjw.wordpress.com/
  - Scala as a platform for statistical computing and data science
  - Brief introduction to Scala and Breeze for statistical computing
  - A functional Gibbs sampler in Scala
  - Parallel Monte Carlo using Scala
- This talk and code examples: https://github.com/darrenjw/statslang-scala (also merged into https://github.com/csgillespie/statslang)